Certifying Solvers for Clique and Maximum Common (Connected) Subgraph Problems

Stephan Gocht[1,2], Ross McBride[3], Ciaran McCreesh[3(✉)], Jakob Nordström[1,2], Patrick Prosser[3], and James Trimble[3]

[1] Lund University, Lund, Sweden
stephan.gocht@cs.lth.se
[2] University of Copenhagen, Copenhagen, Denmark
jn@di.ku.dk
[3] University of Glasgow, Glasgow, Scotland
{ciaran.mccreesh,patrick.prosser}@glasgow.ac.uk,
j.trimble.1@research.gla.ac.uk

Abstract. An algorithm is said to be *certifying* if it outputs, together with a solution to the problem it solves, a proof that this solution is correct. We explain how state of the art maximum clique, maximum weighted clique, maximal clique enumeration and maximum common (connected) induced subgraph algorithms can be turned into certifying solvers by using pseudo-Boolean models and cutting planes proofs, and demonstrate that this approach can also handle reductions between problems. The generality of our results suggests that this method is ready for widespread adoption in solvers for combinatorial graph problems.

1 Introduction

McConnell et al. [40] argue that all algorithm implementations should be *certifying*: that is, along with their output, they should produce an easily verified proof that the output is correct. Given the relative frequency of bugs in constraint programming (CP) solvers and in dedicated algorithms for hard combinatorial problems [7,12,25,42], it would be desirable to see certification becoming a social requirement for all new solvers—as has already happened in the Boolean satisfiability community through proof logging formats such as RUP [27], TraceCheck [5], DRAT [29,30,74], LRAT [13] and GRIT [14]. A proof log is a particular kind of certificate which records the steps taken by a solver in such a way that the correctness of each step can easily be checked, given that

The first and fourth authors were funded by the Swedish Research Council (VR) grant 2016-00782. The fourth author was also supported by the Independent Research Fund Denmark (DFF) grant 9040-00389B. The third, fifth and sixth authors were supported by the Engineering and Physical Sciences Research Council [grant numbers EP/P026842/1 and EP/M508056/1]. Some code development used resources provided by the Swedish National Infrastructure for Computing (SNIC).

H. Simonis (Ed.): CP 2020, LNCS 12333, pp. 338–357, 2020.
https://doi.org/10.1007/978-3-030-58475-7_20

all previous steps are known to be correct; the intent is that verifying a proof log should be very simple, even if solvers carry out complex reasoning.

Until recently, it was generally assumed that proof logging for more powerful CP-style solvers would require either very complicated (and hard to verify) certificates that must be aware of every kind of propagation performed by every constraint [72], or an exponential slowdown [22]. However, it has recently been shown that reasoning over pseudo-Boolean formulae can compactly express all-different reasoning [19], as well as all of the reasoning carried out by state-of-the-art subgraph isomorphism solvers [26], despite pseudo-Boolean reasoning not knowing anything about Hall sets, matchings, vertices, degrees, or paths.

The general idea behind this proof logging is that a constraint satisfaction problem (or other hard problem) is compiled to a pseudo-Boolean (PB) formula—that is, a 0–1 integer linear program. Then, either a witness of satisfiability is provided, or a proof showing that the PB formula implies $0 \geq 1$ is given. (Optimisation and enumeration problems are also supported.) The proofs of unsatisfiability demonstrated so far have consisted of a mix of "reverse unit propagation" (RUP) steps [19,24] to describe the backtracking search tree produced by the solver, and assistance in deriving any information used by propagators that is not immediately apparent to unit propagation (such as Hall sets and Hall violators). In this work, we report that this approach can be used in a more general way to obtain certifying algorithms (with proofs that can be checked by the VeriPB verifier [19]) for a range of other hard problems:

- We show how a wide variety of maximum clique algorithms from the literature can all be enhanced with proof logging, using very similar proof techniques. We also explain how to adapt this proof logging method to cover the inference used by a state of the art maximum weight clique solver. Finally, we discuss certification for all maximal clique enumeration algorithms.
- We also demonstrate proof logging for a state of the art CP-style maximum common induced subgraph algorithm, including for the connected variant of the problem.
- Finally, we look at a reduction from maximum common induced subgraph to maximum clique, which outperforms CP approaches on certain graph classes. We show that this reduction can be expressed inside the proof log, so we can take a PB model that was generated for the CP encoding, but then provide a proof from a clique algorithm—this is a bit like channelling [8], but for proofs. There are also clique-like algorithms with a propagator to enforce connectedness. Because the reduction can be viewed as a bijection, we can continue to express the connectedness constraint only on the CP encoding (where it is much easier to understand than on the clique encoding), but still validate clique-like proofs.

Our main conclusion is that proof logging using pseudo-Boolean reasoning is general and powerful enough to concisely describe the inference used in a wide range of combinatorial graph algorithms. Although the current implementation does not scale well enough to deal with the largest instances, it can already

be used to provide, for the first time, fully verifiable proofs of correctness for some highly nontrivial medium-sized instances. Also, even if the overhead is currently too high to have proof logging switched on by default in production, it provides an excellent tool for debugging of nontrivial optimisation techniques during solver development. This is because incorrectly implemented steps are likely to lead to incorrect proofs, which can be detected even when the results produced by the solver happen to be correct. We believe this tool is mature enough for widespread adoption, and that requiring all solvers be able to output proofs would be a natural and desirable step to increase the confidence in the correctness of state-of-the-art solvers.

2 Clique Problems

A clique in a graph is a set of vertices, where every vertex in this set is adjacent to every other in this set. The problem of finding a maximum-sized clique in a graph is broadly applicable, and there are many dedicated solvers for the problem (which we will discuss below). However, as McCreesh et al. [42] note, at least some of these solvers are buggy—including the one [35] which was used as a subcomponent by the winner of the 2019 PACE Implementation Challenge [28]. We therefore begin with a worked example, showing how a machine-verifiable proof could be constructed to demonstrate and prove the correctness of a solution for a simple maximum clique instance.

Consider the graph in Fig. 1. To prove that the maximum clique size of this graph is four, we have to show two things: that it has a clique of four vertices, and that there is no larger clique. To do so, we use the VeriPB proof verifier, which takes two files as its input: a pseudo-Boolean model in the standard OPB format [55], and a proof log which provides a verifiable solution to this model. Therefore, our first step is to encode the problem of finding a maximum clique in this graph as a pseudo-Boolean model. We have a 0–1 variable x_i for each vertex i in the graph, an objective which is to maximise the sum of the vertices taken, and for every non-adjacent pair of vertices, a constraint saying they cannot both be taken simultaneously. In OPB format, this looks like:

```
* #variable= 12 #constraint= 41
min: -1 x1 -1 x2 -1 x3 -1 x4 ...and so on... -1 x11 -1 x12 ;
1 ~x3 1 ~x1 >= 1 ;
1 ~x3 1 ~x2 >= 1 ;
1 ~x4 1 ~x1 >= 1 ;
* ...and a further 38 similar lines for the remaining non-edges
```

Here the first line is a special header comment, the second line specifies that the objective is to minimise $\sum_{i=1}^{12} -x_i$ (i.e. to maximise the number of vertices selected, $\sum_{i=1}^{12} x_i$, but OPB supports only minimisation), and subsequent lines specify constraints. An expression like `1 ~x3 1 ~x1 >= 1` corresponds to the linear inequality $1\overline{x}_3 + 1\overline{x}_1 \geq 1$, where the overline means negation, $\overline{x}_i = 1 - x_i$.

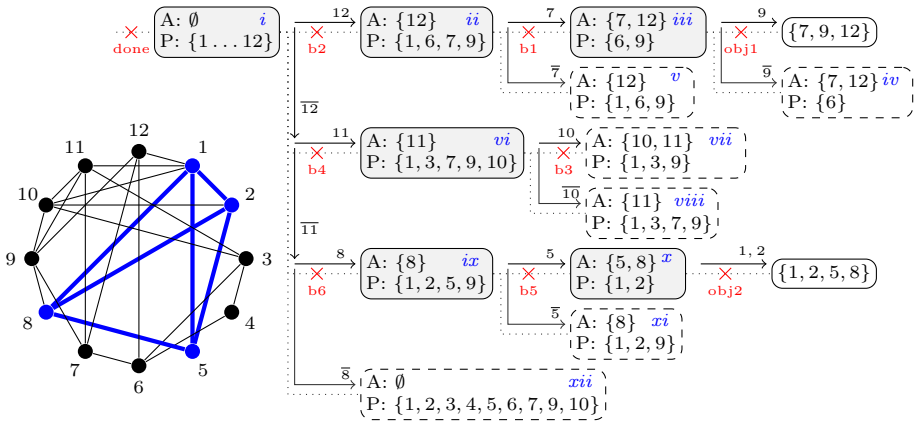


Fig. 1. On the left, a graph, with a 4-vertex clique highlighted. On the right, an illustration of the proof tree used in our worked example to show that this clique is maximum. The solid arrows show the solver's view of the search tree, and are labelled either with a vertex number being accepted or an overlined vertex number being rejected. Shaded boxes represent states in the search tree where we have accepted the vertices labelled "A" and can potentially accept the ones labelled "P", dashed boxes represent states that are eliminated by a bound, and clear boxes are candidate solutions. Roman numerals denote states discussed in the text. Dotted lines show the search tree used by the proof: the crosses with labels correspond to statements that justify a backtrack.

Note the simplicity of this encoding. This is important: the proof we will produce is expressed in terms of this encoding, and because this process is not formally verified, any errors in the encoding could potentially lead to a proof which "proves the wrong thing" being accepted.[1]

Now we move on to the proof. We could produce proofs of the decision problems for 4- and 5-cliques, but the VeriPB format also allows us to verify a branch-and-bound search directly. We now give such a proof, tracing a possible (and intentionally not very good) algorithm execution as we do so. The proof log must begin with a header, as follows (asterisk lines are comments):

```
pseudo-Boolean proof version 1.0
* load the objective function, and the 41 model constraints
f 41 0
```

Typically, maximum clique algorithms maintain two sets during search: a set of accepted vertices, A, which is always a growing clique, and a set of possible vertices P, each of which is adjacent to every vertex in A. Rather than a binary branching scheme, we will iterate over each vertex in P in turn and first accept

[1] We are not aware of any obstacles for providing formal verification for this translation step. However, since this translation is so simple, in this paper we focus on the more challenging task of formally verifying the correctness of solvers' reasoning.

that vertex, then reject it and accept a second vertex instead. We will carry out a typical depth-first branch and bound search, with a rather ad hoc bound for illustration purposes. Our solver will begin in the state labelled i in Fig. 1, with no vertices accepted and every vertex being possible.

Suppose our solver first branches by deciding to accept vertex 12. Then by adjacency, only vertices 1, 6, 7 and 9 remain acceptable; we are in state *ii*. Suppose now we also accept vertex 7. This leaves vertices 6 and 9 possibly to be accepted; we are in state *iii*. We accept vertex 9, which is not adjacent to 6. We have found a maximal clique with three vertices. We therefore record this in the proof log, using an "o" rule. This rule tells the verifier to check that the solution we specified is in fact feasible, and then to create a new constraint, $\sum_{i=1}^{12} x_i \geq 4$, saying that any future solution must be better; this constraint also allows us to backtrack, which is marked as "obj1" in Fig. 1. We log this as:

```
o x7 x9 x12
```

We are now back to having accepted vertices 7 and 12, but now only 6 remains possible; this is state *iv*. Now that we have introduced a new constraint saying we must set at least four variables to true, it is obvious to a human that we are at a dead-end and must backtrack. We now have two options: we can explicitly justify why we can backtrack by deriving a new constraint manually, or we can rely upon some help from the proof verifier.

To derive the constraint manually, we would proceed as follows. If we sum the objective line, every non-adjacency constraint involving x_7 or x_{12}, and the non-adjacency constraint involving x_6 and x_9, we get $\overline{x}_2 + \overline{x}_3 + \overline{x}_4 + \overline{x}_5 + 6\overline{x}_7 + \overline{x}_8 + \overline{x}_{10} + 6\overline{x}_{12} \geq 7$. Now, for any variable x_i, we have an axiom $x_i \geq 0$. By also adding these axioms for each $i \in \{2, 3, 4, 5, 8, 10\}$, and normalising by using the fact that $x_i + \overline{x}_i = 1$, the sum reduces to $6\overline{x}_7 + 6\overline{x}_{12} \geq 1$, which we may then divide by 6 to get $\overline{x}_7 + \overline{x}_{12} \geq 1$ as desired. We *could* express these steps explicitly in the proof log—and we *could* also explain an algorithm a solver could use to know exactly which constraints to sum together and what constant to divide by. But fortunately, there is an easier approach. By using a "u" rule, we may tell the proof verifier to introduce a new constraint which is "obviously" true, given what it knows already. So, we may simply assert:

```
u 1 ~x12 1 ~x7 >= 1 ;
```

and the proof verifier will work out the rest. It is able to do this because this new constraint follows by *reverse unit propagation* (RUP) [19,24]: that is, if we add the negation of this constraint and unit propagate,[2] then contradiction follows without search. We may verify this: the negation of the constraint $\overline{x}_{12} + \overline{x}_7 \geq 1$ is $x_{12} + x_7 \geq 2$. From this, unit propagation infers that both x_7 and x_{12} are 1. Then, using the non-adjacency constraints, all variables except x_6 and x_9

[2] In a PB setting, unit propagation is equivalent to achieving integer bounds consistency [9] on all constraints. This is identical to SAT unit propagation on clausal constraints, but is stronger in general.

will unit propagate to 0. Now, looking at the new objective constraint, we have to set at least four variables to 1, so x_6 and x_9 must both be 1. However, vertices 6 and 9 are non-adjacent, and so there is a constraint forbidding them both to be 1. Thus, RUP can derive a contradiction, and can safely add the asserted constraint—without our hypothetical solver authors having to perform any complicated bookkeeping. This new constraint is labelled "b1" in Fig. 1.

Our solver is now back in the state that it has accepted vertex 12, and it has vertices 1, 6, and 9 to choose from; this is state *v*. Observe that vertices 1 and 6 are non-adjacent, and so it is not possible to make a 4-clique using vertex 12 plus a subset of these vertices. We may therefore backtrack—again, this fact follows using RUP. We label this "b2" in the figure, and log it as:

```
u 1 ~x12 >= 1 ;
```

We are now back at the top of the search tree, having rejected vertex 12 entirely. Suppose we accept vertex 11 next: this leaves vertices 1, 3, 7, 9, and 10 as possibilities, state *vi*. Then suppose we accept vertex 10, leaving vertices 1, 3, and 9 as possibilities, state *vii*. Note that none of these vertices are adjacent, and so we may select at most one of these. To a human, it is now obvious that we may backtrack, but we must give the proof verifier a little help. Before we can use a RUP rule to backtrack, we must derive an at-most-one rule showing that $x_1 + x_3 + x_9 \leq 1$. We may do this as follows:

```
p 1 2 * 19 + 21 + 3 d
p 42 47 +
u 1 ~x11 1 ~x10 >= 1 ;
```

However, these two "p" lines are not easy to read, as expressed: some of the numbers are literal constants, some refer to constraints in the model file, and some refer to constraints we have generated earlier in the verification process. For this discussion, we will therefore take a few liberties with the proof format in our running example. Instead of writing "42" for the objective constraint (which got that number because it was the first introduced constraint, and there are 41 model constraints before it), we will write `obj1`. Similarly, rather than writing 19 to refer to the model constraint $\overline{x}_1 + \overline{x}_9 \geq 1$, we will write `nonadj1_9`. Finally, we will use the notation $\rightsquigarrow$ `name` to give a name to the result of a rule that we will refer to later on in the proof, or to refer to a point in Fig. 1. After this, any remaining numbers are literal constants. Thus, the above snippet becomes:

```
* at most one [ x1 x3 x9 ]
p nonadj1_3 2 * nonadj1_9 + nonadj3_9 + 3 d                    ~> tmp1
p obj1 tmp1 +
u 1 ~x11 1 ~x10 >= 1 ;                                          ~> b3
```

and we may explain the two "p" rules more easily. In the cutting planes proof system for pseudo-Boolean formulae [11] upon which VeriPB is based, we can add together existing constraints, multiply existing constraints by a non-negative

integer constant, and divide existing constraints by a positive integer constant. A "p" rule expresses these steps in reverse Polish notation. The first "p" rule multiplies the non-adjacency constraint $\overline{x}_1 + \overline{x}_3 \geq 1$ by 2 to get $2\overline{x}_1 + 2\overline{x}_3 \geq 2$, adds two more non-adjacency constraints to get $3\overline{x}_1 + 3\overline{x}_3 + 2\overline{x}_9 \geq 4$, and divides this by 3 to get the at-most-one constraint $\overline{x}_1 + \overline{x}_3 + \overline{x}_9 \geq 2$. The second "p" rule adds this to our objective constraint, $\sum_{i=1}^{12} x_i \geq 4$, to show that the remaining nine variables must sum to at least 3. This is now sufficient for reverse unit propagation to justify backtracking step "b3".

A very similar argument allows us to backtrack again: having accepted vertex 11, and rejected vertices 10 and 12, we may pick at most one of vertices 1, 3, and 7, plus possibly vertex 9 (state *viii*). We must help the verifier by generating another at-most-one constraint:

```
* at-most-one [ x1 x3 x7 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d                    ⇝ tmp2
p obj1 tmp2 +
u 1 ~x11 >= 1 ;                                                 ⇝ b4
```

We are back at the top of search. Having rejected vertices 11 and 12, if we now branch accepting vertex 8 (state *ix*), and then vertex 5 (state *x*), the remaining possible vertices 1 and 2 can both be added to form a clique. We thus log this as a solution, which generates a new objective constraint $\sum_{i=1}^{12} x_i \geq 5$.

```
o x1 x2 x5 x8                                                   ⇝ obj2
u 1 ~x8 1 ~x5 >= 1 ;                                            ⇝ b5
```

Backtracking to the top of the search tree from state *xi* can be justified by observing that we may pick at most one of vertices 1 and 9:

```
p obj2 nonadj1_9 +
u 1 ~x8 >= 1 ;                                                  ⇝ b6
```

Finally, having rejected vertices 8, 11, and 12 at the top of search, we are in state *xii*, and the remaining nine vertices can be partitioned into independent sets to create three at-most-one constraints. To allow RUP to unset all nine vertices, we will combine these constraints incrementally, as follows.

```
* at-most-one [ x1 x3 x7 ] [ x2 x4 x9 ] [ x5 x6 x10 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d                    ⇝ tmp3
p obj2 tmp3 +
p nonadj2_4 2 * nonadj2_9 + nonadj4_9 + 3 d                    ⇝ tmp4
p obj2 tmp3 + tmp4 +
p nonadj5_6 2 * nonadj5_10 + nonadj6_10 + 3 d                  ⇝ tmp5
p obj2 tmp3 + tmp4 + tmp5 +
```

The proof terminates by asserting that we have proved unsatisfiability—that is, there is nothing remaining that can beat the best solution we have found. This is done through a RUP check for contradiction, i.e. that $0 \geq 1$, followed by a "c" rule to terminate the proof.

```
u >= 1 ;                                                    ⇝ done
c done 0
```

Having produced this log, we may now hand it and the associated pseudo-Boolean model file to VeriPB, which will successfully verify it.

There is one other important detail that we have omitted from this proof: in practice, it is extremely helpful to the verifier if we delete temporary constraints when they are used, as well as intermediate backtracking constraints after we have backtracked further up the tree. (This is also crucial for the performance of proof logging for SAT [74].) VeriPB supports both deletion of numbered constraints, and a notion of "levels" which allow all constraints generated below a certain depth to be deleted simultaneously.

2.1 Maximum Clique Algorithms in General

The majority of maximum clique algorithms that are aimed at hard, dense graphs make use of backtracking search with branch and bound [4,33,35–37,39,44,51,53,57,58,60,62,66–69,71]. The inference on adjacency performed by all of these algorithms is straightforward, with all of the cleverness being in branching and how bounds are computed [1]. We may therefore produce proof logs for all of these algorithms using only RUP, logging of solutions as they are found, and some additional help for the bounds.

Colour Bounds. If a graph can be coloured using k colours (where adjacent vertices must be given different colours) then it cannot contain a clique of more than k vertices. Producing an optimal colouring is hard (and typically harder in practice than finding a maximum clique), but various greedy methods exist, and have been used to give a dynamic bound during search inside clique algorithms. Suppose we have, after branching, our set of accepted vertices A, a set of undecided vertices P, and have already found a clique of n vertices. If $c(P)$ is the number of colours used in some legal colouring of the subgraph induced by P, then if $|A| + c(P) \leq n$, we can immediately backtrack.

Using cutting planes, if we are given a colouring then it is easy to produce a proof that this bound is valid. By definition, for each pair of vertices in a given colour class, the PB model must have a constraint saying that both vertices cannot be taken simultaneously (because they do not have an edge between them). As we saw in the worked example, it is routine to combine these constraints into an at-most-one constraint, using a single sequence of arithmetic operations that mentions each pairwise constraint only once. We can then sum these new at-most-one constraints, add them to the objective constraint, and the rest of the work follows by unit propagation.

Incremental Colour Bounds. Producing a colouring can be relatively expensive. In order to reduce the number of colourings needed, many solvers reuse colourings. Suppose we have produced colour classes $C_1, \ldots, C_c$. Instead of making a single branching decision, we may branch on accepting each vertex in colour class

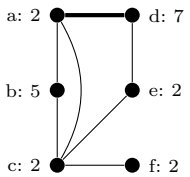
C_c in turn first, followed by those in C_{c-1}, then C_{c-2} and so on, stopping after we have visited only $n - |A| + 1$ colour classes. Ideally, in a proof log, we would not have to produce individual statements to justify not exploring each vertex in each remaining colour class. This is indeed possible: we derive an at-most-one constraint for colour class C_1, and remember its number ℓ_1. We then add this constraint to the objective constraint. Next, we derive an at-most-one constraint for colour class C_2, add this to ℓ_1, and remember its number ℓ_2. Now we sum the objective constraint, ℓ_1, and ℓ_2. We continue until we reach a colour class which was used for branching—again, the worked example made use of this.

Other changes to the details of how colour bounds are produced has formed a substantial line of work in maximum clique algorithms [33,51,53,57,58,62,66–69]. However, proof logging is completely agnostic to this: we care only that we have a valid colouring, and do not need to understand any of the details of the algorithm that produced it.

Stronger Bounds. Even when a good colouring is found, colour bounds can be quite weak in practice. Some clique solvers identify subsets of k colour classes which cannot form a clique of k vertices. For example, San Segundo et al. [60] will find certain cases where there is a pair of colour classes C_1 and C_2, together with a vertex v, such that no triangle exists using v and a vertex each from C_1 and C_2, and uses this to reduce the bound by one for vertex v. If such a case is identified, RUP is sufficient to justify it. Similarly, because pseudo-Boolean unit propagation is at least as strong as SAT unit propagation, bounds using MaxSAT reasoning on top of colour classes [35–37] are also easily justified.

Algorithm Features Not Affecting Proof Logging. Maximum clique algorithms have used a variety of different search orders [44]; as with the details of how colourings are produced, these details are irrelevant for proof logging. Similarly, bit-parallelism [59,61] has no effect on proof logging; thread-parallelism [16,43,45] remains to be seen, but since proof logging is largely I/O bound, it is likely that gains from multi-core parallelism will be lost on current hardware when logging. And finally, running a local search algorithm and "priming" the branch and bound algorithm with a strong initial incumbent [4,39,71] requires only that the new incumbent be logged before the search starts, regardless of how that incumbent was found.

Implementation. We implemented proof logging for the dedicated clique solver which is included in the Glasgow Subgraph Solver [48], and tested it on a system with dual Intel Xeon E5-2697A v4 CPUs, 512 GBytes RAM, and a conventional hard drive, running Ubuntu 18.04. Without proof logging, this solver is able to solve 59 of the 80 instances from the second DIMACS implementation challenge [32] in under 1,000 s. With proof logging enabled, we produced proof logs for 57 of the 59 instances, incurring a mean slowdown of 80.1; the final two instances were cancelled when their proof logs reached 1 TByte in size. We were then able to verify all 57 of these proofs, with verification being a mean of 10.1 times more expensive than writing the proofs. Note that the logging slowdown is to




```
pseudo-Boolean proof version 1.0
f 8 0
o xa xd                                                    ⇝ obj
p nonadja_e 2 * nonadja_f + nonadje_f + 3 d 2 *            ⇝ cc1
p nonadjb_d 5 *                                            ⇝ cc2
p nonadjc_d 2 *                                            ⇝ cc3
p obj cc1 + cc2 + cc3 +                                    ⇝ done
c done 0
```

Fig. 2. On the left, a weighted graph, with a clique of weight ten from vertices a and d highlighted. On the right, a proof that there is no heavier clique.

be expected [26]: the original solver is able to carry out a full recursive call and bounds calculation in under a microsecond. If each such call requires 1 KByte of logged information then this already exceeds the 100 MBytes per second write capabilities of a hard disk by an order of magnitude.

2.2 Weighted Clique Algorithms

In the maximum weight clique problem, vertices have weights, and we are now looking to find the clique with the largest sum of the weights of its vertices, rather than the most vertices. A simple bound for this problem is to produce a colouring, and then sum the maximum weight of each colour class. Consider the example in Fig. 2, and the three colour classes $\{a, e, f\}$, $\{b, d\}$ and $\{c\}$. By looking only at the largest weight in each colour class, we obtain a bound of $2+7+2=11$. This bound may be justified in a cutting planes proof by generating the at-most-one constraints for each colour class as previously, and then multiplying each colour class by its maximum weight before summing them. However, better bounds can be produced by allowing a vertex to appear in multiple colour classes, and by splitting its weight among these colour classes. If we allow vertex d to appear in the second colour class with weight 5 and in the third colour class with weight 2, then our bound is $2+5+2=9$. This technique originates with Babel [2], and is used in algorithms due to Tavares et al. [64,65], which are the current state of the art for many graph classes [46]. From a proof logging perspective, this splitting does not affect how we generate the bound, and so we may generate the proof shown on the right of Fig. 2.

Implementation. We implemented a simple certifying maximum weight clique algorithm using the Tavares et al. [64,65] bound in Python. With a timeout of 1,000 s, we were able to produce proof logs for 174 of the 289 benchmark instances from a recent collection [46]; all were verified successfully.

2.3 Maximal Clique Enumeration

Finally, in some applications we want to find every maximal clique (that is, one which cannot be made larger by adding vertices without removing vertices).

This problem also has a straightforward PB encoding: we express maximality by having a constraint for every vertex v saying that either x_v is selected, or at least one of its non-neighbours is.

The classic Bron-Kerbosch algorithm [6] uses a simple backtracking search, employing special data structures to minimise memory usage; it ensures maximality through a data structure called a *not*-set. We do not explain this data structure here, because it turns out to be equivalent in strength to unit propagation on the above PB model—indeed, to create a proof-logging Bron-Kerbosch algorithm, one needs only output a statement for every found solution, and a statement on every backtrack. More recent variations on this algorithm make use of different branching techniques [20,49,56,70] and supporting data structures [15,21,56], but although these new techniques can make a huge difference to theoretical worst-case guarantees and to empirical runtimes, they do not require any changes to how proof logging is performed.

We are interested in proof logging for this problem because there are discrepancies in tables of published results for some common benchmark instances—for example, does the "celegensneural" instance from the Newman dataset have 856 [21], 1,386 [20], or some other number of maximal cliques? We implemented proof logging for Tomita et al.'s variant of the algorithm [70], and were able to confirm that 1,386 is the correct answer. We were also able to confirm the published values of Eppstein et al. [20] for all of the BioGRID instances, the listed DIMACS instances, and for the Newman instances with no more than 10,000 vertices. We were unable to produce certified results for larger sparse graphs, because the OPB encoding size is linear in the number of *non*-edges in the inputs.

3 Maximum Common Induced Subgraph Algorithms

The maximum common induced subgraph problem can be defined in various equivalent ways, but the most useful is that we are given two graphs, and must find an injective partial mapping from the first graph to the second, where adjacent vertices are mapped to adjacent vertices and non-adjacent vertices are mapped to non-adjacent vertices, mapping as many vertices as possible. The problem arises in applications including in chemistry and biology [18,23,54]. However, in many cases, the common subgraph is required to be *connected*: that is, if we take any two assigned vertices from the first graph, then we must be able to find a path from one to the other without using unassigned vertices.

McCreesh et al. [41] compared two approaches to the problem, one based upon CP [50,73] and one based upon reduction to clique [3,17,34,54], and found that the best approach varied depending upon the kinds of graph being used. Since then, improvements have come from two different lines of research: one based upon weakening subgraph isomorphism algorithms [31], and one called McSplit which replaces general algorithms and data structures used in CP with much faster domain-specific ones [38,47]. We will discuss CP and McSplit, and then the clique reduction later, but first we must provide an appropriate PB encoding.

3.1 Pseudo-Boolean Encodings

To encode maximum common induced subgraph in PB form, we may adapt the subgraph isomorphism encoding of Gocht et al. [26]. For each vertex f in the first graph F, and for each vertex s in the second graph S, we have a variable $x_{f,s}$ which takes the value 1 if f is mapped to s; we also have a variable $x_{f,\perp}$ if f is unassigned. We then have exactly-one constraints over each set of $x_{f,-}$ variables, at-most-one constraints over each set of $x_{-,s}$ variables for injectivity, and induced adjacency constraints which are expressed using Gocht et al.'s second encoding,

$$
\begin{aligned}
x_{f,\perp} + \sum_{s \in \mathrm{V}(S)} x_{f,s} = 1 \qquad & f \in \mathrm{V}(F) \\
\sum_{f \in \mathrm{V}(F)} x_{f,s} \leq 1 \qquad & s \in \mathrm{V}(S) \\
\overline{x}_{f,s} + x_{g,\perp} + \sum_{t \in \mathrm{N}(s)} x_{q,t} \geq 1 \qquad & f \in \mathrm{V}(F),\ q \in \mathrm{N}(f),\ s \in \mathrm{V}(S) \\
\overline{x}_{f,s} + x_{g,\perp} + \sum_{t \in \overline{\mathrm{N}}(s)} x_{q,t} \geq 1 \qquad & f \in \mathrm{V}(F),\ q \in \overline{\mathrm{N}}(f),\ s \in \mathrm{V}(S)
\end{aligned}
$$

and the objective is to maximise the sum of the non-$\perp$ variables.

For the connected version of the problem, expressing connectedness as a constraint is a little trickier. Our encoding is informed by two simple observations: a subgraph with k vertices is connected if, for every pair of vertices in the subgraph, there is a walk of length no more than k between them, and secondly, for $k > 1$, there is a walk of length $2k$ between two vertices f and g if and only if there is some vertex h such that there are walks of length k between f and h and also between h and g.

Therefore, we first introduce auxiliary variables $x^1_{f,g}$ for every pair of vertices f and g in the first graph.[3] If f and g are non-adjacent, these variables are forced to false; otherwise we add constraints to specify that $x^1_{f,g}$ is true if and only if both $x_{f,\perp}$ and $x_{g,\perp}$ are false. In other words, $x^1_{f,g}$ is true precisely if f and g are adjacent and in the chosen subgraph. Writing $f \sim_F g$ and $f \not\sim_F g$ to mean vertices f and g are adjacent or not adjacent in the graph F respectively, this is:

$$
\begin{aligned}
\overline{x}^1_{f,g} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \not\sim_F g \\
\overline{x}^1_{f,g} + \overline{x}_{f,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g \\
\overline{x}^1_{f,g} + \overline{x}_{g,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g \\
x^1_{f,g} + x_{f,\perp} + x_{g,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g
\end{aligned}
$$

Next, we introduce auxiliary variables $x^2_{f,g}$, which will tell us if there is a walk of length 2 between vertices f and g. To do this, for each other vertex h,

[3] In all of what follows, these variables are equivalent under the exchange of f and g, and so we may halve the number of variables needed by exchanging f and g if $f > g$. We do this in practice, but omit this in the description for clarity.

we have a variable $x^2_{f,h,g}$ which we constrain to be true if and only if there is a walk of length 1 from f to h, and from h to g. Now, $x^2_{f,g}$ may be constrained to be true if and only if either there is a walk of length 1 between f and g, or at least one $x^2_{f,h,g}$ variable is true. We then repeat this process for walks of length 4, 8, and so on, until we reach a length k which equals or exceeded the number of vertices in the first graph. For $k \in \{2, 4, 8, \ldots, 2^{\lceil \log |\mathrm{V}(F)| \rceil}\}$:

$$
\begin{aligned}
x^{k/2}_{f,h} + \overline{x}^{k}_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^{k/2}_{h,g} + \overline{x}^{k}_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^{k}_{f,h,g} + \overline{x}^{k/2}_{f,h} + \overline{x}^{k/2}_{h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
\overline{x}^{k}_{f,g} + x^{k/2}_{f,g} + \sum_{h \in \mathrm{V}(F) \setminus \{f,g\}} x^{k}_{f,h,g} &\geq 1 && f, g \in \mathrm{V}(F) \\
x^{k}_{f,g} + \overline{x}^{k}_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^{k}_{f,g} + \overline{x}^{k/2}_{f,g} &\geq 1 && f, g \in \mathrm{V}(F)
\end{aligned}
$$

Finally, to enforce connectedness, for each pair of vertices f and g, we require that either $x_{f,\perp}$ or $x_{g,\perp}$ or $x^k_{f,g}$ is true.

$$
x_{f,\perp} + x_{g,\perp} + x^{k}_{f,g} \geq 1 \qquad f, g \in \mathrm{V}(F), k = 2^{\lceil \log |\mathrm{V}(F)| \rceil}
$$

An important property of this encoding is that all the auxiliary variables are *dependent*: that is, for every solution to the maximum common connected induced subgraph problem, there is exactly one feasible way of setting the auxiliary variables. In other words, the number of solutions to the PB encoding is exactly the same as the number of solutions to the real problem.

3.2 Proof Logging for Constraint Programming Algorithms

The McSplit algorithm [47] performs a CP-style backtracking search [50,73], looking to map as many vertices from the first graph as possible to distinct vertices in the second graph. We will therefore continue to use RUP to generate proofs. For adjacency and non-adjacency constraints, McSplit's reasoning is equivalent to unit propagation on our PB constraints, and so no help is needed. For the bound, McSplit performs "all different except $\perp$" propagation, but with the number of occurrences of $\perp$ constrained to beat the best solution found so far. Due to the special structure of the domains during search, it is able to do this in linear time, without needing the usual matching and components algorithm [52]. However, when it fails, it produces a sequence of Hall sets, and so we may reuse the justification technique described by Elffers et al. [19] with only a simple modification to cope with the objective function.

For the connected variant, McSplit uses a restricted branching scheme [47,73] rather than a conventional propagator: once at least one vertex is assigned a

non-null value, it may only branch on vertices adjacent to a vertex already assigned a non-null value. If no such vertices exist, it backtracks. Interestingly, this requires no explicit support in proof logging: by carefully stepping through the auxiliary variables in the PB encoding, level by level, it can be seen that RUP will propagate all remaining variables to false when in this situation.

Therefore, implementing proof logging in McSplit requires four kinds of statement to be logged. Firstly, any new incumbent must be noted, as in the previous section. Secondly, all backtracks must be logged using a RUP rule. Thirdly, whenever the bound function detects that the current state may be pruned, we must derive a new constraint justifying this. And fourthly, it is extremely helpful to delete intermediate constraints using "level" statements. Again, this proof logging is completely agnostic to changes to the branching heuristic [38].

We implemented this proof logging inside the original McSplit implementation, and tested it for both the connected and non-connected variants of the problem on a commonly used set of benchmark instances [10,63]. We successfully verified McSplit's solutions to all 16,300 instances of no more than 25 vertices. Proof logging introduced a mean slowdown of 67.0 and 298.9 for non-connected and connected respectively, whilst verification was a further 13.4 and 21.6 times slower; again, writing to hard disk was by far the biggest bottleneck, as McSplit can make over five million recursive calls per second.

3.3 Maximum Common (Connected) Subgraph via Clique

An alternative approach to the maximum common subgraph problem is via a reduction to the maximum clique problem [3,34,54]. This reduction resembles the microstructure encoding of the CP representation, and is the best known approach on labelled graphs; we refer to McCreesh et al. [41] for a detailed explanation. From a proof logging perspective, one might expect that this encoding would require a whole new PB representation, or perhaps a large change to how proof logging is performed by a maximum clique algorithm. However, this is not the case: given the PB model for a maximum common subgraph problem from earlier in this section, we can derive the non-adjacency constraints needed for the clique model described in the previous section using only RUP, whilst the objective function needs no rewriting at all. Therefore, the only changes needed to a proof-logging clique algorithm is in the lookup of constraint identifiers.

McCreesh et al. [41] also show how a maximum clique algorithm can be adapted to deal with the connected variant of the problem, by embedding a propagator inside the clique algorithm. From a proofs perspective, we can work with the PB model and the clique reformulation, similar to channelling [8]—and since connectedness propagation requires no explicit proof logging with the original PB representation, it also requires no proof logging when performed inside a clique algorithm.

We therefore reimplemented McCreesh et al.'s clique common (connected) subgraph algorithm [41], and added proof logging support. Proof logging immediately caught a bug in our reimplementation that testing had failed to identify: we were only updating the incumbent when a maximal clique was found, which

is correct in conventional clique algorithms but not for the connected variant, but this very rarely caused incorrect results. Once corrected, for both variants of the problem, we were able to verify all 11,400 instances of no more than 20 vertices from the same set of instances [10,63]. Proof logging introduced an average slowdown of 28.6 and 39.7 for non-connected and connected respectively, and verification was on average a further 11.3 and 73.1 times slower.

4 Conclusion

We have shown that pseudo-Boolean proof logging is sufficiently powerful and flexible to make certification possible for a wide range of graph solvers. Particularly of note is how proof logging is largely agnostic towards most changes to details in algorithm behaviour (such as search order, methods for calculating bounds, and underlying algorithms and data structures), and how it is able to deal with reformulation or changes of representation. This suggests that requiring certification should not be an undue burden on solver authors going forward. We also stress the simplicity of implementation: for every algorithm we considered, proof logging only required access to information that was already easily available inside the existing solvers. In particular, we do not need to implement any form of pseudo-Boolean constraint processing in order to generate these proofs, nor does a solver have to in any way understand or otherwise reason about the proofs it is producing. Furthermore, in each case, adding in support for proof logging was considerably easier than implementing the algorithm itself.

It is important to remember that proof logging does not prove that any algorithm or solver is correct. Instead, it provides a proof that a *claimed solution* is correct—and if a solution was produced using unsound reasoning, this will be caught, even if the solution is correct, or if it was produced by a correct algorithm being run on faulty hardware or with a buggy compiler. Additionally, this process does not verify that the encoding from a high level model to the PB representation is correct. To offset this, the encodings we use are deliberately simple, and when a more complex internal representation is used (such as in the clique model for maximum common subgraph), we can log the reformulation and verify the log in terms of the simpler model. This reformulation also suggests that for competitions, providing a standard encoding would not be a problem.

Although proof logging introduces considerable overheads (particularly when compared to the techniques used in the SAT community, which do not need to deal with powerful but highly efficient propagators), it can still be used to verify medium-sized instances involving tens of millions of inference steps. Given the abundance of buggy solver implementations that *usually* produce correct answers, we suggest that all authors of dedicated graph solvers should adopt proof logging from now on, and that competition organisers should strongly consider requiring proof logging support from entrants. For larger and harder instances, proof logging can be disabled, but because proof logging does not require intrusive changes to solver internals, this would still give us a large increase in confidence in the correctness of results compared to conventional testing.

References

1. Atserias, A., Bonacina, I., de Rezende, S.F., Lauria, M., Nordström, J., Razborov, A.A.: Clique is hard on average for regular resolution. In: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, 25–29 June 2018, pp. 866–877 (2018)
2. Babel, L.: A fast algorithm for the maximum weight clique problem. Computing **52**(1), 31–38 (1994)
3. Balas, E., Yu, C.S.: Finding a maximum clique in an arbitrary graph. SIAM J. Comput. **15**(4), 1054–1068 (1986)
4. Batsyn, M., Goldengorin, B., Maslov, E., Pardalos, P.M.: Improvements to MCS algorithm for the maximum clique problem. J. Comb. Optim. **27**(2), 397–416 (2014)
5. Biere, A.: Tracecheck (2006). http://fmv.jku.at/tracecheck/
6. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
7. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_6
8. Cheng, B.M.W., Lee, J.H.M., Wu, J.C.K.: Speeding up constraint propagation by redundant modeling. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 91–103. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61551-2_68
9. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Sattar, A., Kang, B. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 49–58. Springer, Heidelberg (2006). https://doi.org/10.1007/11941439_9
10. Conte, D., Foggia, P., Vento, M.: Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. **11**(1), 99–143 (2007)
11. Cook, W.J., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discret. Appl. Math. **18**(1), 25–38 (1987)
12. Cook, W.J., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Math. Program. Comput. **5**(3), 305–344 (2013)
13. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
14. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_7
15. Dasari, N.S., Ranjan, D., Zubair, M.: pbitMCE: a bit-based approach for maximal clique enumeration on multicore processors. In: 20th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2014, Hsinchu, Taiwan, 16–19 December 2014, pp. 478–485 (2014)
16. Depolli, M., Konc, J., Rozman, K., Trobec, R., Janezic, D.: Exact parallel maximum clique algorithm for general and protein graphs. J. Chem. Inf. Model. **53**(9), 2217–2228 (2013)
17. Durand, P.J., Pasari, R., Baker, J.W., Tsai, C.C.: An efficient algorithm for similarity analysis of molecules. Internet J. Chem. **2**(17), 1–16 (1999)

18. Ehrlich, H.C., Rarey, M.: Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. Wiley Interdiscip. Rev.: Comput. Mol. Sci. **1**(1), 68–79 (2011)
19. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using pseudo-boolean reasoning. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, 7–12 February 2020, pp. 1486–1494 (2020)
20. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. ACM J. Exp. Algorithmics **18** (2013)
21. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 364–375. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20662-7_31
22. Gange, G., Stuckey, P.J.: Certifying optimality in constraint programming, February 2019. Presentation at KTH Royal Institute of Technology. Slides https://www.kth.se/polopoly_fs/1.879851.1550484700!/CertifiedCP.pdf
23. Gay, S., Fages, F., Martinez, T., Soliman, S., Solnon, C.: On the subgraph epimorphism problem. Discret. Appl. Math. **162**, 214–228 (2014)
24. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, 2–4 January 2008 (2008)
25. Gillard, X., Schaus, P., Deville, Y.: SolverCheck: declarative testing of constraints. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 565–582. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_33
26. Gocht, S., McCreesh, C., Nordström, J.: Subgraph isomorphism meets cutting planes: solving with certified solutions. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020 [scheduled for July 2020, Yokohama, Japan, postponed due to the Corona pandemic], pp. 1134–1140 (2020)
27. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), Munich, Germany, 3–7 March 2003, pp. 10886–10891. IEEE Computer Society (2003)
28. Hespe, D., Lamm, S., Schulz, C., Strash, D.: WeGotYouCovered: the winning solver from the PACE 2019 implementation challenge, vertex cover track. CoRR abs/1908.06795 (2019)
29. Heule, M., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 181–188 (2013)
30. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 345–359. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_24
31. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, California, USA, 4–9 February 2017, pp. 3907–3914 (2017)

32. Johnson, D.S., Trick, M.A.: Introduction to the second DIMACS challenge: cliques, coloring, and satisfiability. In: Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, 11–13 October 1993, pp. 1–10 (1993)
33. Konc, J., Janežič, D.: An improved branch and bound algorithm for the maximum clique problem. MATCH Commun. Math. Comput. Chem. **58**(3), 569–590 (2007)
34. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. CALCOLO **9**(4), 341–352 (1973). https://doi.org/10.1007/BF02575586
35. Li, C., Jiang, H., Manyà, F.: On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. Comput. Oper. Res. **84**, 1–15 (2017)
36. Li, C.-M., Jiang, H., Xu, R.-C.: Incremental MaxSAT reasoning to reduce branches in a branch-and-bound algorithm for MaxClique. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 268–274. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19084-6_26
37. Li, C.M., Quan, Z.: An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010 (2010)
38. Liu, Y., Li, C., Jiang, H., He, K.: A learning based branch and bound for maximum common subgraph related problems. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, 7–12 February 2020, pp. 2392–2399 (2020)
39. Maslov, E., Batsyn, M., Pardalos, P.M.: Speeding up branch and bound algorithms for solving the maximum clique problem. J. Glob. Optim. **59**(1), 1–21 (2014)
40. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
41. McCreesh, C., Ndiaye, S.N., Prosser, P., Solnon, C.: Clique and constraint models for maximum common (connected) subgraph problems. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 350–368. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_23
42. McCreesh, C., Pettersson, W., Prosser, P.: Understanding the empirical hardness of random optimisation problems. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 333–349. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_20
43. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms **6**(4), 618–635 (2013)
44. McCreesh, C., Prosser, P.: Reducing the branching in a branch and bound algorithm for the maximum clique problem. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 549–563. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_40
45. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. ACM Trans. Parallel Comput. **2**(1), 8:1–8:27 (2015)
46. McCreesh, C., Prosser, P., Simpson, K., Trimble, J.: On maximum weight clique algorithms, and how they are evaluated. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 206–225. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_14

47. McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 712–719 (2017)
48. McCreesh, C., Prosser, P., Trimble, J.: The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In: Gadducci, F., Kehrer, T. (eds.) ICGT 2020. LNCS, vol. 12150, pp. 316–324. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51372-6_19
49. Naudé, K.A.: Refined pivot selection for maximal clique enumeration in graphs. Theor. Comput. Sci. **613**, 28–37 (2016)
50. Ndiaye, S.N., Solnon, C.: CP models for maximum common subgraph problems. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 637–644. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_48
51. Nikolaev, A., Batsyn, M., Segundo, P.S.: Reusing the same coloring in the child nodes of the search tree for the maximum clique problem. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 275–280. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19084-6_27
52. Petit, T., Régin, J.-C., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 451–463. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_31
53. Prosser, P.: Exact algorithms for maximum clique: a computational study. Algorithms **5**(4), 545–587 (2012)
54. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. **16**(7), 521–533 (2002)
55. Roussel, O., Manquinho, V.M.: Input/output format and solver requirements for the competitions of pseudo-Boolean solvers, January 2016. Revision 2324. http://www.cril.univ-artois.fr/PB16/format.pdf
56. San Segundo, P., Furini, F., Artieda, J.: A new branch-and-bound algorithm for the maximum weighted clique problem. Comput. Oper. Res. **110**, 18–33 (2019)
57. San Segundo, P., Lopez, A., Batsyn, M.: Initial sorting of vertices in the maximum clique problem reviewed. In: Pardalos, P.M., Resende, M.G.C., Vogiatzis, C., Walteros, J.L. (eds.) LION 2014. LNCS, vol. 8426, pp. 111–120. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09584-4_12
58. San Segundo, P., Lopez, A., Batsyn, M., Nikolaev, A., Pardalos, P.M.: Improved initial vertex ordering for exact maximum clique search. Appl. Intell. **45**(3), 868–880 (2016)
59. San Segundo, P., Matía, F., Rodríguez-Losada, D., Hernando, M.: An improved bit parallel exact maximum clique algorithm. Optim. Lett. **7**(3), 467–479 (2013)
60. San Segundo, P., Nikolaev, A., Batsyn, M., Pardalos, P.M.: Improved infrachromatic bound for exact maximum clique search. Informatica Lith. Acad. Sci. **27**(2), 463–487 (2016)
61. San Segundo, P., Rodríguez-Losada, D., Jiménez, A.: An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Res. **38**(2), 571–581 (2011)
62. San Segundo, P., Tapia, C.: Relaxed approximate coloring in exact maximum clique search. Comput. Oper. Res. **44**, 185–192 (2014)
63. Santo, M.D., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recogn. Lett. **24**(8), 1067–1079 (2003)
64. Tavares, A.W.: Algoritmos exatos para problema da clique maxima ponderada. Ph.D. thesis, Universidade federal do Ceará (2016)

65. Tavares, W.A., Neto, M.B.C., Rodrigues, C.D., Michelon, P.: Um algoritmo de branch and bound para o problema da clique máxima ponderada. In: Proceedings of XLVII SBPO, vol. 1 (2015)
66. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J. Glob. Optim. **37**(1), 95–111 (2007)
67. Tomita, E., Seki, T.: An efficient branch-and-bound algorithm for finding a maximum clique. In: Calude, C.S., Dinneen, M.J., Vajnovszki, V. (eds.) DMTCS 2003. LNCS, vol. 2731, pp. 278–289. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45066-1_22
68. Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., Wakatsuki, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique. In: Rahman, M.S., Fujita, S. (eds.) WALCOM 2010. LNCS, vol. 5942, pp. 191–203. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11440-3_18
69. Tomita, E., Sutani, Y., Higashi, T., Wakatsuki, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. IEICE Trans. Inf. Syst. **96-D**(6), 1286–1298 (2013)
70. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. **363**(1), 28–42 (2006)
71. Tomita, E., Yoshida, K., Hatta, T., Nagao, A., Ito, H., Wakatsuki, M.: A much faster branch-and-bound algorithm for finding a maximum clique. In: Zhu, D., Bereg, S. (eds.) FAW 2016. LNCS, vol. 9711, pp. 215–226. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39817-4_21
72. Veksler, M., Strichman, O.: A proof-producing CSP solver. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010 (2010)
73. Vismara, P., Valery, B.: Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In: Le Thi, H.A., Bouvry, P., Pham Dinh, T. (eds.) MCO 2008. CCIS, vol. 14, pp. 358–368. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87477-5_39
74. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31