VeriPB: The Easy Way to Make Your Combinatorial Search Algorithm Trustworthy

Stephan Gocht[1,2], Ciaran McCreesh[3], and Jakob Nordström[2,1]

[1] Lund University, Lund, Sweden
[2] University of Copenhagen, Copenhagen, Denmark
[3] University of Glasgow, Glasgow, Scotland
stephan.gocht@cs.lth.se, ciaran.mccreesh@glasgow.ac.uk, jn@di.ku.dk

Abstract. We give an overview of the VeriPB tool, which can be used to make certifying solvers for combinatorial optimisation problems.

The VeriPB tool[4] provides an easy way of making combinatorial search algorithms *certifying*. Certifying solvers, which output a proof of correctness alongside a claimed solution, are now standard in the Boolean satisfiability (SAT) community, but have not seen much uptake elsewhere. The key problem with the SAT approach to proof logging is that their chosen proof formats, CNF and DRAT [13], make it extremely impractical to describe the more sophisticated reasoning used in constraint programming propagators [4] and in other combinatorial search algorithms [6, 7]. In contrast, the psuedo-Boolean models and cutting planes proof system used by VeriPB makes it easy to justify combinatorial arguments. This tool demonstration gives a solver author's perspective of using VeriPB to make an algorithm implementation trustworthy.

When a solver claims than an instance of an NP-complete problem is unsatisfiable, it is hard to be confident that the solver is correct. Similarly, for optimisation problems, it is hard to be sure that a claimed solution is indeed optimal, and for enumeration problems, it is hard to be sure that no solutions have been missed. The VeriPB tool handles all of these scenarios through *proof logging*, which is a particular form of *certifying* [9]. The key steps involved are:

- The problem is expressed as a pseudo-Boolean (PB) problem instance, encoded in the standard OPB file format [10]. This can either be done by the solver, or in the case of solver competitions, can be provided by the competition organiser. A PB instance is simply an integer linear program where all the variables have domain $\{0, 1\}$; conveniently, CNF clauses can be represented directly as PB constraints.
- As it executes, the solver outputs a machine-readable proof log which records the steps it took to reach its solution. This proof log consists of *reverse unit propagation* (RUP) steps [4, 5] which record every time the solver backtracks, together with additional manual constraint derivations using *cutting planes* proof system rules [2] to justify any complex constraint propagation

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

or bounds; these manual derivations are used to ensure that all inferences performed by the solver can be reflected inside the verifier just through using unit propagation. For satisfiable, enumeration, and optimisation instances, this log also records solutions or new incumbents as they are found.
– A proof verifier such as VeriPB takes the OPB file and the proof log, and checks whether the proof is valid. A proof verifier is a very simple piece of software when compared to a typical solver, and so it is much easier to trust. In particular, the proof verifier can only carry out very simple inference steps and only as directed, and does not perform any search.

From the point of view of end users and solver authors, this approach has a number of advantages compared to full formal verification [3], and compared to earlier proof logging approaches [11–13], which together mean we now believe it is practical to pursue proof logging as the new "socially acceptable standard" for solver implementers.

Firstly, proof logs can be stored, and audited at a later date. It is even reasonably simple to implement an independent verifier, because the verifier does not need to understand constraint propagation or bounds. For example, because all-different reasoning can be justified compactly using cutting planes steps, the verifier can verify Hall set reasoning [4], without knowing what a Hall set is or how all-different propagation works. Similarly, the wide variety of bound and inference functions used in modern subgraph-finding algorithms [6, 7] can all be verified without the verifier having any knowledge of graphs.

Secondly, this form of proof logging is simple to implement inside solvers. For all of the examples we have tried so far [4, 6, 7], it has taken *much* less time (between a factor of two and a factor of many hundreds) to implement proof logging in an existing solver than it took to implement the solver itself. This remained true even for non-experts. The key to this is RUP [4]: for search, the solver needs only output the trail every time it backtracks. Meanwhile, most constraint propagation steps do not require explicit proof derivations: RUP ensures that any inference steps which follow by integer bounds consistency [1] from the provided PB model (which for clausal constraints, is the same as unit propagation in SAT, but in general is more powerful) are handled implicitly. For more complex propagators or bounds functions, their behaviour must be justified, but this can be done without needing to consider the trail; furthermore, it is often easy to reuse justification templates between different algorithms and solvers.

Thirdly, the process is at least reasonably efficient. A particular goal has been to ensure that proof logs are, in some sense, not longer than the amount of work carried out by a solver. The cutting planes proof system appears to be particularly suitable here, being able to express a wide range of combinatorial arguments [4, 6, 7]. This would not be the case if we used, for example, the resolution proof system, which would require an exponential blowup for justifying Hall set reasoning [8, 11].

Fourthly, the process certifies *solutions*, rather than proving a solver correct. This is a mixed blessing. It does not guarantee that a solver will never produce an incorrect answer, but it does ensure that if an incorrect answer is ever produced

(or if a correct answer is produced using unsound reasoning), then it can be detected. This holds even if the error was due to a compiler bug or a hardware fault, or due to the solver relying upon an algorithm whose purported proof of correctness turned out to be spurious.

And finally, we have found that proof logging can help with solver development, catching bugs early during the implementation process that conventional testing had missed. VeriPB includes a number of features to help with this, such as the ability to trace proof logs as they are executed, the ability to assert that derived constraints imply expected consequences, and the ability to instruct the verifier to accept certain facts on faith, to enable incremental development.

References

1. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Proceedings. pp. 49–58 (2006)
2. Cook, W.J., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discrete Applied Mathematics **18**(1), 25–38 (1987)
3. Dubois, C.: Formally verified constraints solvers: a guided tour (2020), invited talk at the 13th Conference on Intelligent Computer Mathematics (CICM)
4. 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, New York, NY, USA. pp. 1486–1494 (2020)
5. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008 (2008)
6. Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., Trimble, J.: Certifying solvers for clique and maximum common (connected) subgraph problems. In: Proceedings of the Twenty-Sixth International Conference on Principles and Practice of Constraint Programming, CP 2020 (2020)
7. 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. pp. 1134–1140 (2020)
8. Haken, A.: The intractability of resolution. Theor. Comput. Sci. **39**, 297–308 (1985)
9. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
10. Roussel, O., Anquinho, V.M.: Input/output format and solver requirements for the competitions of pseudo-Boolean solvers (2012), http://www.cril.univ-artois.fr/PB12/format.pdf
11. Stuckey, P.J.: Certifying optimality in constraint programming (February 2019), talk at KTH Royal Institute of Technology
12. 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, July 11-15, 2010 (2010)
13. Wetzler, N., Heule, M., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Vienna, Austria, July 14-17, 2014. Proceedings. pp. 422–429 (2014)