Solving very hard problems: Cube-and-Conquer, a hybrid SAT solving method

Marijn J.H. Heule[*]
The University of Texas at Austin

Oliver Kullmann
Swansea University, UK

Victor W. Marek
University of Kentucky

Abstract

A recent success of SAT solving has been the solution of the boolean Pythagorean Triples problem [Heule *et al.*, 2016], delivering the largest proof yet, of 200 terabytes in size. We present this and the underlying paradigm Cube-and-Conquer, a powerful general method to solve big SAT problems, based on integrating the "old" and "new" methods of SAT solving.

1 Introduction

We consider the problem of solving very hard concrete problem instances of a decision problem in NP, like searching for a specific configuration or for a specific combinatorial design. An important aspect here is that we want to solve both YES- and NO-instances, that is, we not only want to find solutions, but we also want to be able to determine that no solution exists. The tools at our disposal are

(I) special purpose solvers, typically some form of informed backtracking;

(II) Constraint Satisfaction solvers [Rossi *et al.*, 2006], based on propagation of inferences by constraints;

(III) SAT solvers [Biere *et al.*, 2009], for inputs in CNF, using less local, but more global forms of inference.

One would assume that the above order correlates to the amount of specific knowledge we have for the problem: if there is a lot of intricate knowledge, then (I) should be best, if there are many different constraint types, with good propagation properties, then (II) should be best, and for the rest (III) should be best. Surprisingly, SAT solving is getting so strong that indeed (III) seems today the best solution in most cases.

An example where only a solution by (I) is known is the determination that there is no projective plane of order 10 (a certain combinatorial design), based on special (relatively complicated) algebraic structure, and pruning the search space by symmetries and other tricks [Lam, 1991]. Their solution is similar to the computer proof of the Four Colour theorem [Wilson, 2013], namely being based on a man-made case-distinction, where the computer "ticks off possibilities", but the computations are much more involved. For modern standards their search space is not extremely big, so the special computations should be rather easy to replicate — but one really wants to solve just the *basic problem formulation*. To the best of our knowledge the effort has not been replicated, and there is definitely no formal proof.

As general purpose approaches we have (II) and (III). On small-scale problems, (II) appears more successful, but the methods seem not to scale well. On the other hand, (III) seems to scale surprisingly well. An early success for a very hard problem was [Kouril and Paul, 2008], and two recent successes are [Konev and Lisitsa, 2015; Codish *et al.*, 2016], while we will talk here about the story behind our own [Heule *et al.*, 2016], solving the long outstanding "Pythagorean Triples Problem". We explain this problem below in Section 2, while for the moment it suffices to say that it consists of solving a series of satisfiable problems, which are relatively easy, and to decide and *prove* unsatisfiability of one very hard problem. Different from the above mentioned usage of special purpose methods (I), for these SAT problems there is no mathematical structure known, which would allow to determine a feasible search space in advance, but all what one can say (currently) about these search spaces is that their sizes are far beyond the number of particles in the universe — and thus the *magic* of SAT solving is needed, based on the *SAT revolution* by "modern SAT solvers".

1.1 Attacking very hard SAT problems

Now how does SAT-solving work? We concentrate in this article on complete methods, while for incomplete methods see the overview [Kautz *et al.*, 2009]. The "old method" is called "look-ahead", and improves backtracking by estimating as good as possible what would happen in the different search directions, and chooses one which seems best; see Subsection 3.1 for more information. The "new method" is called "CDCL" ("conflict-driven clause-learning"), and just tries to find a satisfying assignment using only very basic inferences, just enough so that when the search fails (a "conflict" is found), the negation of the "main" assignments made (note that this is a disjunction of literals, i.e., a clause of a CNF), when "learned", prunes the search space; see Subsection 3.2 for more. For the "SAT revolution" the CDCL solvers are responsible, very often being able to solve very big problems. So one would guess, that CDCL would also be the best

[*]Supported by the National Science Foundation under grant CCF-1526760 and by AFRL Award FA8750-15-2-0096.

method for the very hard problems considered in this article. However it seems that CDCL solvers need a "weak spot", that they are very good in finding short proofs, but if none exists, then they are performing badly. This appears actually to be rather natural, since their design excludes, for performance reasons, any kind of global "overview". Fortunately, this is precisely the strength of look-ahead solvers.

This motivates our method "Cube-and-Conquer" (C&C), where both methods, "old and new", work together to solve SAT-problems: first the problem is split in an intelligent way via look-ahead (its speciality) into many subproblems (the "cubes"), and then these subproblems are "conquered" by CDCL-solvers (which now works well on these chopped problems); see Section 4.

1.2 Proofs are needed!

So we have now strong tools to solve very hard SAT problems. What about correctness of these results, which can be affected by hardware and software errors? According to the general nature of problems in NP, in case of a YES-answer we typically have short certificates which are easy to check: in the case of SAT these are typically the satisfying assignments found (or reconstructed). But for the NO-answers there seem to be no short certificates (if they would exist in general, then we had NP=coNP). And so extracting proofs from solver runs and checking them independently is a highly relevant question, and the construction of the 200TB proof is a major achievement of [Heule *et al.*, 2016], and will be discussed in Section 5. This extraction of proofs of unsatisfiability, which then can be checked by possibly certified checkers on possibly certified hardware, is superior to aim at certified solvers, since due to the enormous complexity of SAT-solving tasks, solvers are far more complicated than checkers, and progress here is also far more rapidly (and needed — although checking a 200TB proof is also a nontrivial task). Furthermore the extracted proofs are interesting objects of study.

2 Mathematical applications

Problems coming from Ramsey theory [Graham *et al.*, 1990] yield very good benchmarks for SAT solving. A basic question studied in this theory is as follows: Consider a set X and some notion of structure between the elements. Originally there are lots of these structures in X — now can we partition X into two parts such that all of these structures are destroyed? Sure there are cases where this is the case, but Ramsey theory is interested especially in resilience there, forms of structures where for big enough X there will always remain at least one such structure untouched. If we translate "structure" as some special subset of X, then we obtain a hypergraph with vertex set X, and destroying all structures means to 2-colour the hypergraph, i.e., assigning one of two col-ORs to every vertex such that no hyperedge is monochromatic (every hyperedge has both col-ORs in it). And the interesting case is when X indeed is *not* 2-colourable. Ramsey theory in general considers also infinite settings, but in many cases w.l.o.g. one can restrict attention to finite X, using that in many relevant logics inconsistencies have a finite character. So then it becomes interesting to ask for the size of the smallest X exhibiting non-2-colourability, which is then called some form of "Ramsey number".

The relevant type of Ramsey problem for the application of this paper is the "Schur problem", based on "Schur triples", where $X = \{1, \ldots, n\}$ for a natural number n, and the hyperedges are the subsets $\{a, b, c\} \subseteq X$ with $a + b = c$. This problem was introduced in [Schur, 1917], which showed the *existence* of all "Schur numbers", considering now $m \geq 2$ colours: for n large enough we obtain a non-m-colourable problem. We obtain a much more challenging problem when restricting the triples to *Pythagorean triples* $a^2 + b^2 = c^2$. This problem was posed by Ronald Graham in the 1980s, offering in the Erdős-style \$100 for its solution. It was even unknown whether for n large enough we would obtain a non-2-colourable problem at all [Croot and Lev, 2006, Subsection 3.9]. This was solved [Heule *et al.*, 2016], showing that indeed such n exists. Namely for $n = 7824$ the problem is still 2-colourable, while for $n = 7825$ it is non-2-colourable. The proof for the first statement is easy to show — just a partitioning of $\{1, \ldots, 7824\}$ into two parts such that for all Pythagorean triples $a^2 + b^2 = c^2$ for natural numbers $1 \leq a, b, c \leq 7824$ every triple hits both parts. But the proof of the non-2-colourability is naturally much more involved, and the already highly compressed extracted proof had the size of 200TB, the "largest proof ever" [Lamb, 2016]. Indeed, as already mentioned, the extraction of such proofs is a very interesting question in itself (considered in Section 5) — unsatisfiability often needs strict verification.

3 Basic outline of SAT solving

We now give a short high-level overview on the two main methods ("old and new") for SAT solving, to gain a basic understanding, so that the potential for collaboration between the two methods can be better understood.

3.1 Look-ahead

The look-ahead method is a natural further development of the backtracking method; see [Heule and van Maaren, 2009; Kullmann, 2009] for a fuller account. Backtracking here means that for the propositional formula F to be solved a variable v is picked according to the branching-heuristics, and the two cases, setting v to `false` and `true`, are considered recursively. So we obtain a binary (splitting or branching) tree, with the instantiations of F at its nodes. If all leaves are found unsatisfiable, then the original F is unsatisfiable, otherwise F is satisfiable. A very basic backtracking SAT solver is the DLL-solver [Davis *et al.*, 1962]. The DLL-algorithm uses *unit-clause propagation* (UCP) at each node to find the most basic inferences for SAT solving. This uses the fact that the input of (standard) SAT solving is actually a conjunctive normal form (CNF), a conjunction of disjunction of literals, where literals are variables or their negations. If now a unit-clause, a clause of length one, exists, then the single literal in it has to be set to `true`, and so one variable can be assigned. Possibly this assignment creates further unit-clauses, and the whole process of making all such assignments is called UCP.

Now the basic idea of look-ahead is that for the heuristics to choose variable v, it actually should look at the results

of UCP after setting v to `false` resp. `true`, to see what *really* happens. Naturally, if one of these two UCPs leads to a contradiction, then the other branch is forced, and we found actually a logically valid inference; the underlying general reduction is called *failed-literal elimination*. If we now add the basic idea for the heuristics choosing v, to minimise and balance the splitting tree, assuming *unsatisfiability* as the worst-case, then we have the rough outline of the look-ahead method. Due to the relatively heavy processing load at each node ("in-processing"), look-ahead are not able to solve very large formulas (it might take days just to reach a leaf *once*).

Look-ahead SAT-solving in its simplest form as a (non-deterministic) proof system is nearly exactly *tree-resolution*: this proof system will be explained in Section 5, but for the moment all what matters is that the proofs are trees, i.e., there is no overlap between different branches, which indeed is the main advantage *and* the main weakness of look-ahead solving — the branches of the computations don't interact. We note here that this means that look-ahead solver can run for a long time, since nothing accumulates (from different branches).

3.2 Conflict-driven clause-learning

The CDCL-method has many subtle points, and for an overview see [Marques-Silva *et al.*, 2009]. However, from an abstract point of view it is easy to describe, and that suffices for our purposes (since we want just to see how look-ahead and CDCL complement each other). We start with the backtracking tree, but now without look-ahead, just going down the tree, until a contradiction was found (otherwise we are done). Then the assignments leading to this conflict are analysed, and "learned", i.e., adding the disjunction of the negated assignments (a clause) to the clause-base. In the ordinary situation, the solver then backtracks until the point where the additional clause prunes the search space, but if current progress is deemed too slow, then a restart is performed, throwing away the backtracking tree altogether (but keeping the learned clauses). It is important to perform as little work as possible to reach a conflict, since very large problems have to be handled. So very efficient UCP is crucial, not looking at the whole formula, but only working towards the conflicts. Furthermore no "overview heuristics" are used (as for look-ahead), but a dynamic heuristics based on usage of variables in recently learned clauses. Since learned clauses are added to the clause-base, and can be re-used, the underlying proof system is now *dag-resolution* (see Section 5); the details are subtle here, but see [Pipatsrisawat and Darwiche, 2011] for a basic paper here. So in principle a more powerful proof system is utilised here but this comes at a price: the efficiency of CDCL solvers depends on *removing* most learned clauses during their run (by some heuristics). So their dag (directed acyclic graph) structure, the reuse of learned clauses, is local, happens only in a relatively short time frame. It also doesn't make much sense to run a modern CDCL solver (which very aggressively deletes clauses) for a long time — it won't finish.

4 A hybrid method: C&C

Now we can understand the basic of C&C easily: Look-ahead solvers are very good at splitting problems, while CDCL-solvers are very good at solving problems of with local structure. So use a look-ahead solver initially (the cubing phase), but cut off the branching tree appropriately, and let then the CDCL-solver finish the tasks. The following points are important for the cut-off line:

- The cut-off point along a branch must utilise the look-ahead solver's "understanding" of achievement, not just some simple measure like tree-depth — this makes an essential difference to typical parallelisation schemes.
- It is important that the number of leaves is large, for hard problems in the millions. So the subproblems are not created dynamically, but statically (in advance).
- The CDCL-solver should take for each problem at most, say, a minute (they can do amazing things in this time).

The idea appeared first in [Ahmed *et al.*, 2014], in the context of solving problems from Ramsey theory. This yields an evidentiary fact that problems from Ramsey theory are indeed good benchmarks for SAT solvers, leading to the discovery of new ideas. The framework was then developed systematically in [Heule *et al.*, 2012].

4.1 Perfect parallelisation

Via C&C very good parallelisation is possible with little overhead and possibly millions of processors (for very hard problems). For industrial problems, [Heule *et al.*, 2012] outlines automatic configuration of the splitting process, while for truly hard problems it is natural (and important) to fine-tune the process. For the conquer-phase we already have prepared many processes (much more than processors), so here there is no need for load-levelling or other complications. Experience shows that if the splitting achieves to bring down the runtime to a minute or less, then CDCL-solving times are stable. For extremely hard problems, like our boolean Pythagorean Triples problem, indeed a two-level splitting was needed:

- In this case first on a single processor a splitting (cubing) into 10^6 subproblems was done; since such very hard instances are relatively small (if they are feasible), this didn't take much time, in this case about 15 minutes.
- Each of these 10^6 subproblems was then solved by C&C.
- This can be understood as a second-splitting splitting in parallel (using 800 processors in this case), to split into altogether $40 \cdot 10^9$ subproblems.
- So here one might say that actually much shorter CDCL-runtimes were used. In reality for each of the 10^6 subproblems a refined C&C version was used, using an incremental CDCL-solver, so that the second-level splitting was used by the CDCL-solver only for guidance; but this was not absolutely essential.

Altogether 21,900 hours for cubing and 13,200 hours for conquering were needed, altogether roughly 2 days with 800 cores. This example also shows that parallelisation of the cube-phase can be done easily. Again there is no real need for load-levelling — if there would be much variation when splitting, then this would mean that the look-ahead solver is already doing at least some solving, not just splitting, which can be observed and avoided by cutting off earlier.

4.2 An astounding observation

This looks alright then: we gave partitioning of the whole problem to a specialist (the cube-solver), and then achieved good parallelisation. But actually this is not the end of the story, but a surprise is waiting. For a typical problem, let N be the number of subproblems, the number of leaves of the cubing tree. If $N = 1$, then we have pure CDCL, while if N is large enough, then we have pure look-ahead (no subproblems left to be solved). The typical case is that the optimal N is somewhere in between (so that the subproblems become relatively easy for CDCL). However, the observed optimum in many cases is one order of magnitude (or more) faster than what the best single method could achieve. So this method is not just some parallelisation, but the combination of "old and new" in a sense has a better grasp on the problem. Since this happens very often, we believe it is something fundamental. The rough hypothesis is that most problems have a "global" tree structure, which can be used to split them up (and that is what a look-ahead cube-solver is good at), and a "local" dag structure, which can be solved by the CDCL-conquer solver. Due to the local dag-structure, look-ahead on its own performs badly, while the global tree-structure can not be captured by CDCL-solvers with their "local senses".

4.3 Specialised heuristics for cubing

Before coming to the problem of *proving* unsatisfiability, some words on the splitting-heuristics used by the cube-solver for [Heule *et al.*, 2016]; the basics of the general theory of branching heuristics for look-ahead solvers one finds in [Kullmann, 2009]. An important idea for the branching-heuristics for look-ahead solvers is to predict the success of future UCPs, and to try to maximise these (balancing this over the two branches). As experimentation shows, together with similarities to worst-case analysis of algorithms: when performing the look-ahead, the basic measure is to count the *new clauses* (only) — the more the better (unit-clauses come from shortened clauses), and the shorter a clause the higher its weight (it's closer to a unit-clause). [Heule *et al.*, 2016] now refined this idea by distinguishing the literals within a new clause, computing some heuristic value for them, which measures how "likely" it seems that this literal becomes `false` via UCP. These ideas have been developed originally for random CNFs, and so in a sense the Pythagorean Triples problem shows behaviour similar to random CNFs (for the splitting).

5 Extracting proofs

If the essence of the solution of the boolean Pythagorean Triples problem would be the number "7825", the numerical information about the point where unsatisfiability is reached, then naturally there would be less pressure on actually having a verifiable *proof*, and one would just wait for independent re-computations, as is the case currently with all really big computations outside of the SAT-realm (only here we have the ability to extract really big proofs). But in this case, there is (at least currently) no independent "mathematical" existence proof, i.e., without our result it wouldn't be even known whether there exists that turning point at all, and thus a real *proof* is needed. Now what is a "proof" here? Since the problems considered are very naturally formulated as SAT problems, we only consider propositional logic; for the wider context of proving mathematical results see [Edwards, 2016]. The most important proof system in the SAT context is *resolution*, which is actually identical to the semantics for clauses: considering two clauses (disjunctions of literals), the only possibility that a new clause can be inferred is that these clauses have exactly one clash (one pair of complementary/negated literals), and then their *resolvent* follows logically, which is the disjunction of all the literals in both clauses minus the pair of clashing literals. One distinguishes between *tree-resolution*, where the derivation of the contradiction has the form of the tree, that is, intermediate results can not be re-used, but have to be re-derived, and full or *dag-resolution*, where the derivation has the form of a dag, that is, now intermediate results can be re-used. Considering the close relation between SAT solving and resolution (as mentioned in Section 3), extracting resolution proofs is the natural first step towards efficient extraction of proofs from solver runs. Now producing such proofs is too expensive for the solver, and thus UCP has to be integrated into the proof format, so that basically just the sequence of learned clauses is needed. However, for fundamental reasons, especially preprocessing techniques, and also symmetry handling, can not be simulated efficiently by resolution (provably). A very powerful proof system is *extended resolution* (ER), which can be understood as "Resolution with Definitions". ER itself is not practical, but a combinatorial extension of it based on "blocked clauses", also integrating UCP, yields the system DRAT [Wetzler *et al.*, 2014], which is the current standard, allowing to easily extract the proofs from the solver run (while verification is reasonable efficient in linear time). The basic proof step is adding a clause, and while resolution only allows to add clauses which preserve all solutions, DRAT allows to add clauses which might require flipping of satisfying assignments on single variables.

6 Conclusion and outlook

We have discussed the C&C method, which might be the strongest method for many hard combinatorial problems. In the future C&C needs a solid foundation concerning problem structure (see the "astounding observation" in Subsection 4.2) and splitting heuristics. Meanwhile an independent verification of the proof took place [Cruz-Filipe *et al.*, 2016]. Theory and practice of this important field is likely just at the beginning; and "proof mining" hopefully at some point will yield valuable insights, into SAT solving and into the problem structure. This leads us to the question about the "meaning" of proofs of 200TB size, where, as to be expected, controversies arose; see [Heule and Kullmann, 2017 to appear] for further discussions. Finally we note, that our application of SAT to Ramsey theory has the special feature of yielding the only known proof of a general existence statement (for higher-order objects, beyond natural numbers). Whether this is just an accident or a deeper aspect is further discussed in [Heule and Kullmann, 2017 to appear], and relates to deep unsolved problems in the foundations of mathematics.

References

[Ahmed *et al.*, 2014] Tanbir Ahmed, Oliver Kullmann, and Hunter Snevily. On the van der Waerden numbers $\mathrm{w}(2;3,t)$. *Discrete Applied Mathematics*, 174:27–51, September 2014.

[Biere *et al.*, 2009] Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Codish *et al.*, 2016] Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the Ramsey number R(4,3,3) using abstraction and symmetry breaking. *Constraints*, 21(3):375–393, July 2016.

[Croot and Lev, 2006] Ernie Croot and Vsevolod F. Lev. Open problems in additive combinatorics. In Andrew Granville, Melvyn B. Nathanson, and József Solymosi, editors, *Additive Combinatorics*, CRM Proceedings & Lecture Notes, pages 207–234. American Mathematical Society, 2006.

[Cruz-Filipe *et al.*, 2016] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. Technical Report arXiv:1610.06984v2 [cs.LO], arXiv, October 2016.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[Edwards, 2016] Chris Edwards. Automating proofs. *Communications of the ACM*, 59(4):13–15, April 2016.

[Graham *et al.*, 1990] Ronald Lewis Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey theory*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & sons, New York, Chichester, Brisbane, 1990.

[Heule and Kullmann, 2017 to appear] Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Communications of the ACM*, 2017, to appear.

[Heule and van Maaren, 2009] Marijn J. H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Biere et al. [2009], chapter 5, pages 155–184.

[Heule *et al.*, 2012] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing (HVC 2011)*, volume 7261 of *Lecture Notes in Computer Science (LNCS)*, pages 50–65. Springer, 2012.

[Heule *et al.*, 2016] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean Pythagorean Triples problem via Cube-and-Conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[Kautz *et al.*, 2009] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Incomplete algorithms. In Biere et al. [2009], chapter 6, pages 185–203.

[Konev and Lisitsa, 2015] Boris Konev and Alexei Lisitsa. Computer-aided proof of Erdös discrepancy properties. *Artificial Intelligence*, 224:103–118, July 2015.

[Kouril and Paul, 2008] Michal Kouril and Jerome L. Paul. The van der Waerden number $W(2,6)$ is 1132. *Experimental Mathematics*, 17(1):53–61, 2008.

[Kullmann, 2009] Oliver Kullmann. Fundaments of branching heuristics. In Biere et al. [2009], chapter 7, pages 205–244.

[Lam, 1991] C. W. H. Lam. The search for a finite projective plane of order 10. *The American Mathematical Monthly*, 98(4):305–318, April 1991.

[Lamb, 2016] Evelyn Lamb. Maths proof smashes size record: Supercomputer produces a 200-terabyte proof – but is it really mathematics? *Nature*, 534:17–18, June 2016.

[Marques-Silva *et al.*, 2009] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [2009], chapter 4, pages 131–153.

[Pipatsrisawat and Darwiche, 2011] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[Schur, 1917] Issai Schur. Über die Kongruenz $x^m + y^m = z^m \pmod p$. *Jahresbericht der Deutschen Mathematikervereinigung*, 25:114–117, 1917.

[Wetzler *et al.*, 2014] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, pages 422–429. Springer, 2014.

[Wilson, 2013] Robin Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Princeton University Press, revised edition, 2013.