Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer

Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek

The University of Texas at Austin, Swansea University, and University of Kentucky

Abstract. The *boolean Pythagorean Triples problem* has been a longstanding open problem in Ramsey Theory: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$? A prize for the solution was offered by Ronald Graham over two decades ago. We solve this problem, proving in fact the impossibility, by using the *Cube-and-Conquer* paradigm, a hybrid SAT method for hard problems, employing both look-ahead and CDCL solvers. An important role is played by dedicated look-ahead heuristics, which indeed allowed to solve the problem on a cluster with 800 cores in about 2 days. Due to the general interest in this mathematical problem, our result requires a formal proof. Exploiting recent progress in unsatisfiability proofs of SAT solvers, we produced and verified a proof in the DRAT format, which is almost 200 terabytes in size. From this we extracted and made available a compressed certificate of 68 gigabytes, that allows anyone to reconstruct the DRAT proof for checking.

1 Introduction

Propositional satisfiability (SAT, for short) is a formalism that allows for representation of all finite-domain constraint satisfaction problems. Consequently, all decision problems in the class NP, as well as all search problems in the class FNP [8,29,34,19], can be polynomially reduced to SAT. Due to great progress with *SAT solvers*, many practically important problems are solved using such reductions. SAT is especially an important tool in hardware verification, for example model checking [7] and reactive systems checking. In this paper we are, however, dealing with a different application of SAT, namely as a tool in computations of configurations in a part of Mathematics called *Extremal Combinatorics*, especially *Ramsey theory.* In this area, the researcher attempts to find various configurations that satisfy some combinatorial conditions, as well as values of various parameters associated with such configurations [47].

One important result of Ramsey theory, the van der Waerden Theorem [43], has been studied by the SAT community [13]. That theorem says that for all natural numbers k and l there is a number n, so that whenever the integers $1, \dots, n$ are partitioned into k sets, there is a set containing an arithmetic progression of length l. A good deal of effort has been spent on specific values of the corresponding number theoretic function, $\mathrm{vdW}(k, l)$. Two results on specific values: $\mathrm{vdW}(2, 6) = 1132$ and $\mathrm{vdW}(3, 4) = 293$, were obtained by M. Kouril [31,30] using specialized FPGA-based SAT solvers. Other examples include the Schur Theorem [41] on sum-free subsets, its generalization known as Rado's Theorem [40], and a generalization of van der Waerden numbers [4]. In this paper we investigate two areas:

1. We show the "boolean Pythagorean triples partition theorem" (Theorem 1), an analogue of Schur's Theorem.
2. We develop methods to compute numbers in Ramsey theory by SAT solvers.

Cooper and Overstreet [9] discussed the Ramsey property for Pythagorean triples, which are $(a, b, c) \in \mathbb{N}^3$ with $a^2 + b^2 = c^2$. They studied the question whether all partitions of integers into two parts must exhibit one part with a Pythagorean triple. If for some $n > 2$ every partition of $1, \dots, n$ into two parts ensures a Pythagorean triple in one part, then that property holds for all partitions of $1, \dots, m$ for $m > n$ and hence for partitioning the entire set of natural numbers into two parts. The main mathematical result of this paper is:

Theorem 1. *The set* $\{1, \dots, 7824\}$ *of natural numbers can be partitioned into two parts, such that no part contains a Pythagorean triple* $(a, b, c \in \mathbb{N}$ *with* $a^2 + b^2 = c^2)$*, while this is impossible for* $\{1, \dots, 7825\}$*.*

To emphasize, the situation of Theorem 1 is not as in previous applications of SAT to Ramsey theory, where SAT only "filled out the numerical details", but the existence of these numbers was not known (and as such is a good success of Automated Theorem Proving). The following generalization is a form of the Erdős and Graham problem [17,11].

Conjecture 1. For every $k \geq 2$ there exist $\mathrm{Ptn}(k) \in \mathbb{N}$ (the "Pythagorean triple number"), such that $\{1, \dots, \mathrm{Ptn}(k) - 1\}$ can be partitioned into k parts with no part containing a Pythagorean triple, while this is impossible for $\{1, \dots, \mathrm{Ptn}(k)\}$.

The proof of Theorem 1 is by considering two SAT problems. One showing that $\{1, \dots, 7824\}$ can be partitioned into two parts such that no part contains a Pythagorean triple (i.e., the case $n = 7824$ is satisfiable). The other one showing that any partitioning of $\{1, \dots, 7825\}$ into two parts contains a Pythagorean triple (i.e., the case $n = 7825$ is unsatisfiable). Now a Pythagorean triple-free partition for $n = 7824$ is checkable in a second, but the *absence* of such a partition for $n = 7825$ requires a more "durable proof" than just the statement that we run a SAT solver (in some non-trivial fashion!) which answered UNSAT — to become a mathematically accepted theorem, our assertion for $n = 7825$ carries a stronger burden of proof. Fortunately, the SAT community has spent a significant effort to develop techniques that allow to extract, out of a failed attempt to get a satisfying assignment, an actual *proof of the unsatisfiability.*

It is worth noting the similarities and differences to the endeavours of extending mathematical arguments into actual *formal proofs*, using tools like *Mizar* [1] and *Coq* [2]. Cases, where intuitions (or convictions) about completeness of mathematical arguments fail, are known [45]. So T. Hales in his project *flyspeck* [3] extracted and verified his own proof of the *Kepler Conjecture.* Now the core of the argument in such examples has been constructed by mathematicians. Very different from that, the proofs for unsatisfiability coming from SAT solvers are, from a human point of view, a giant heap of random information (no direct understanding is involved). But we don't need to search for the proof — the present generation of SAT solvers supports emission of unsatisfiability proofs and standards for such proofs exist [46], as well as checkers that the proof is valid. However the proof that we will encounter in our specific problem is of very large size. In fact, even *storing* it is a significant task, requiring significant compression. We will tackle these problems in this paper.

2 Preliminaries

CNF Satisfiability. For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a finite set of literals; so it may contain complementary literals, in which case the clause is tautological. The empty clause is denoted by $\bot$. If convenient, we write a clause as a disjunction of literals. Since a clause is a set, no literal occurs several times, and the order of literals in it does not matter. A (CNF) *formula* is a conjunction of clauses, and thus clauses may occur several times, and the order of clauses does matter; in many situations these distinctions can be ignored, for example in semantical situations, and then we consider in fact finite sets of clauses.

A *partial assignment* is a function τ that maps a finite set of literals to $\{0,1\}$, such that for $v \in \{0,1\}$ holds $\tau(x) = v$ if and only if $\tau(\bar{x}) = \neg v$. A clause C is satisfied by τ if $\tau(l) = 1$ for some literal $l \in C$, while τ satisfies a formula F if it satisfies every clause in F. If a formula F contains $\bot$, then F is unsatisfiable. A formula F *logically implies* another formula F', denoted by $F \models F'$, if every satisfying assignment for F also satisfies F'. A transition $F \rightsquigarrow F'$ is *sat-preserving*, if either F is unsatisfiable or both F, F' are satisfiable, while the transition if *unsat-preserving* if either F is satisfiable or both F, F' are unsatisfiable. Stronger, F, F' are *satisfiability-equivalent* if both formulas are satisfiable or both unsatisfiable, that is, iff the transition $F \rightsquigarrow F'$ is both sat- and unsat-preserving. We note that if $F \models F'$, then $F \rightsquigarrow F'$ is sat-preserving, and that $F \rightsquigarrow F'$ is sat-preserving iff $F' \rightsquigarrow F$ is unsat-preserving. Clause addition is always unsat-preserving, clause elimination is always sat-preserving.

Resolution and Extended Resolution. The resolution rule (see [18, Subsections 1.15-1.16]) infers from two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$ the *resolvent* $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, by resolving on variable x. C is logically implied by any formula containing C_1 and C_2. For a given CNF formula F, the *extension rule* [42] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding the *extended resolution clauses* $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula. The addition of these clauses is sat-equivalent.

Unit Propagation. For a CNF formula F, *unit propagation* simplifies F based on unit clauses; that is, it repeats the following until fixpoint: if there is a unit clause $\{l\} \in F$, remove all clauses that contain the literal l from the set F and remove the literal $\bar{l}$ from the remaining clauses in F. This process is sat-equivalent. If unit propagation on formula F produces complementary units $\{l\}$ and $\{\bar{l}\}$, we say that unit propagation *derives a conflict* and write $F \vdash_1 \bot$ (this relation also holds if $\bot$ is already in F).

Ordinary resolution proofs (or "refutations" – derivations of the empty clause) just add resolvents. This is too inefficient, and is extended via unit propagation as follows. For a clause C let $\overline{C}$ denote the conjunction of unit clauses that falsify all literals in C. A clause C is an *asymmetric tautology* with respect to a CNF formula F if $F \wedge \overline{C} \vdash_1 \bot$. This is equivalent to the clause C being derivable from F via *input resolution* [20]: a sequence of resolution steps using for every resolution step at least one clause of F. So addition of resolvents is generalised by addition of asymmetric tautologies (where addition steps always refer to the current (enlarged) formula, the original axioms plus the added clauses). Asymmetric tautologies, also known as *reverse unit propagation* (RUP) clauses, are the most common learned clauses in *conflict-driven clause learning* (CDCL) SAT solvers (see [38, Subsection 4.4]). We note that this extension is irrelevant from the proof-complexity point of view, since it can be simulated by linearly many resolution steps, however for practical applications exploitation of the power of fast unit propagation algorithms is essential.

RAT clauses. We are seeking to add sat-preserving clauses beyond logically implied clauses. The basic idea is as follows (proof left as instructive exercise):

Lemma 1. *Consider a formula F, a clause C and a literal $x \in C$. If for all $\bar{x} \in D \in F$ holds $F \models C \cup (D \setminus \{\bar{x}\})$, then addition of C to F is sat-preserving.*

In order to render the condition $F \models C \cup (D \setminus \{\bar{x}\})$ polytime-decidable, we stipulate that the right-hand clause must be derivable by input resolution:

Definition 1 ([28]). *Consider a formula F, a clause C and a literal $x \in C$ (the "pivot"). We say that C has* RAT *("Resolution asymmetric tautology") on x w.r.t. F if for all $D \in F$ with $\bar{x} \in D$ holds that $F \wedge \overline{C} \wedge \overline{D \setminus \{\bar{x}\}} \vdash_1 \bot$.*

By Lemma 1, addition of RAT-clauses is sat-preserving. Every non-empty asymmetric tautology C for F has RAT on any $x \in C$ w.r.t. F. It is also easy to see that the three extended resolution clauses are RAT clauses (using the new variable for the pivot literals). All preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [28].

3 Proofs of Unsatisfiability

A *proof of unsatisfiability* (also called a *refutation*) for a formula F is a sequence of sat-preserving transitions which ends with some formula containing the empty clause. There are currently two prevalent types of unsatisfiability proofs: *resolution proofs* and *clausal proofs*. Both do not display the sequence of transformed

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRAT proof

```

  -1 0
d -1 2 4 0
   2 0
   0
```

Fig. 1. Left, a formula in DIMACS CNF format, the conventional input for SAT solvers which starts with `p cnf` to denote the format, followed by the number of variables and the number of clauses. Right, a DRAT refutation for that formula. Each line in the proof is either an addition step (no prefix) or a deletion step identified by the prefix "`d`". Spacing is used to improve readability. Each clause in the proof must be a RAT clause using the first literal as pivot, or the empty clause as an asymmetric tautology.

formulas, but only list the axioms (from F) and the additions and (possibly) deletions. Several formats have been designed for resolution proofs [48,16,5] (which only add clauses), but they all share the same disadvantages. Resolution proofs are often huge, and it is hard to express important techniques, such as conflict clause minimization, with resolution steps. Other techniques, such as bounded variable addition [37], cannot be polynomially-simulated by resolution at all. Clausal proof formats [46,44,23] are syntactically similar; they involve a sequence of clauses that are claimed to be sat-preserving, starting with the given formula. But now we might add clauses which are not logically implied, and we also might remove clauses (this is needed now in order to enable certain additions, which might depend on global conditions).

Definition 2 ([46]). *A* DRAT proof *("Deletion Resolution Asymmetric Tautology") for a formula F is a sequence of additions and deletions of clauses, starting with F, such that each addition is the addition of a RAT clause w.r.t. the current formula (the result of additions and deletions up to this point), or, in case of adding the empty clause, unit-clause propagation on the current formula yields a contradiction. A* DRAT refutation *is a DRAT proof containing $\bot$.*

DRAT refutations are correct proofs of unsatisfiability (based on Lemma 1 and the fact, that deletion of clauses is always sat-preserving). Furthermore they are checkable in cubic time. Since the proof of Lemma 1 is basically the same as the proof for [32, Lemma 4.1], by adding unit propagation appropriately one can transfer [32, Corollary 7.2] and prove, that the power of DRAT refutations is up to polytime transformations the same as the power of Extended Resolution.

Example 1. Figure 1 shows an example DRAT refutation. Consider the CNF formula $F = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (b \vee c \vee \bar{d}) \wedge (\bar{b} \vee \bar{c} \vee d) \wedge (a \vee c \vee d) \wedge (\bar{a} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$, shown in DIMACS format in Fig. 1 (left), where 1 represents a, 2 is b, 3 is c, 4 is d, and negative numbers represent negation. The first clause in the proof, $(\bar{a})$, is a RAT clause with respect to F because all possible resolvents are asymmetric tautologies:

$$\begin{array}{rcl} F \wedge (a) \wedge (\bar{b}) \wedge (c) \vdash_1 \bot & \text{using} & (a \vee b \vee \bar{c}) \\ F \wedge (a) \wedge (\bar{c}) \wedge (\bar{d}) \vdash_1 \bot & \text{using} & (a \vee c \vee d) \\ F \wedge (a) \wedge (b) \wedge (d) \vdash_1 \bot & \text{using} & (a \vee \bar{b} \vee \bar{d}) \end{array}$$

4 Cube-and-Conquer Solving

Arguably the most effective method to solve most hard combinatorial problems via SAT technology is the *cube-and-conquer* paradigm [25], abbreviated by C&C, due to strong performance and easy parallelization. C&C consists of two phases. In the first phase, a look-ahead SAT solver [26] partitions the problem into many (millions of) subproblems. These subproblems, expressed as "cubes" (conjunctions) of the decisions (the literals set to true), are solved using a CDCL solver, also known as the "conquer" solver. The intuition behind this combination of paradigms is that look-ahead heuristics focus on global decisions, while CDCL heuristics focus on local decisions. Global decisions are important to split the problem, while local decisions are effective when there exist a short refutation. So the idea behind C&C is to partition the problem until a short refutation arises. C&C can solve hard problems much faster than either pure look-ahead or pure CDCL. The problem with pure look-ahead solving is that global decisions become poor decisions when a short refutation is present, while pure CDCL tends to perform rather poor when there exist no short refutation. We will demonstrate that C&C outperforms pure CDCL and pure look-ahead in Section 6.2. Apart from improved performance on a single core, C&C allows for easy parallelization. The subproblems are solved independently, so they are distributed on a large cluster. In practice this results in almost a linear speedup. Other parallel SAT solving paradigms, such as portfolio solvers, don't scale that well, making C&C the preferred approach for very hard problems.

There are two C&C variants: solving one cube per solver and solving multiple cubes by a single incremental solver. The first approach allows solving cubes in parallel, while the second approach allows for reusing heuristics and learned clauses while solving multiple cubes. In our computation we combined them, via a two-staged splitting, to exploit both advantages: having much parallelism, while reusing heuristics and learned clauses as well. First the problem is split into 10^6 cubes, and then each corresponding subproblem is split again and fed to the incremental solver, which receives the input (sub-)formula and a sequence of cubes[1]. For each cube, the problem is solved under the assumption that the cube is true. If the solver can refute that assumption, it returns a subset of the cube that was required to find a refutation. Learned clauses may not depend on all literals in the cube. Hence, they can be useful for solving future cubes as well. The SAT solver keeps some learned clauses, typically the shorter ones, while solving multiple cubes.

5 Solving the boolean Pythagorean Triples Problem

Our framework for solving hard problems consists of five phases: encode, transform, split, solve, and validate. The focus of the encode phase is to make sure that representation of the problem as SAT instance is valid. The transform phase

[1] In practice this is done using a single incremental CNF file. For details about the format, see `http://www.siert.nl/icnf/`.

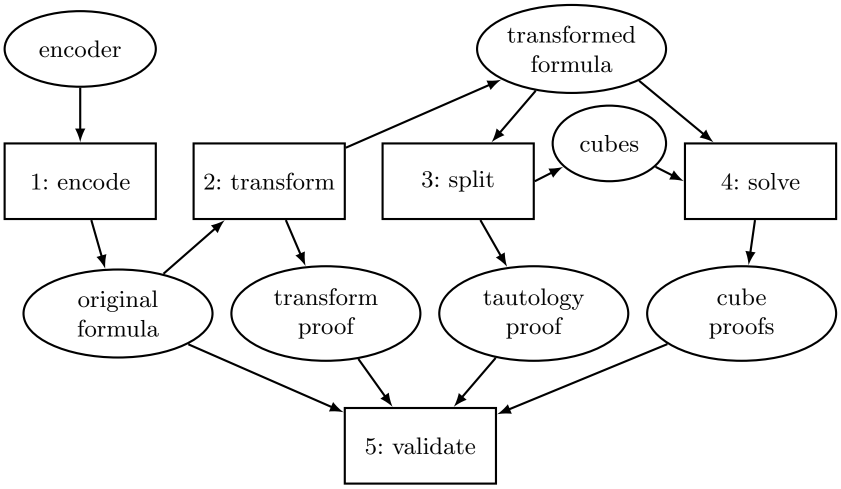


Fig. 2. Illustration of the framework to solve hard combinatorial problems. The phases are shown in the rectangle boxes, while the input and output files for these phases are shown in oval boxes.

reformulates the problem to reduce the computation costs of the later phases. The split phase partitions the transformed formula into many, possibly millions of subproblems. The subproblems are tackled in the solve phase. The validation phase checks whether the proofs emitted in the prior phases are a valid refutation for the original formula. Figure 2 shows an illustration of the framework.

5.1 Encode

The first phase of the framework focusses on making sure that the problem to be solved is correctly represented into SAT. In the second phase the representation will be optimized. The DRAT proof format can express all transformations.

Formula F_n expresses whether the natural numbers up to n can be partitioned into two parts with no part containing a triple (a, b, c) such that $a^2 + b^2 = c^2$. One set will be called the positive part, while the other will be called the negative part. F_n uses Boolean variables x_i with $i \in \{1, \ldots, n\}$. The assignment x_i to true / false, expresses that i occurs in the positive / negative part, respectively. For each triple (a, b, c) such that $a^2+b^2 = c^2$, there is a constraint $\textsc{NotEqual}(a, b, c)$ in F_n, or in clausal form: $(x_a \vee x_b \vee x_c) \wedge (\bar{x}_a \vee \bar{x}_b \vee \bar{x}_c)$.

5.2 Transform

The goal of the transformation phase is to massage the initial encoding to execute the later phases more efficiently. A proof for the transformations is required to ensure that the changes are valid. Notice that a transformation that would be helpful for one later phase, might be harmful for another phase. Selecting transformations to apply is therefore typically a balance between different tradeoffs. For example, bounded variable elimination [15] is a preprocessing technique that tends to speed up the solving phase. However, this technique is generally harmful for the splitting phase as it obscures the look-ahead heuristics.

We applied two transformation steps to the formulas used during our experiments, F_{7824} and F_{7825} . The first step is blocked clause elimination (BCE) [27]. BCE on these formulas has the following effect (due to Pythagorean triples intersecting in at most one element): if a, b, or c occurs only in one constraint $\textsc{NotEqual}(a, b, c)$, then eliminate this constraint. BCE is applied until fixpoint: removing a constraint $\textsc{NotEqual}(a, b, c)$ because a occurs once, reduces the occurrences of b and c by one. As a result b or c may occur only once after the removal, allowing further elimination. A solution for the formula after the transformation may not satisfy the original formula, however this can be easily repaired [27]. The numerical effects of this reduction are as follows: F_{7824} has 6492 (occurring) variables and 18930 clauses, F_{7825} has 6494 variables and 18944 clauses, while after BCE-reduction we get 3740 variables and 14652 clauses resp. 3745 variables and 14672 clauses.

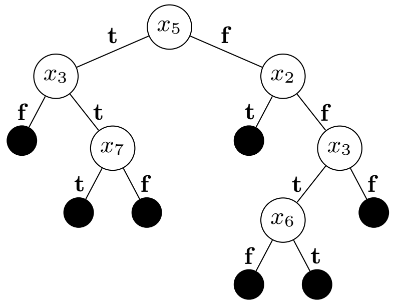
The second transformation is symmetry breaking [10]. The Pythagorean Triples encoding has one symmetry: the two parts are interchangeable. To break this, we can pick an arbitrary variable x_i and assign it to true (or, equivalently, put in the positive part). In practice it is best to pick the variable x_i that occurs most frequently in F_n. For the two formulas used during our experiments, the most occurring variable is x_{2520} which was used for symmetry breaking.

Bounded variable elimination (a useful transformation in general) was not applied. Bounded variable elimination substitutes away all variables x_i if the resulting formula after substitution has fewer or an equal number of clauses. For our Pythagorean Triples encoding, initially variables occurring positively at most three times would be eliminated. However these substitutions add clauses that have a length larger than three. This would seriously harm the look-ahead heuristics, as the specialized 3-SAT heuristics can no longer be used.

In order to validate the transformations applied during this phase, a proof is required for each transformation. We will refer to this proof as the transformation proof. The first part of this proof, expressing BCE is easy: Simply list all clauses that are eliminated as clause deletion steps. The second part is more complicated. Symmetry breaking can be expressed in the DRAT format, but it is tricky. A recent paper [24] explains how to construct this part of the transformation proof.

5.3 Split

Partitioning is crucial to solve hard combinatorial problems. Effective partitioning is based on global heuristics [25] — in contrast to the "local" heuristics used in CDCL solvers. The result of partitioning is a binary branching tree of which the leaf nodes represent a subproblem of the original problem. The subproblem is constructed by adding the conjunction of decisions that lead to the leaf as unit clauses. Figure 3 shows such a partitioning as a binary tree with seven leaf nodes (left) and the corresponding list of seven cubes (right). The cubes are shown in the `inccnf` format that is used for incremental solvers to guide their search.



cube file in `inccnf` format

```
a  5 -3  0
a  5  3  7  0
a  5  3 -7  0
a -5  2  0
a -5 -2  3 -6  0
a -5 -2  3  6  0
a -5 -2 -3  0
```

Fig. 3. A binary branching tree (left) with the decision variables in the nodes and the polarity on the edges. The corresponding cube file (right) in the `inccnf` format. The prefix `a` denotes *assumptions*. Positive numbers express positive literals, while negative numbers express negative literals. Each cube (line) is terminated with a `0`.

Splitting heuristics are crucial in solving a problem efficiently. In practice, the best heuristics are based on *look-aheads* [26,33]. In short, a look-ahead refers to assigning a variable to a truth value followed by unit propagation and measuring the changes to the formula during the propagation. It remains to find good measures. The simplest measure is to count the number of assigned variables; measures like that can be used for tie-breaking, but as has been realised in the field of heuristics [33], the expected future gains for unit-clause propagation, given by *new short clauses*, are more important than the current reductions. The default heuristic in C&C, which works well on most hard-combinatorial problems, weighs all new clauses using weights based on the length of the new clause (with an exponential decay of the weight in the length). However for our Pythagorean Triples encoding, using a refinement coming from random 3-SAT turned out to be more powerful. Here all newly created clauses are binary, i.e., ternary clauses that become binary during the look-ahead. The weight of a new binary clause depends on the occurrences of its two literals in the formula, estimating how likely the become falsified. This better performance is not very surprising as the formulas F_n are somewhat related to random 3-SAT formulas: i) all clauses have length three; and ii) the distribution of the occurrences of literals is similar.

5.4 Details regarding the heuristics

The heuristics used for splitting extends the *recursive weight heuristics* [39], based on earlier work [36,35,14,12], by introducing minimal and maximal values α, β, and choosing different parameters, optimized for the special case at hand. A look-ahead on literal l measures the difference between a formula before and after assigning l to true followed by simplification. Let F (or F_l) denote the formula before (or after) the look-ahead on l, respectively. We assume that F and F_l are fully simplified using unit propagation. Thus $F_l \setminus F$ is the set of new clauses, and the task is to weigh them; we note that each clause in $F_l \setminus F$ is binary. Each literal is assigned a heuristic value $h(l)$ and the weight $w_{y \vee z}$ for $(y \vee z) \in F_l \setminus F$ is $h(\bar{y}) \cdot h(\bar{z})$. The values of $h(l)$ are computed using multiple iterations $h_0(l), h_1(l), \ldots$, choosing the level with optimal performance, balancing the predictive power of the heuristics versus the cost to compute it. The idea of the heuristic values $h_i(l)$ is to approximate how strongly the literal l is forced to true by the clauses containing l (via unit propagation). First, for all literals l, $h_0(l)$ are initialized on 1: $h_0(x) = h_0(\bar{x}) = 1$. At each step, the heuristics values $h_i(x)$ are scaled using the average value μ_i:

$$\mu_i = \frac{1}{2n} \sum_{x \in \mathrm{var}(F)} \big(h_i(x) + h_i(\bar{x})\big). \tag{1}$$

Finally, in each next iteration, the heuristic values $h_{i+1}(x)$ are computed in which literals y get weight $h_i(\bar{y})/\mu_i$. Weight γ expresses the relative importance of binary clauses. This weight could also be seen as the heuristic value of a falsified literal. Additionally, we have two other parameters, α expressing the minimal heuristic value and β expressing maximum heuristic value.

$$h_{i+1}(x) = \max(\alpha, \min(\beta, \sum_{(x \vee y \vee z) \in F} \Big(\frac{h_i(\bar{y})}{\mu_i} \cdot \frac{h_i(\bar{z})}{\mu_i}\Big) + \gamma \sum_{(x \vee y) \in F} \frac{h_i(\bar{y})}{\mu_i})). \tag{2}$$

In each node of the branching tree we compute $h(l) := h_4(l)$ for all literals occurring in the formula. We use $\alpha = 8$, $\beta = 550$, and $\gamma = 25$. The "magic" constants differ significantly compared to the values used for random 3-SAT formulas where $\alpha = 0.1$, $\beta = 25$, and $\gamma = 3.3$ appear optimal [39]. The branching variable x chosen is a variable with maximal $H(x) \cdot H(\bar{x})$, where $H(l) := \sum_{y \vee z \in F_l \setminus F} w_{y \vee z}$.

5.5 Solve

The solving phase is the most straightforward part of the framework. It takes the transformed formula and cube files as input and produces a proof of unsatisfiability of the transformed formula. Two different approaches can be distinguished in general: one for "easy" problems and one for hard problems. A problem is considered easy when it can be solved in reasonable time, say within a day on a single core. In that case, a single cube file can be used and the incremental SAT solver will emit a single proof file. The more interesting case is when problems are hard and two levels of splitting are required, allowing parallel solving.

The boolean Pythagorean triples problem F_{7825} is very hard and required two level splitting: the total runtime was approximately $35{,}000$ CPU hours, or equivalently 4 CPU years. Any problem requiring that amount of resources has to be solved in parallel. The first level consists of partitioning the problem into a million subproblems, for details see Section 6.2. Each subproblem is represented by a cube φ_i with $i \in \{1, \ldots, 10^6\}$ expressing a conjunction of decisions. On the second level of splitting, each subproblem $F_{7825} \wedge \varphi_i$ is partitioned again using the same look-ahead heuristics. In contrast to the first level, the cubes generated on the second level are not used to create multiple subproblems. Instead, the second level cubes are provided to a specialized SAT solver together with a subproblem $F_{7825} \wedge \varphi_i$. The second level cubes are used to guide the CDCL solver. The advantage of guiding the CDCL solver is that learned clauses computed while solving one cube and can be reused when solving another cube.

For each subproblem $F_{7825} \wedge \varphi_i$, the SAT solver produces a DRAT refutation. Most state-of-the-art SAT solvers currently support the emission of such proofs. One can check that the emitted proof of unsatisfiability is valid for $F_{7825} \wedge \varphi_i$. In this case, no changes to the proof logging of the solver are required. However, in order to create an unsatisfiability proof of F_{7825} by concatenating the proofs of subproblems, all lemmas generated while solving $F_{7825} \wedge \varphi_i$ need to be extended with $\overline{\varphi_i}$ and the SAT solver is not allowed to delete clauses from F_{7825}.

5.6 Validate

The last phase of the framework validates the results of the earlier phases. First, the encoding into SAT needs to be validated. This can be done by proving that the encoding tool is correct using a theorem prover. Alternatively, a small program can be implemented whose correctness can be checked manually. For example, our encoding tool consists of only 19 lines of C code. For details and validation files, check out `http://www.cs.utexas.edu/~marijn/ptn/`.

The second part consists of checking the three types of DRAT proofs produced in the earlier phases: the transformation, tautology, and the cube proofs. DRAT proofs can be merged easily by concatenating them. The required order for merging the proofs is: transformation proof, cube proofs, and tautology proof.

Transformation Proof The transformation proof expresses how the initial formula, created by the encoder, is converted into a formula that is easier to solve. This part of the proof is typically small. The latest version of the `drat-trim` checker supports validating transformation proofs without requiring the other parts of the proof, based on a compositional argument [22].

Cube Proofs The core of the validation is checking whether the negation of each cube, $\overline{\varphi}_i$, is implied by the transformed formula. Since we partitioned the problem using 10^6 cubes, there are 10^6 of cube proofs. We generated and validated them all. However, their total size is too large to share: almost 200 terabyte in the DRAT format. We tried to compress the proof using a range of dedicated clause compression techniques [21] combined with state-of-the-art general purpose tools, such as `bzip2` or `7z`. After compression the total proof size was still 14 terabytes. So instead we provide the cube files for the subproblems as a certificate. Cube files can be compressed heavily, because they form a tree. Instead of storing all cubes as a list of literals, shown as in Figure 3, it is possible to store only one literal per cube. Storing the literal in a binary format [21] followed by `bzip2` allowed us to store all the cube files using “only” 68 gigabytes of disk space. Our specialized `glucose` 3.0 variant accepts cube files as additional input and can reproduce the DRAT proofs in about 13,000 CPU hours. Checking these proofs requires about 16,000 CPU hours, so reproducing the DRAT proofs almost doubles the validation effort. This is probably a smaller burden than downloading and decompressing many terabytes of data.

Tautology Proof A cube partitioning is valid, i.e., covers the complete search space, if the disjunction of cubes is a tautology. This needs to be checked during the validation phase. Checking this can be done by negating the disjunction of cubes and feed the result to a CDCL solver which supports proof logging. If the solver can refute the formula, then the disjunction of cubes is a tautology. We refer to the proof emitted by the CDCL solver as the *tautology proof.* This tautology proof will be part of the final validation effort.

6 Results

This section offers details of solving the boolean Pythagorean Triples problem[2]. All experiments were executed on the Stampede cluster[3]. Each node on this cluster consists of an Intel Xeon Phi 16-core CPU and 32 Gb memory. We used cube solver `march_cc` and conquer solver `glucose` 3.0 during our experiments.

6.1 Heuristics

In our first attempt to solve the Pythagorean triples problem, we partitioned the problem (top-level and subproblems) using the default decision heuristic in the cube solver `march_cc` for 3-SAT formulas. After some initial experiments, we estimated that the total runtime of solving F_{7825} would be roughly 300,000 CPU hours on our cluster. To reduce the computation costs, we (manually) optimized the magic constants in `march_cc`, resulting in the heuristic presented in Section 5.4. The new heuristics reduced the total runtime to 35,000 CPU hours, so almost an order of magnitude. Table 1 shows the results of various heuristics on five randomly selected subproblems. We optimized `march_cc` in favor of the other heuristics to make the comparison more fair. In particular, we turned off look-ahead preselection, which is helpful for the new heuristics (and thus used in the computation), but harmful for the other heuristics.

6.2 Cube and Conquer

The first step of the solving phase was partitioning the transformed formula into many subproblems using look-ahead heuristics. Our cluster account allowed for running on 800 cores in parallel. We decided to partition the problem into a multiple of 800 to perform easy parallel execution: exactly 10^6. Partitioning the formula into 10^6 subproblems ensured that the conquer time of solving most subproblems is less than two minutes, a runtime with the property that proof validating can be achieved in a time similar to the solving time.

[2] Files and tools can be downloaded at `http://www.cs.utexas.edu/~marijn/ptn/`

[3] `https://www.tacc.utexas.edu/systems/stampede`

Table 1. Solving times for C&C using different look-ahead heuristics and pure CDCL. The top left, bottom left, and right numbers expresses the cube, conquer, and their sum times, respectively. *Ptn 3-SAT* is 3-SAT heuristics optimized for Pythagorean triple problems; *rnd 3-SAT* is the 3-SAT heuristics optimized for random 3-SAT (default); *count bin* is the sum of new binary clauses; and *count var* is the number of assigned variables.

cube #	*Ptn 3-SAT*		*rnd 3-SAT*		*count bin*		*count var*		*pure CDCL*
104302	152.98 75.50	**228.48**	608.46 174.94	783.40	263.23 150.71	413.94	789.43 263.79	1053.22	1372.87
268551	74.03 33.83	**107.86**	92.09 48.82	140.91	98.93 55.83	154.76	487.45 220.27	707.72	150.06
934589	136.94 74.44	**211.38**	206.28 121.99	328.27	156.78 107.16	263.94	529.21 235.70	764.91	631.91
950025	143.69 74.09	**217.78**	152.49 99.67	252.16	203.18 138.09	341.27	550.47 226.99	777.46	330.61
980757	112.22 30.41	**142.63**	170.34 53.90	224.24	181.14 60.53	241.67	685.04 160.93	845.97	155.57

A simple way of splitting a problem into 10^6 subproblems is to build a balanced binary branching tree of depth 20. However, using a balanced binary branching tree results in poor performance on hard combinatorial problems [25]. A more effective partitioning heuristic picks the leaf nodes such that the number of assigned variables in those nodes are equal. We observed that the best heuristics for Pythagorean Triples formulas is to count the number of binary clauses in each node. Recall that all clauses in the transformed formula are ternary. Selecting nodes in the decision tree that have about $3,000$ binary clauses resulted in 10^6 subproblems. Figure 4 (left) shows a histogram of the depth of the branching tree (or, equivalently, the size of the cube) of the selected nodes. Notice that the smallest cube has size 12 and the largest cubes have size 49.

Figure 4 (right) shows the time for the cube and conquer runtimes averaged per size of the cubes. The peak average of the cube runtime is around size 24, while the peak of the conquer runtime is around size 26. The cutoff heuristics of the cube solver for second level splitting were based on the number of unassigned variables, 3450 variables to be precise.

A comparison between the cube, conquer, and validation runtimes is shown in Figure 5. The left scatter plot compares cube and conquer runtimes. It shows that within our experimental setup the cube computation is about twice as expensive compared to the conquer computation. The right scatter plot compares the validation and conquer runtimes. It shows that these times are very similar. Validation runtimes grow slightly faster compared to conquer runtimes. The average cube, conquer, and validation times for the million subproblems are 78.87, 47.52, and 60.62 seconds, respectively.

Figure 6 compares the cube+conquer runtimes to solve the subproblems with the runtimes of pure CDCL (using `glucose` 3.0) and pure look-ahead (using `march_cc`). The plot shows that cube+conquer clearly outperforms pure CDCL.

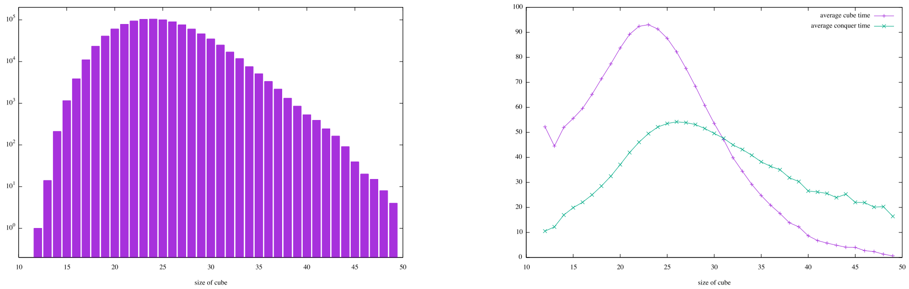


Fig. 4. Left, a histogram (logarithmic) of the cube size of the 10^6 subproblems. Right, average runtimes per size for the cube and conquer phase to solve subproblems.

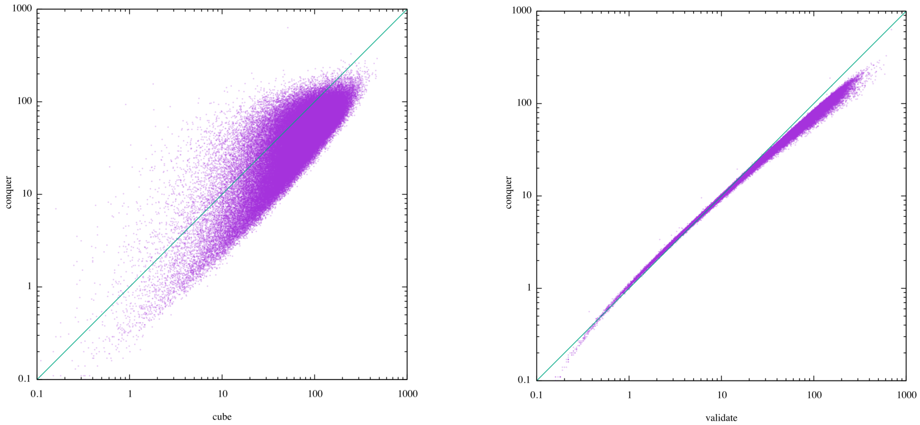


Fig. 5. Left, a scatter plot comparing the cube and conquer time per subproblem. Right, a scatter plot comparing the validation and conquer time per subproblem.

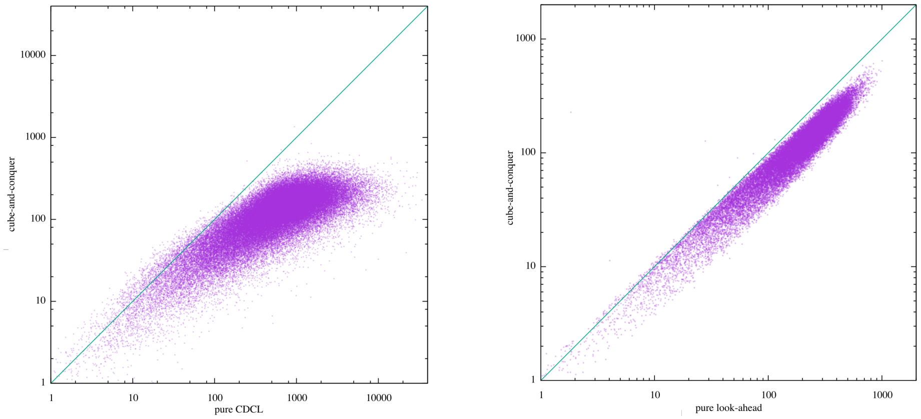


Fig. 6. Scatterplots comparing C&C to pure CDCL (left) and pure look-ahead (right) solving methods on the Pythagorean Triples subproblems.

Frequently speed-ups of two orders or magnitude can be observed on the harder instances. Pure look-ahead is also slower compared to cube+conquer, but the differences are smaller: on average cube+conquer is about twice as fast.

6.3 Extreme Solutions

Of the 10^6 subproblems that were created during the splitting phase, only one subproblem is satisfiable for the extreme case, i.e., $n = 7824$. This suggests that the formula after symmetry breaking has a big *backbone*. A variable belongs to backbone of a formula if it is assigned to the same truth value in all solutions. We computed the backbone of F_{7824}, which consists of 2304 variables. The backbone reveals why it is impossible to avoid Pythagorean Triples indefinitely when partitioning the natural numbers into two parts: variables x_{5180} and x_{5865} are both positive in the backbone, forcing x_{7825} to be negative due to $5180^2 + 5865^2 = 7825^2$. At the same time, variables x_{625} and x_{7800} are both negative in the backbone forcing x_{7825} to be positive due to $625^2 + 7800^2 = 7825^2$.

A satisfying assignment does not necessarily assign all natural numbers up to 7824 that occur in Pythagorean Triples. For example, we found a satisfying assignment that assigns only 4925 out of the 6492 variables occurring in F_{7824}. So not only is F_{7824} satisfiable, but it has a huge number of solutions.

7 Conclusions

We solved and verified the boolean Pythagorean Triples problem using C&C. The total solving time was about 35,000 hours and the verification time about 16,000 hours. Since C&C allows for massive parallelization, resulting in almost linear-time speedups, the problem was solved altogether in about two days on our cluster. Apart from strong computational resources, dedicated look-ahead heuristics were required to achieve these results. In future research we want to further develop effective look-ahead heuristics that will work for such hard combinatorial problems out of the box. We expect that parallel computing combined with look-ahead splitting heuristics will make it feasible to solve many other hard combinatorial problems that are too hard for existing techniques. Moreover, we argue that solutions to such problems require certificates that can be validated by the community — similar to the certificate we provided for the boolean Pythagorean Triples problem. A fundamental question is whether Theorem 1 has a “mathematical” (human-readable) proof, or whether the gigantic (sophisticated) case-distinction, which is at the heart of our proof, is the best there is?! It is conceivable that Conjecture 1 is true, but for each k has only proofs like our proof, where the size of these proofs is growing so quickly, that Conjecture 1 is actually not provable in current systems (like ZFC).

Acknowledgements The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing grid resources that have contributed to the research results reported within this paper.

References

1. Mizar proof checker, accessed: November 2015.
2. Coq proof manager, accessed: November 2015.
3. The site of *flyspeck* project, the formal verification of the proof of Kepler Conjecture, accessed: November 2015.
4. Tanbir Ahmed, Oliver Kullmann, and Hunter Snevily. On the van der Waerden numbers w$(2; 3, t)$. *Discrete Applied Mathematics*, 174:27–51, September 2014.
5. Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
6. 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.
7. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
8. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, 1971.
9. Joshua Cooper and Ralph Overstreet. Coloring so that no Pythagorean triple is monochromatic. 2015. arXiv:1505.02222.
10. James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proc. KRö96, 5th Int. Conf. on Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.
11. Ernie Croot and Vsevolod F. Lev. Open problems in additive combinatorics, 2011. `http://people.math.gatech.edu/~croot/E2S-01-11.pdf`. Accessed January 2016.
12. Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-SAT formulae. In *Theory and Applications of Satisfiability Testing 2003*, pages 486–501, 2003.
13. Michael R. Dransfield, Victor W. Marek, and Mirosław Truszczyński. Satisfiability and computing van der Waerden numbers. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, pages 1–13, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
14. Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 248–253, 2001.
15. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT 2005*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
16. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
17. Pál Erdös and Ronald Lewis Graham. *Old and new problems and results in combinatorial number theory*. Monographie de l'Enseignement mathématique. L'Enseignement mathématique, Université de Genève, Genève, 1980.
18. John Franco and John Martin. A history of satisfiability. In Biere et al. [6], chapter 1, pages 3–74.
19. Michael R. Garey and David S. Johnson. *Computers and Intractability / A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

20. Lawrence J. Henschen and Lawrence Wos. Unit refutations and Horn sets. *Journal of the Association for Computing Machinery*, 21(4):590–605, October 1974.
21. Marijn J. H. Heule and Armin Biere. Clausal proof compression. In *11th International Workshop on the Implementation of Logics*, 2015.
22. Marijn J. H. Heule and Armin Biere. Compositional propositional proofs. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 444–459, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
23. Marijn J. H. Heule, Warren A. Hunt, Jr, and Nathan Wetzler. Verifying refutations with Extended Resolution. In *CADE*, volume 7898 of *LNAI*, pages 345–359. Springer, 2013.
24. Marijn J. H. Heule, Warren A. Hunt, Jr, and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE-25*, volume 9195 of *LNCS*, pages 591–606. Springer, 2015.
25. Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 50–65, 2011.
26. Marijn J. H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Biere et al. [6], chapter 5, pages 155–184.
27. Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *LNCS*, pages 129–144. Springer, 2010.
28. Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.
29. Richard M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
30. Michal Kouril. Computing the van der Waerden number $W(3,4) = 293$. *INTEGERS: Electronic Journal of Combinatorial Number Theory*, 12(A46):1–13, 2012.
31. Michal Kouril and Jerome L. Paul. The van der Waerden number $W(2,6)$ is 1132. *Experimental Mathematics*, 17(1):53–61, 2008.
32. Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, October 1999.
33. Oliver Kullmann. Fundaments of branching heuristics. In Biere et al. [6], chapter 7, pages 205–244.
34. Leonid Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9:115–116, 1973.
35. Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.
36. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371. Morgan Kaufmann Publishers, 1997.
37. Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference 2012*, 2012.
38. Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [6], chapter 4, pages 131–153.

39. Sid Mijnders, Boris de Wilde, and Marijn J. H. Heule. Symbiosis of search and heuristics for random 3-SAT. In David Mitchell and Eugenia Ternovska, editors, *Third International Workshop on Logic and Search (LaSh 2010)*, 2010.
40. Richard Rado. Some partition theorems. In *Colloquia mathematica Societatis János Bolyai 4, Combinatorial theory and its applications III*, pages 929–936. North-Holland, Amsterdam, 1970.
41. Issai Schur. Über die Kongruenz $x^m + y^m = z^m \pmod p$. *Jahresbericht der Deutschen Mathematikervereinigung*, 25:114–117, 1917.
42. Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.
43. Bartel L. van der Waerden. Beweis einer Baudetschen Vermutung. *Nieuw Archief voor Wiskunde*, 15:212–216, 1927.
44. Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.
45. Vladimir Voevodski. Lecture at ASC 2008, How I became interested in foundations of mathematics, accessed: November 2015.
46. Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.
47. Hantao Zhang. Combinatorial designs by SAT solvers. In Biere et al. [6], chapter 17, pages 533–568.
48. Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.