The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Schur Number Five

Marijn J. H. Heule
Computer Science Department
The University of Texas at Austin
2317 Speedway, M/S D9500
Austin, Texas 78712-0233

Abstract

We present the solution of a century-old problem known as *Schur Number Five*: What is the largest (natural) number n such that there exists a five-coloring of the positive numbers up to n without a monochromatic solution of the equation $a + b = c$? We obtained the solution, $n = 160$, by encoding the problem into propositional logic and applying massively parallel satisfiability solving techniques on the resulting formula. We constructed and validated a proof of the solution to increase trust in the correctness of the multi-CPU-year computations. The proof is two petabytes in size and was certified using a formally verified proof checker, demonstrating that any result by satisfiability solvers—no matter how large—can now be validated using highly trustworthy systems.

Introduction

In the beginning of the 20^{th} century, Issai Schur studied whether every coloring of the positive (natural) numbers with finitely many colors results in monochromatic solutions of the equation $a + b = c$. This work gave rise to the concept of so-called *Schur numbers*: Schur number k, denoted by $S(k)$, is defined as the largest number n for which there exists a k-coloring of the positive numbers up to n with no monochromatic solution of $a + b = c$.[1] For example, $S(2) = 4$: Assume we use the two colors red and blue. If we color 1 with red, we have to color 2 with blue due to $1 + 1 = 2$. This forces us to color 4 with red because of $2 + 2 = 4$. After this, 3 must become blue due to $1 + 3 = 4$. But then, no matter if we color 5 with red or blue, we end up with a monochromatic solution of $1 + 4 = 5$ or $2 + 3 = 5$.

Although Schur's Theorem states that $S(k)$ is finite for any finite value of k (Schur 1917), determining the exact values of $S(k)$ is an open problem in elementary number theory (Guy 1994). In fact, only the values $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, and $S(4) = 44$ have been known so far (Golomb and Baumert 1965). We came up with a highly optimized automated-reasoning method for showing that $S(5) = 160$.

To obtain this solution, we first encoded the Schur Number Five problem into propositional logic and then applied satisfiability (SAT) solving techniques to solve the resulting formula. This approach has been successful in recent years, leading to the solution of hard open problems such as the problem of determining the sixth van der Waerden number (Kouril and Paul 2008), the Erdős discrepancy problem (Konev and Lisitsa 2015), and the Pythagorean triples problem (Heule, Kullmann, and Marek 2016). Trying to solve a SAT encoding for Schur Number Five with off-the-shelf SAT solving tools turned out to be a hopeless endeavor. We therefore came up with a dedicated approach, which is intended to be applicable to related problems as well. We modified existing tools to efficiently solve our encoding. Still, even with our optimized approach, the total computational effort to solve the problem was over 14 CPU years.

If it takes a computer several CPU years to solve a problem, it is only natural to question the correctness of the supposed solution. To deal with this issue, we automatically constructed a proof of the propositional formula that encodes the main statement. The size of this proof is more than two petabytes, making it about ten times larger than "the largest math proof ever" (Lamb 2016). Despite its tremendous size, we were able to verify the correctness of the proof with a formally verified proof checker. Due to recent progress in proof validation (Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2017), checking the correctness of such proofs is now nearly as efficient as the actual construction of the proofs by a SAT solver (Cruz-Filipe et al. 2017; Lammich 2017). In our case, the time spent on proof checking was a little more than 36 CPU years.

The main contributions of this paper are as follows:

- We constructed a propositional formula that is satisfiable if and only if $S(5) \geq 161$. Our proof of unsatisfiability for this formula is over two petabytes in size.
- We certified the proof using a program formally verified by ACL2 (Kaufmann and Moore 1997), thereby providing high confidence in the correctness of our result.
- We enumerated all 2 447 113 088 five-colorings of the numbers 1 to 160 without a monochromatic $a + b = c$.
- We designed a decision heuristic that allows solving Schur number problems efficiently and enables linear-time speedups even when using thousands of CPUs.
- We developed an efficient hardness predictor for partitioning a hard problem into millions of easy subproblems.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] An alternative definition used in the literature picks the smallest n s.t. all k-colorings of 1 to n result in a monochromatic solution. The values of $S(k)$ differ by one, depending on the definition.

Schur Numbers and Variants

Schur number k, denoted by $S(k)$, is defined as the largest (natural) number n such that there exists a k-coloring of the numbers 1 to n without a monochromatic solution of the equation $a + b = c$ with $1 \leq a, b, c \leq n$. The first Schur numbers $S(1) = 1$, $S(2) = 4$, and $S(3) = 13$ can be determined manually while $S(4) = 44$ was computed decades ago (Golomb and Baumert 1965). The best known lower bounds for higher Schur numbers are: $S(5) \geq 160$ (Exoo 1994), $S(6) \geq 536$, and $S(7) \geq 1680$ (Fredricksen and Sweet 2000). We prove that $S(5) = 160$.

The early upper bounds $S(k) \leq \lfloor k!e \rfloor$ (Schur 1917) have later been improved to $S(k) \leq \lfloor k!(e - \frac{1}{24}) \rfloor$ (Irving 1973). Upper bounds on $S(k)$ can also be obtained via the connection to the Ramsey numbers $R_k(3)$, which denote the smallest n such that any k-coloring of the edges of the fully connected graph on n vertices yields a monochromatic triangle: $S(k) \leq R_k(3) - 2$ (Schur 1917). The first three numbers $R_k(3)$ are known: $R_1(3) = 3$, $R_2(3) = 6$, and $R_3(3) = 17$ (Greenwood and Gleason 1955).

Several variants of Schur numbers have been proposed. The oldest variant, known as *weak* Schur number k and denoted by $WS(k)$, requires a to be smaller than b, thus weakening $1 \leq a, b, c \leq n$ to $1 \leq a < b < c \leq n$ (Irving 1973). Hence, $WS(k) \geq S(k)$. Only the four smallest weak Schur numbers are known: $WS(1) = 2, WS(2) = 8, WS(3) = 23$, and $WS(4) = 66$ (Blanchard, Harary, and Reis 2006).

Another variant is the *modular* Schur number k, denoted by $S_{\mathrm{mod}}(k)$, asking for the largest n such that a k-coloring of the numbers 1 to n exists without a monochromatic solution of the equation $a + b \equiv c \pmod{n+1}$ with $1 \leq a, b, c \leq n$ (Abbott and Wang 1977). This variant is stronger than the classical notion, hence $S(k) \geq S_{\mathrm{mod}}(k)$. However, for all known Schur numbers it holds that $S(k) = S_{\mathrm{mod}}(k)$ and this equality is conjectured to hold in general (Abbott and Wang 1977). Our result implies the equality for $k = 5$.

An even stronger variant is the *palindromic* Schur number k, denoted by $S_{\mathrm{pd}}(k)$, for which the numbers i and $n+1-i$ with $1 \leq i \leq n/2$ have the same color—except in case $2i = n+1-i$. This variant is also known as *symmetric sum-free sets* (Fredricksen and Sweet 2000) and is mainly used to determine lower bounds for the weaker variants. We have $S_{\mathrm{mod}}(k) \geq S_{\mathrm{pd}}(k)$ in general. However, the numbers are equal for the known values. This is a new result for $k = 5$.

The big question is whether $S(k) = S_{\mathrm{mod}}(k) = S_{\mathrm{pd}}(k)$ for any k. We can probably answer this question only if the answer is no. Already showing this equality for $k = 6$ is expected to be extremely challenging.

Technical Background

Below we present the most important background concepts related to the more technical part of this paper.

Propositional logic. We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable v (a *positive literal*) or the negation $\overline{v}$ of a variable v (a *negative literal*). The *complementary literal* $\overline{l}$ of a literal l is defined as $\overline{l} = \overline{v}$ if $l = v$ and $\overline{l} = v$ if $l = \overline{v}$. A *clause* is a disjunction of literals. A *formula* is a conjunction of clauses. For a literal, clause, or formula F, $var(F)$ denotes the variables in F. For convenience, we treat $var(F)$ as a variable if F is a literal, and as a set of variables otherwise.

Satisfiability. An *assignment* is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). An assignment is *total* w.r.t. a formula if it assigns a truth value to all variables occurring in the formula; otherwise it is *partial*. A literal l is *satisfied* (*falsified*) by an assignment α if l is positive and $\alpha(var(l)) = 1$ ($\alpha(var(l)) = 0$, resp.) or if it is negative and $\alpha(var(l)) = 0$ ($\alpha(var(l)) = 1$, resp.). We also denote with α the conjuction of literals that are satisfied by that assignment; such a conjunction is called a *cube*. A clause is satisfied by an assignment α if it contains a literal that is satisfied by α. Finally, a formula is satisfied by an assignment α if all its clauses are satisfied by α. A formula is *satisfiable* if there exists an assignment that satisfies it; otherwise it is *unsatisfiable*. A formula F *entails* a formula G, denoted by $F \models G$, if every assignment that satisfies F also satisfies G. F *weakly entails* G, denoted by $F \models_{\mathrm{w}} G$, if satisfiability of F implies satisfiability of G.

Proofs of Unsatisfiability. It is easy to check that an alleged satisfying assignment is valid. However, a certificate that a formula has no solution (i.e., is unsatisfiable) can be huge and costly to validate. We produce proofs of unsatisfiability in the DRAT proof system (Järvisalo, Heule, and Biere 2012), which is the standard in state-of-the-art SAT solving. Given a formula F, a DRAT proof of unsatisfiability is a sequence $C_1, \ldots, C_m$ of clauses where C_m is the empty clause $\bot$. For every clause C_i, it must hold that C_i is a *resolution asymmetric tautology* (RAT) with respect to $F \cup \{C_1, \ldots, C_{i-1}\}$. The addition of a RAT to a formula preserves satisfiability and since the empty clause is trivially unsatisfiable, a DRAT proof witnesses the unsatisfiability of the original formula F. DRAT also allows the deletion of clauses from a formula to improve the performance of proof validation. Note that clause deletion preserves satisfiability.

Encoding

To solve Schur Number Five, we first encode the existence of *certificates* as propositional formulas and then exploit the strength of a parallel SAT solver to efficiently determine whether these formulas are satisfiable. A certificate $S(k, n)$ is a k-coloring of the numbers 1 to n with no monochromatic solution of $a + b = c$ for $1 \leq a, b, c \leq n$. A certificate $S(k, n)$ provides a lower bound for the corresponding Schur problem: $S(k) \geq n$. The size of a certificate $S(k, n)$ is n. An *extreme certificate* is a certificate of maximum size. Figure 1 shows some extreme certificates for the known Schur numbers as well as a palindromic certificate $S(5, 160)$ — which is also an extreme certificate following the presented upper bound result. There is one extreme certificate modulo symmetry (i.e., modulo permuting the colors) with $k \in \{1, 2\}$. These certificates are palindromes and thus modular. There exist three extreme certificates $S(3, 13)$ modulo symmetry.

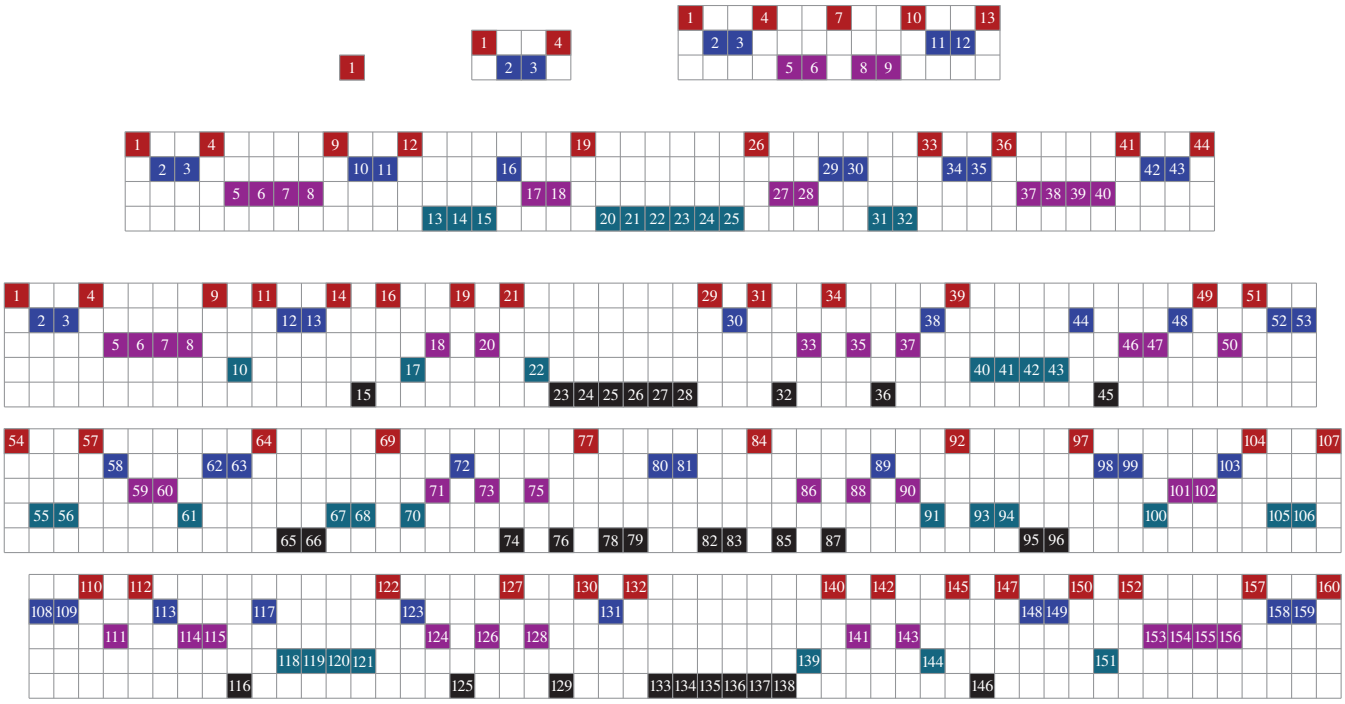

Figure 1: Some extreme (and palindromic) certificates of the known Schur numbers and a palindromic certificate $S(5, 160)$.

All of them are modular and palindromes. They differ only regarding the color of number 7, which can have any color. There are 273 extreme certificates $S(4, 44)$ modulo symmetry, of which 24 are modular and palindromes (Fredricksen and Sweet 2000).

To establish that $S(k) = n$, we need to show that there exists a certificate $S(k, n)$ but no certificate $S(k, n + 1)$. We thus define a family of propositional formulas F_n^k, each of which encodes the existence of a certificate $S(k, n)$. A satisfying assignment of F_{160}^5 can be computed in less than a minute by enforcing that the initial numbers cannot have the last color (Fredricksen and Sweet 2000). The main challenge addressed in this paper is proving unsatisfiability of F_{161}^5 to show that $S(5) < 161$. This requires many CPU years of computation even with optimized heuristics.

For the formula F_n^k, we use Boolean variables v_j^i with $1 \leq i \leq k$ and $1 \leq j \leq n$. Intuitively, a variable v_j^i is true if and only if number j has color i in the certificate. The formula has three kinds of clauses: *positive*, *negative*, and *optional*. The positive clauses encode that every number j must have at least one color. They are of the form $(v_j^1 \vee \cdots \vee v_j^k)$ for $1 \leq j \leq n$. The negative clauses encode that for every solution of the equation $a+b = c$, the numbers a, b, and c cannot have the same color i. They are of the form $(\overline{v}_a^i \vee \overline{v}_b^i \vee \overline{v}_c^i)$ with $a + b = c$ and $1 \leq a, b, c \leq n$. Finally, the optional clauses encode that every number has at most one color. They are of the form $(\overline{v}_j^h \vee \overline{v}_j^i)$ for $1 \leq h < i \leq k$. A commonly used SAT preprocessing technique, called blocked clause elimination (Järvisalo, Biere, and Heule 2012), would remove the optional clauses. However, the optional clauses are required when counting or enumerating certificates.

Example 1. Formula F_4^2 consists of the following clauses:

$$(v_1^1 \vee v_1^2) \wedge (v_2^1 \vee v_2^2) \wedge (v_3^1 \vee v_3^2) \wedge (v_4^1 \vee v_4^2) \wedge$$
$$(\overline{v}_1^1 \vee \overline{v}_2^1) \wedge (\overline{v}_1^1 \vee \overline{v}_2^1 \vee \overline{v}_3^1) \wedge (\overline{v}_1^1 \vee \overline{v}_3^1 \vee \overline{v}_4^1) \wedge (\overline{v}_2^1 \vee \overline{v}_4^1) \wedge$$
$$(\overline{v}_1^2 \vee \overline{v}_2^2) \wedge (\overline{v}_1^2 \vee \overline{v}_2^2 \vee \overline{v}_3^2) \wedge (\overline{v}_1^2 \vee \overline{v}_3^2 \vee \overline{v}_4^2) \wedge (\overline{v}_2^2 \vee \overline{v}_4^2) \wedge$$
$$(\overline{v}_1^1 \vee \overline{v}_1^2) \wedge (\overline{v}_2^1 \vee \overline{v}_2^2) \wedge (\overline{v}_3^1 \vee \overline{v}_3^2) \wedge (\overline{v}_4^1 \vee \overline{v}_4^2)$$

The first line shows the positive clauses, the second and third line the negative clauses, and the last line the optional clauses. Notice that $(\overline{v}_1^1 \vee \overline{v}_2^1 \vee \overline{v}_3^1)$ and $(\overline{v}_1^2 \vee \overline{v}_2^2 \vee \overline{v}_3^2)$ are subsumed by $(\overline{v}_1^1 \vee \overline{v}_2^1)$ and $(\overline{v}_1^2 \vee \overline{v}_2^2)$, respectively.

Symmetry Breaking

A certificate symmetry σ for a Schur number problem is a mapping from any certificate onto another certificate of that problem. Schur number problems have the certificate symmetry σ_{col} that permutes the colors. Due to σ_{col}, SAT solvers would explore all $5! = 120$ color permutations when solving formulas F_n^5. In the following, we describe how to fully and compactly break this symmetry by enforcing a lexicographical ordering on the colors (Crawford et al. 1996).

Breaking the certificate symmetry σ_{col} for the first two colors is easy: We just assign the first color to number 1 and the second color to number 2. Adding the unit clauses (v_1^1) and (v_2^2) to the formula will enforce this. Note that the two numbers must be colored differently because of the equation $1 + 1 = 2$.

Breaking σ_{col} for the third color is more involved. At least one of the numbers 3, 4, and 5 can have neither the first nor the second color due to $S(2) = 4$. We break the symmetry of the third color as follows: If number 4 has neither the

first nor the second color, we color it with the third color. Otherwise, if number 3 has neither the first nor the second color, we color number 3 with the third color. Otherwise, we color number 5 with the third color. We picked number 4 as starting point as it is more constrained due to the equation $2 + 2 = 4$ and the clause (v_2^2). It therefore allows a more compact symmetry-breaking predicate, which consists of the clauses $(\overline{v}_3^5)$, $(\overline{v}_4^4)$, $(\overline{v}_4^5)$, $(v_4^3 \vee \overline{v}_3^4)$, $(v_3^4 \vee \overline{v}_5^5)$, and $(v_3^3 \vee v_4^3 \vee \overline{v}_5^4)$. Finally, to distinguish between the fourth and the fifth color, we assign the fourth color to the first number that does not have the first, second, or third color. We encode this with clauses of the form $(v_1^4 \vee \cdots \vee v_i^4 \vee \overline{v}_{i+1}^5)$. We require these clauses only for $i \leq S(3) = 13$.

Generating the original formulas F_n^k can be easily achieved with a dozen lines of code. In contrast, the addition of compact symmetry-breaking predicates is more complicated and may therefore result in errors. Let R_n^k be the formula obtained from F_n^k by adding symmetry-breaking predicates. To ensure correctness of the symmetry breaking, we constructed a proof, called the *re-encoding proof*, that the satisfiability of F_{161}^5 implies the satisfiability of R_{161}^5.

Decision Heuristics

We used the *cube-and-conquer* method (Heule et al. 2012) for SAT solving as it is arguably the most effective method for solving very hard combinatorial problems. This method was also used for solving the Erdős discrepancy problem (Konev and Lisitsa 2015) and the Pythagorean triples problem (Heule, Kullmann, and Marek 2016).

Cube-and-conquer is a hybrid parallel SAT solving paradigm that combines *look-ahead* techniques (Heule and van Maaren 2009) with *conflict-driven clause learning* (CDCL) (Marques-Silva, Lynce, and Malik 2009): Look-ahead techniques are used for splitting a given problem into many (millions or even billions of) subproblems which are then solved with CDCL solvers. Since the subproblems are independent, they can be easily solved in parallel without requiring communication.

The aim of look-ahead techniques is to find variable assignments that simplify a formula as much as possible. This is achieved with so-called *look-aheads*: A look-ahead on a literal l with respect to a formula F first assigns l to true and then simplifies F to obtain a formula F'. After this, it determines a heuristic value by computing the "difference" between F and F' (details are given below). A variable v is considered useful for splitting a formula F if the look-aheads on both v and $\overline{v}$ have a high heuristic value. Typically, look-ahead techniques select the variable v for which the product of the heuristic values of v and $\overline{v}$ is the largest.

The effectiveness of look-ahead heuristics depends on measuring the difference between the formula F and the simplified formula F'. A reasonably effective measure, which is also easy to compute, is the difference in the number of variables: $|var(F)|-|var(F')|$. This measure is used in the cube-and-conquer solver TREENGELING (Biere 2013), which solved most benchmarks of the SAT Competition 2016 (Balyo, Heule, and Järvisalo 2017). An alternative, more costly measure, considers the clauses that have been reduced, but not satisfied, during the simplification, i.e., the clauses in $F' \setminus F$. These clauses are typically assigned a weight, with shorter clauses getting a larger weight. During our initial experiments for solving Schur Number Five, we observed that the clause-based heuristics is much more effective than the variable-based one. However, our initial experiments—based on splitting the problem into millions of subproblems and solving randomly selected subproblems—indicated that finding the solution of Schur Number Five would require many decades of CPU time.

The key to reducing the computational effort of solving Schur Number Five is a new measurement method. We first discuss the main weakness of the weighted-sum heuristics before we describe our new method. Recall that the Schur number encoding uses $\mathcal{O}(n)$ positive clauses of length k and $\mathcal{O}(kn^2)$ negative clauses of length 3. Thus, no matter on what literal we look ahead, most clauses in $F' \setminus F$ originate from negative clauses. Moreover, a clause in $F' \setminus F$ that originates from a negative clause has length 2, while a clause in $F' \setminus F$ that originates from a positive clause can be larger. Commonly used heuristics favor shorter clauses and thus favor clauses that originate from negative clauses. Because of this, the heuristic value of look-aheads is dominated by reduced negative clauses. However, it appears that favoring reduced positive clauses is more effective.

Example 2. Recall F_4^2, but now without redundant clauses:

$$(v_1^1 \vee v_1^2) \wedge (v_2^1 \vee v_2^2) \wedge (v_3^1 \vee v_3^2) \wedge (v_4^1 \vee v_4^2) \wedge$$
$$(\overline{v}_1^1 \vee \overline{v}_2^1) \wedge (\overline{v}_1^1 \vee \overline{v}_3^1 \vee \overline{v}_4^1) \wedge (\overline{v}_2^1 \vee \overline{v}_4^1) \wedge$$
$$(\overline{v}_1^2 \vee \overline{v}_2^2) \wedge (\overline{v}_1^2 \vee \overline{v}_3^2 \vee \overline{v}_4^2) \wedge (\overline{v}_2^2 \vee \overline{v}_4^2)$$

Let us look ahead on literal $\overline{v}_3^1$: Assigning variable v_3^1 to false satisfies $(\overline{v}_1^1 \vee \overline{v}_3^1 \vee \overline{v}_4^1)$ and reduces $(v_3^1 \vee v_3^2)$ to (v_3^2), thereby forcing the variable v_3^2 to true. This in turn reduces the negative clause $(\overline{v}_1^2 \vee \overline{v}_3^2 \vee \overline{v}_4^2)$ to $(\overline{v}_1^2 \vee \overline{v}_4^2)$. The only clause that is reduced, but not satisfied, is $(\overline{v}_1^2 \vee \overline{v}_4^2)$. Hence, looking ahead on $\overline{v}_3^1$ yields $F' \setminus F = (\overline{v}_1^2 \vee \overline{v}_4^2)$.

We now present our generalization of an effective heuristic for uniform random 3-SAT instances (Li 1999) to arbitrary CNFs. Given a literal l and a formula F, let $occ(F, l)$ denote the number of occurrences of l in F. The weight of a clause $C \in F$, denoted by $w(F, C)$, is computed as follows:

$$w(F, C) = \frac{\sum_{l \in C} occ(F, \bar{l})}{2^{|C|} \cdot |C|}$$

The $|C|$ in the denominator reduces the sum to the average and $2^{|C|}$ ensures a larger weight for shorter clauses. We noticed that the sum works much better than the product for arbitrary CNFs, in contrast to random 3-SAT formulas (Dubois and Dequen 2001). The heuristic value of a variable v w.r.t. a formula F, denoted by $H(F, v)$, is computed as follows (with F' and F'' referring to the formulas obtained by look-aheads on F with the literals v and $\overline{v}$, respectively):

$$H(F, v) = \Big(\sum_{C \in F' \setminus F} w(F, C) \Big) \cdot \Big(\sum_{C \in F'' \setminus F} w(F, C) \Big)$$

In each node of the search tree, the variable with the highest heuristic value is selected as splitting variable.

Partitioning

A crucial part of solving Schur Number Five is the partitioning of the propositional formulas R^5_{160} and R^5_{161} into millions of easy subproblems. We use the former formula to compute all extreme certificates $S(5, 160)$ and the latter formula for the upper bound result. We constructed a single partition for both formulas. A partition is a set of cubes (or equivalently, a set of variable assignments). The disjunction of cubes in a partition must be a tautology in order to ensure that the cubes cover the entire search space. By applying a cube to a formula, one obtains a subproblem of that formula. Each of the subproblems arising from a partition can be solved in parallel, thereby allowing massively parallel computation. Moreover, these subproblems are partitioned again to solve them more efficiently (on a single core).

The top-level partition is constructed as follows: We use the look-ahead decision heuristic described above to build a binary search tree over the space of possible variable assignments. In this tree, every non-leaf node is assigned a splitting variable. The left outgoing edge of a node assigns its variable to true while the right one assigns it to false. Each node in the tree represents the variable assignment corresponding to all assignments on the path from the root node. In case the formula in a node (i.e., the formula obtained from the original formula by applying the assignment represented by that node) becomes "easy", we stop splitting. The partition consists of all assignments that are represented by the leaf nodes of the tree. We require a measure that captures the hardness of a formula in each node. A rough measure suffices here, since we will fine-tune this partition later. We observed that the number of binary clauses in a formula is a reasonable measure for the hardness of Schur number subproblems. The more binary clauses, the more constrained the subproblem (and thus easier to solve). For example, stopping with the splitting as soon as a formula in a node has more than 3700 binary clauses results in about 9 millions of mostly easy subproblems of R^5_{160} and R^5_{161}.

The Hidden Strength of Cube-and-Conquer

Cube-and-conquer is not only useful for partitioning a hard problem into many subproblems that can be solved in parallel, but also to boost performance of solving a problem on a single core. Let N be the number of cubes in a partition. A low value of N indicates that the problem is split into a low number of subproblems, meaning that it is mainly solved with CDCL ($N = 1$ means pure CDCL) while a larger value indicates a more extensive splitting based on look-aheads.

If we experiment with different values for N when trying to solve a problem on a single core, we can observe an interesting pattern: For low values of N, an increase of N leads to an increase of the total runtime—apparently some subproblems are about as hard as the original one. If we increase N further, the total runtime starts to decrease and at some point it can even become significantly smaller compared to solving the problem with CDCL alone (again running both on a single core). Yet when N becomes really large, the runtime increases again. At this point, splitting starts to dominate the total costs. Figure 2 shows this pattern on a subproblem of R^5_{161}, where the optimal value for N is around 10 000.

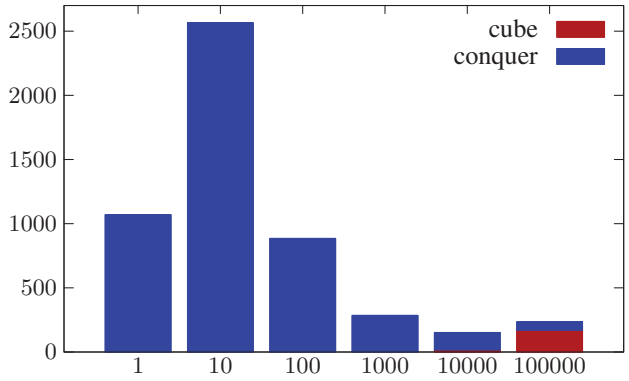

Figure 2: Comparison of the total runtime in seconds (y-axis) for solving a subproblem of R^5_{161} using different numbers of cubes (x-axis) on a single core (no parallelism).

We developed a mechanism that approximates the optimal number to realize fast performance. The mechanism stops splitting if the number of remaining variables in a node drops below the value of parameter δ. The number of remaining variables is a useful measure as it strictly decreases, whereas number of binary clauses can oscillate. Initial experiments showed that such oscillation can slowdown the solver on easy problems. We initialize δ to 0, meaning that we keep splitting until δ is increased. We only increase δ when look-ahead techniques can refute a node, which naturally terminates splitting. In this case, δ is set to the number of remaining variables in that node. The increase is motivated as follows: If look-ahead techniques can refute a node, then we expect CDCL to refute that node—as well as similar nodes—more efficiently.

The value of δ is decreased in each node of the search tree to ensure that look-ahead techniques refute nodes once in a while. We experimented with various methods to implement the decrement and observed that the size of the decrease should be related to the depth of a node in the search tree. The closer a node is to the root, the larger the decrement. More specifically, we used the following update function (with d referring to the depth of the node that performs the update and parameter e referring to *down exponent* and parameter f referring to the *down factor*):

$$\delta := \delta\big(1 - f^{d^e}\big)$$

If the value of f is close to 0, then δ climbs to a value at which look-ahead techniques will rarely refute a node. On the other hand, if the value of f is close to 1, then δ drops quickly to 0, so that practically all leaf nodes will be refuted by look-ahead techniques. The value of e determines the influence of the depth. If e is close to 0, then the depth is ignored during the update, while if e is close to 1 (or even larger), then the depth is dominant. During our experiments, various combinations of values of e and f resulted in strong performance. Examples of effective values are $e = 1.0$ and $f = 0.6$, $e = 0.5$ and $f = 0.1$, and $e = 0.3$ and $f = 0.02$. For the final experiments we used $e = 0.3$ and $f = 0.02$.

Hardness Predictor and Partition Balancing

Most modern microprocessors used in clusters, including the Intel Xeon chips we used, have many CPU cores yet relatively little memory—at least for our application, which runs many SAT solvers and theorem provers in parallel. A major challenge was technical in nature: maximizing the CPU usage (with hyper-threading) without running out of memory.

Although most subproblems generated in the top-level partition could be solved within reasonable time (less than two minutes) some subproblems required hours of computation. Moreover, solving these hard subproblems required disproportionally more memory. Solving a few hard problems on the same chip at the same time could kill all threads. We therefore fine-tuned the initial partition.

We used the mechanism to partition subproblems as a hardness predictor of subproblems by using a high down exponent and a small down factor: $e = 1.0$ and $f = 0.1$. Using a small down factor boosts the partitioning runtime, but produces typically too few cubes. However, here we only care about the runtime. It turned out that the runtime of partitioning with $e = 1.0$ and $f = 0.1$ is larger than a second for hard subproblems and significantly smaller for the easier ones. We extended the partition by splitting subproblems if this hardness predictor took over a second. Splitting was continued until none of the subproblems was predicted to be hard. To limit the size of the partition, we merged two cubes if they had the same parent node and the sum of their hardness predictor times was less than 0.1 second.

Solving Subproblems

Our top-level partition of R^5_{160} and R^5_{161}, denoted by P^5, consists of $10\,330\,615$ cubes after partition balancing. Figure 3 (left) shows a histogram of the size of the cubes in P^5. The smallest cube has size 13 while the largest cube has length 62, showing that the binary tree associated with the cubes is quite unbalanced. Notice that the size of most cubes is in the range from 20 to 40, which is a large interval.

Figure 3 (middle) shows a histogram of the number of binary clauses in subproblems, i.e., the resulting formulas after applying the cubes. Notice that the interval here is small: Most subproblems have between 3650 and 3850 binary clauses. As stated earlier, the number of binary clauses is a useful rough measure for the hardness of subproblems.

For each cube $\alpha \in P^5$, we solved[2] the problem $R^5_{160} \wedge \alpha$ using our cube-and-conquer solver consisting of a modified version of MARCH_CU (Heule et al. 2012) as look-ahead (cube) solver and GLUCOSE 3.0 (Audemard and Simon 2009) as CDCL (conquer) solver. The cube solver modifications consist of integrating the presented decision heuristic and replacing the cutoff procedure by the presented down factor mechanism. In case $R^5_{160} \wedge \alpha$ was unsatisfiable, we stored the proof of unsatisfiability, which is also a proof of unsatisfiability of $R^5_{161} \wedge \alpha$. There were only 961 cubes $\alpha \in P_5$ for which $R^5_{160} \wedge \alpha$ turned out to be satisfiable. For those cubes, we computed the proof of unsatisfiability of $R^5_{161} \wedge \alpha$. These proofs together form the *implication proof*.

[2]The tools and proof parts presented in this paper are available at https://www.cs.utexas.edu/~marijn/Schur/.

We solved the subproblems on the Lonestar 5 cluster of the Texas Advanced Computing Center (TACC). Each compute node consists of a Xeon E5-2690 v3 (Haswell) chip with 24 cores running on 2.6 GHz. Hyper-threading was enabled, resulting in 48 logical CPUs per node. We ran the experiments on 50 nodes in parallel, resulting in running 2400 copies of our cube-and-conquer solver in parallel. The total runtime was roughly $27\,600$ CPU hours for the partition phase and roughly $95\,600$ CPU hours for the conquer phase. The total costs to compute Schur Number Five was just over 14 CPU years, but less than three days in wall-clock time on the Lonestar 5 cluster. Figure 3 (right) shows a histogram of the runtimes (rounding times to the nearest 5 seconds). A large fraction of the subproblems can be solved within 20 to 40 seconds. Most subproblems are solvable within two minutes and few are somewhat harder. The subproblems were partitioned into a total of 65 billion cubes and the number of conflict clauses added in the conquer phase was 11 trillion.

The computation of Schur Number Five and the Pythagorean triples problem differ in the balance between the partition phase and the conquer phase. For Schur Number Five, almost 78% of the computation was devoted to the conquer phase, while for the Pythagorean triples problem this was only 38%. This difference can be explained as follows: For both problems, a heuristic was chosen to continue splitting until the total runtime would start to increase (based on the solving time of randomly selected subproblems). In the case of Schur Number Five, this point is reached earlier. The Pythagorean triples problem was solved on the older Stampede cluster of TACC, which hinders a clean runtime comparison. We estimate that the conquer phase of solving Schur Number Five required about ten times more computation resources than solving the Pythagorean triples problem.

No Backbone, but Backdoors

The *backbone* of a CNF formula is the set of literals that are assigned to true in all satisfying total assignments. Many formulas that encode the existence of extreme certificates of problems in Ramsey Theory (Graham, Rothschild, and Spencer 1990), such as the van der Waerden numbers (Kouril and Paul 2008) and the Pythagorean triples problem (Heule, Kullmann, and Marek 2016), have large backbones after symmetry breaking—even if the number of satisfying total assignments is enormous. However, the backbone of R^5_{160} consists only of the literals that are assigned by the symmetry-breaking predicates.

The lack of a substantial backbone suggests that there may exist a symmetry that is not broken. It turns out that palindromic Schur number problems have certificate symmetry $\sigma_{\cdot p}$ that maps each number i onto $i \cdot p \pmod{n+1}$ with n being the size of the certificate and p any number that is relatively prime to $n+1$ (Fredricksen and Sweet 2000). However, $\sigma_{\cdot p}$ is not a certificate symmetry of classic Schur number problems. The size of the backbone can therefore be explained by the equivalence $S(5) = S_{\mathrm{pd}}(5) = 160$ and the certificate symmetry $\sigma_{\cdot p}$ of palindromic Schur number problems.

Although the backbone of R^5_{160} is small, we observed that there are several backdoors to large clusters of solutions. A

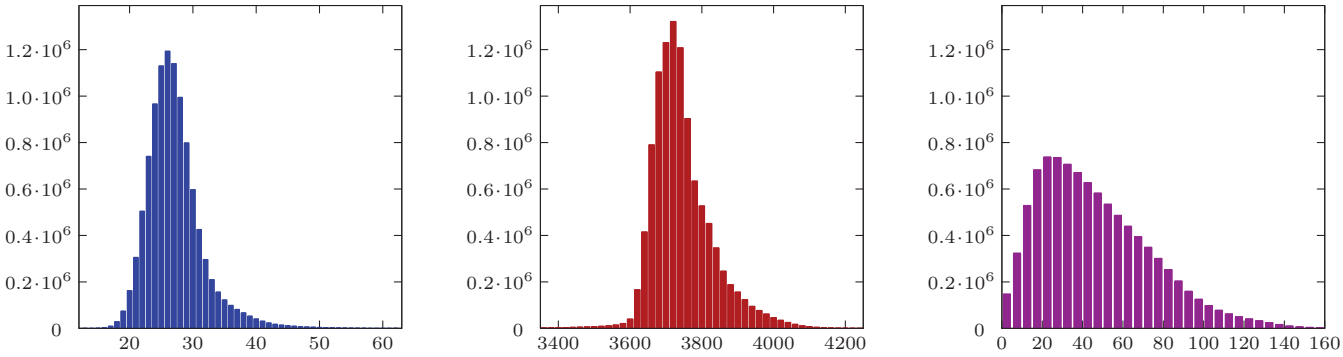

Figure 3: Histograms showing statistics regarding the sizes of cubes (left, ranging from 13 to 62), the number of binary clauses in the formula under that cube (middle), and the time to solve the subformulas (right, in seconds).

backdoor of a CNF formula F is a partial assignment β such that $F \wedge \beta$ can be solved efficiently (in polynomial time) using a given algorithm (Williams, Gomes, and Selman 2003). We selected subsumption elimination (SE) (Eén and Biere 2005) followed by blocked clause elimination (BCE) (Järvisalo, Biere, and Heule 2012) as the basis for backdoors of R^5_{160}. Both SE and BCE are confluent and run in polynomial time. BCE removes blocked clauses until fixpoint. BCE solves a given formula if and only if the fixpoint is the empty formula, which is trivially satisfiable.

We computed the backdoors as follows: We started with the top-level cubes (assignments) under which R^5_{160} is satisfiable. For each such assignment α, we computed the backbone of $R^5_{160} \wedge \alpha$. The backbone can be computed using various SAT calls: Literal l belongs to the backbone if and only if $R^5_{160} \wedge \alpha \wedge (\bar{l})$ is unsatisfiable. In most cases, the backbone assignment turned out to be a backdoor of R^5_{160}. In the remaining cases, we extended the backbone assignments using look-aheads until they became backdoors. This procedure resulted in 1616 backdoors of R^5_{160}, which—by construction—cover all satisfying assignments.

We used these backdoors to compute all 2 447 113 088 extreme certificates $S(5, 160)$ with the SHARPSAT solver (Thurley 2006) in a few seconds. Out of these extreme certificates, 315 853 824 are modular and of the modular ones, 334 752 are palindromes. The latter number has been conjectured before (Fredricksen and Sweet 2000).[3]

Correctness

We produced and certified a proof of unsatisfiability of F^5_{161} to increase confidence in the correctness of our result. The proof consists of three parts: The *re-encoding proof*, the *implication proof*, and the *tautology proof*. The re-encoding proof shows the correctness of our symmetry-breaking technique, i.e., that the satisfiability of F^5_{161} implies the satisfiability of the re-encoded formula R^5_{161}. The implication proof includes for each cube $\alpha \in P^5$, a proof of unsatisfiability of $R^5_{161} \wedge \alpha$. This shows that R^5_{161} implies each clause $C = \neg\alpha$. The tautology proof shows that the disjunction of cubes is a tautology, i.e., the cubes together cover the entire search space. Let $\overline{P^5}$ denote the negation of P^5, i.e., the CNF formula that has each clause $C = \neg\alpha$ with $\alpha \in P^5$. The disjunction of cubes is a tautology if and only if $\overline{P^5}$ is unsatisfiable. We show unsatisfiability of F^5_{161} via

$$\overbrace{F^5_{161} \models_{\mathrm{w}} R^5_{161}}^{\text{re-encoding proof}} \underbrace{\models \overline{P^5}}_{\text{implication proof}} \overbrace{\models \bot}^{\text{tautology proof}}$$

The proof parts have been constructed in the DRAT format, which facilitates expressing techniques that remove satisfying assignments, such as symmetry breaking (Heule, Hunt Jr., and Wetzler 2015). Recent progress in verified proof checking (Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2017) reduced proof validation costs such that they became comparable to the costs of solving. We converted the DRAT proofs into LRAT proofs, a new format that was recently introduced to allow efficient proof checking using a theorem prover (Cruz-Filipe et al. 2017). We used a verified LRAT proof-checker (Heule et al. 2017), written in the language of the ACL2 theorem proving system (Kaufmann and Moore 1997), and applied it to certify these LRAT proofs.

Only the encoding of the Schur Number Five problem into propositional logic (i.e., the generation of F^5_{161}) was not checked using a theorem prover. We chose to skip verification of this part because the encoding can be implemented using a dozen lines of straightforward C code.

Re-encoding Proof. The purpose of the re-encoding proof is to express our symmetry-breaking techniques—used for breaking the color symmetry σ_{col}—in the DRAT proof system. We did this using an existing method (Heule, Hunt Jr., and Wetzler 2015): For each clause in a symmetry-breaking predicate, the method adds a new definition stating that if the lexicographical ordering is violated, two colors are swapped. However, the unit clauses (v^1_1) and (v^2_2) cannot be added immediately as enforcing the lexicographical order for the first two colors requires multiple swaps. Instead of the unit clause

[3]The paper (Fredricksen and Sweet 2000) states that the number of palindromic extreme certificates $S(5, 160)$ is 309 408. However, the described method produces 334 752 such certificates.

(v_1^1), we learn $(\overline{v}_1^2)$, $(\overline{v}_1^3)$, $(\overline{v}_1^4)$, and $(\overline{v}_1^5)$ which together imply (v_1^1). The case of (v_2^2) is similar.

For example, the first unit clause in the re-encoding proof is $(\overline{v}_1^5)$, which is learned as follows. Any assignment that assigns v_1^5 to true, violates the lexicographical ordering. In particular it violates $v_1^4 \geq v_1^5$ as v_1^5 to true forces v_1^4 to false via the optional clause $(\overline{v}_1^4 \vee \overline{v}_1^5)$. We add constraints to the formula stating that if v_1^5 is assigned to true, then every variable v_i^4 is swapped with v_i^5 and the other way around. Expressing this swap using DRAT steps requires introducing auxiliary variables. Afterwards the unit clause $(\overline{v}_1^5)$ is implied.

The re-encoding proof is 35 megabytes in size (uncompressed DRAT) and consists of almost a million clause addition steps and a similar number of clause deletion steps. That is reasonably large considering that it only breaks the color symmetry σ_{col}. However, compared to the implication proof (discussed below) the size is negligible.

Implication Proof. We proved that R_{161}^5 is unsatisfiable by showing that there exists a formula, in our case $\overline{P^5}$, such that (1) every clause in the formula is logically implied by R_{161}^5, and (2) the formula can be easily shown to be unsatisfiable. The implication proof includes, for each cube $\alpha \in P^5$, a proof of unsatisfiability of $R_{161}^5 \wedge \alpha$. The size of the implication proof is 0.88 petabytes in the compressed DRAT format produced by GLUCOSE and 2.18 petabytes in the compressed LRAT format produced by the DRAT-TRIM proof checker. The latter format is used by the formally verified checker. As a comparison, the proof of the Pythagorean triples problem is 200 terabytes in the uncompressed DRAT format (Heule, Kullmann, and Marek 2016). Lightweight proof compression shrinks DRAT proofs of Schur number problems to approximately 45% of their size, while LRAT proofs are reduced to about 30% of their size. DRAT proofs of Schur number problems have lots of small numbers, while LRAT proofs have large numbers. This causes the different effectiveness in proof compression. Based on the DRAT compression rate, the Schur Number Five proof is about ten times as large in the same format. Producing the compressed LRAT proof required almost 20.5 CPU years while certifying it required another 15.6 CPU years.

Tautology Proof. The tautology proof describes that the disjunction of cubes is a tautology, i.e., that the cubes cover the entire search space. We showed this by proving that $\overline{P^5}$ is unsatisfiable. The cubes produced by our partition method form a binary tree of assignments by construction. The tautology proof consists of $|\overline{P^5}|-1$ resolution steps, each time resolving two clauses whose corresponding cubes have the same parent node in the binary tree. The size of formula $\overline{P^5}$ is 1 gigabyte and the size of the tautology proof is 3 gigabytes in the uncompressed DRAT format.

Certifying the Proof

The size of the proof demands a parallel certification approach and storing intermediate results. Below we describe our method, which uses widely used tools.

- $F_{161}^5 \models_{\text{w}} R_{161}^5$: We provided the ACL2 theorem prover with the formulas F_{161}^5, R_{161}^5, and the re-encoding proof. After validating this proof, it returns the parsed formulas $F_{161}'^5$ and $R_{161}'^5$ and a verified statement that $F_{161}'^5 \models_{\text{w}} R_{161}'^5$. Correctness of the parsing is checked using the Unix tool `diff` by comparing F_{161}^5 with $F_{161}'^5$ and R_{161}^5 with $R_{161}'^5$.
- $R_{161}^5 \models \overline{P^5}$: We check that every clause $C \in \overline{P^5}$ is implied by R_{161}^5. The theorem prover receives R_{161}^5, C, and a proof of unsatisfiability of $R_{161}^5 \wedge \neg C$. The theorem prover returns the parsed formula $R_{161}'^5$, parsed clause C', and a statement that $R_{161}'^5 \models C'$. Again `diff` is used to check the equivalence of the formulas R_{161}^5 and $R_{161}'^5$. Clause C' is stored for the next step.
- $\overline{P^5} \models \bot$: We construct $\overline{P'^5}$ by concatenating all clauses C' implied by R_{161}^5 in the prior step, simply using the Unix tool `cat`. The theorem prover is provided with $\overline{P'^5}$ and a proof of its unsatisfiability, and proves that the parsed formula $\overline{P''^5}$ is unsatisfiable. The last check, again using `diff`, validates that $\overline{P''^5}$ equals the stored formula $\overline{P'^5}$.

Conclusions and Future Work

We proved that $S(5) = 160$ using massively parallel SAT solving. To achieve this result, we designed powerful look-ahead heuristics and developed a cheap hardness predictor to partition a hard problem into millions of manageable subproblems. These subproblems were solved using our cube-and-conquer solver. The resulting proof is over two petabytes in size in a compressed format. We certified the correctness of the proof using the ACL2 theorem proving system. Given the enormous size of the proof, we argue that any result produced by SAT solvers can now be validated using highly trustworthy systems with reasonable overhead.

A century after Issai Schur proved the existence of Schur numbers, we now know the value of the first five. Determining Schur number six will be extremely challenging and might be beyond any computational method. A more realistic problem is the computation of the fifth *weak* Schur number $WS(5)$. Just a few years ago, it was shown that $WS(5) \geq 196$ (Eliahou et al. 2012), while it has been conjectured since the 1950s that $WS(5) = 196$ (Walker 1952). This appears relatively close to the value of $S(5)$. However, we expect the corresponding propositional formula to be much harder to solve due to the lack of binary negative clauses in the encoding of weak Schur numbers.

Acknowledgements

The author is supported by NSF under grant CCF-1526760 and by AFRL Award FA8750-15-2-0096. The author thanks Benjamin Kiesl, Jasmin Blanchette, Matt Kaufmann, Armin Biere, Victor Marek, Scott Aaronson, and the anonymous reviewers for their valuable input to improve the quality of the paper. The author acknowledges 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

Abbott, H. L., and Wang, E. T. H. 1977. Sum-free sets of integers. *Proceedings of the American Mathematical Society* 67:11–16.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *21st International Joint Conference on Artificial Intelligence*, 399–404.

Balyo, T.; Heule, M. J. H.; and Järvisalo, M. 2017. SAT competition 2016: Recent developments. In *AAAI 2017*, 5061–5063.

Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Biere, A. 2013. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013* 51.

Blanchard, P. F.; Harary, F.; and Reis, R. 2006. Partitions into sum-free sets. *Integers* 6. #A07.

Crawford, J. M.; Ginsberg, M. L.; Luks, E. M.; and Roy, A. 1996. Symmetry-breaking predicates for search problems. In *KR 1996*, 148–159. Morgan Kaufmann.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt Jr., W. A.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient certified RAT verification. In *CADE 26*, 220–236. Springer.

Cruz-Filipe, L.; Marques-Silva, J. P.; and Schneider-Kamp, P. 2017. Efficient certified resolution proof checking. In *TACAS 2017*, 118–135. Springer.

Dubois, O., and Dequen, G. 2001. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *17th International Joint Conference on Artificial Intelligence – IJCAI'01*, 248–253. Morgan Kaufmann.

Eén, N., and Biere, A. 2005. Effective preprocessing in SAT through variable and clause elimination. In *8th International Conference on Theory and Applications of Satisfiability Testing – SAT 2005*, 61–75. Springer.

Eliahou, S.; Marín, J.; Revuelta, M.; and Sanz, M. 2012. Weak Schur numbers and the search for G.W. Walker's lost partitions. *Computers & Mathematics with Applications* 63(1):175 – 182.

Exoo, G. 1994. A lower bound for Schur numbers and multicolor Ramsey numbers of K_3. *The Electronic Journal of Combinatorics* 1. #R8.

Fredricksen, H., and Sweet, M. M. 2000. Symmetric sum-free partitions and lower bounds for Schur numbers. *Electronic Journal of Combinatorics* 7. #R32.

Golomb, S. W., and Baumert, L. D. 1965. Backtrack programming. *Journal of the ACM* 12(4):516–524.

Graham, R. L.; Rothschild, B. L.; and Spencer, J. H. 1990. *Ramsey Theory, 2nd Edition*. Wiley.

Greenwood, R. E., and Gleason, A. M. 1955. Combinatorial relations and chromatic graphs. *Canadian Journal of Mathematics* 7:1–7.

Guy, R. K. 1994. *Unsolved problems in number theory; 2nd ed.* Problem Books in Mathematics Unsolved Problems in Intuitive Mathematics. New York: Springer.

Heule, M. J. H., and van Maaren, H. 2009. *Look-Ahead Based SAT Solvers*, Volume 185 of Biere et al. (2009). chapter 5, 155–184.

Heule, M. J. H.; Kullmann, O.; Wieringa, S.; and Biere, A. 2012. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *7th International Haifa Verification Conference – HVC 2011*, 50–65. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; Kaufmann, M.; and Wetzler, N. D. 2017. Efficient, verified checking of propositional proofs. In *ITP 2017*, 269–284. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. D. 2015. Expressing symmetry breaking in DRAT proofs. In *CADE 25*, 591–606. Springer.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and verifying the Boolean Pythagorean Triples problem via Cube-and-Conquer. In *Theory and Applications of Satisfiability Testing – SAT 2016*, 228–245. Springer.

Irving, R. W. 1973. An extension of Schur's theorem on sum-free partitions. *Acta Arithmetica* 25:55–64.

Järvisalo, M.; Biere, A.; and Heule, M. J. H. 2012. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning* 49(4):583–619.

Järvisalo, M.; Heule, M. J. H.; and Biere, A. 2012. Inprocessing rules. In *IJCAR 2012*, 355–370. Springer.

Kaufmann, M., and Moore, J. S. 1997. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering* 23(4):203–213.

Konev, B., and Lisitsa, A. 2015. Computer-aided proof of Erdős discrepancy properties. *Artificial Intelligence* 224(C):103–118.

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

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

Lammich, P. 2017. Efficient verified (UN)SAT certificate checking. In *CADE 26*, 237–254. Springer.

Li, C. M. 1999. A constraint-based approach to narrow search trees for satisfiability. *Information processing letters* 71(2):75–80.

Marques-Silva, J. P.; Lynce, I.; and Malik, S. 2009. *Conflict-Driven Clause Learning SAT Solvers*, Volume 185 of Biere et al. (2009). chapter 4, 131–153.

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

Thurley, M. 2006. sharpSAT - counting models with advanced component caching and implicit BCP. In *SAT 2006*, volume 4121 of *LNCS*, 424–429. Springer.

Walker, G. 1952. A problem in partitioning. *American Mathematical Monthly* 59:253.

Williams, R.; Gomes, C. P.; and Selman, B. 2003. Backdoors to typical case complexity. In *18th International Joint Conference on Artificial Intelligence*, 1173–1178.