PhD Research Proposal

Curtis Bright

February 29, 2016

Abstract

We propose to combine the research areas of *satisfiability solving* and *symbolic computation* to develop efficient procedures which can perform combinatorial searches. Our proposed algorithm and tool combines efficient general-purpose search procedures (like those in modern SAT solvers) with domain-specific knowledge (like those in computer algebra systems) to verify, couterexample, and study mathematical conjectures (especially in combinatorics) and the structures they refer to.

Contents

1 Introduction

"**Brute**-brute force has no hope. But clever, inspired brute force is the future." – Doron Zeilberger[1]

1.1 Motivation

At the 40th International Symposium on Symbolic and Algebraic Computation [Á15] the invited talk *Building Bridges between Symbolic Computation and Satisfiability Checking* by Erika Ábrahám made the observation that in practice there is a disconnect between the disciplines of SAT solving and symbolic computation. Additionally, she made the case that developing algorithms and tools which combine insights from both these fields is a promising line of research which could be beneficial to both communities. To this end, we propose a research project which uses the techniques from these fields with the aim of aiding progress on conjectures in mathematics, particularly those in combinatorics.

Many conjectures in combinatorial mathematics are simple to state but very hard to verify. For example, a conjecture like the Hadamard [CD07] might assert the existence of certain combinatorial objects in an infinite number of cases, which makes exhaustive search impossible. In such cases, mathematicians often resort to finite verification (i.e., verification up to some finite bound) in the hopes of learning some meta property of the class of combinatorial structures they are investigating, or discover a counterexample to such conjectures. However, even finite verification of combinatorial conjectures can be challenging because the search space for such conjectures is often exponential in the size of the structures they refer to. This makes straightforward brute-force search impractical.

In recent years, conflict-driven clause learning (CDCL) Boolean SAT solvers [BHvMW09, MSS$^+$99, MMZ$^+$01] have become very efficient general-purpose search procedures for a large variety of applications. Indeed, SAT solvers are probably the best general-purpose search procedures we currently have. Despite this remarkable progress these algorithms have worst-case exponential time complexity, and may not perform well by themselves for many search applications—but they can become more efficient with appropriately encoded domain-specific knowledge. By contrast, computer algebra systems (CAS) such as Maple [CFG$^+$86], Mathematica [Wol99], and Sage [BHvMW09] are often a rich storehouse of domain-specific knowledge, but do not generally contain sophisticated general-purpose search procedures.

Fortunately the strengths of modern SAT solvers and CAS are complementary, i.e., the domain-specific knowledge of a CAS can be crucially important in cutting down the search space associated with combinatorial conjectures, while at the same time the clever heuristics of SAT solvers, in conjunction with CAS, can efficiently search a wide variety of such spaces.

1.2 Background on SAT/SMT solvers

A *propositional formula* is an expression involving Boolean variables and logical connectives such as AND ($\land$), OR ($\lor$), and NOT ($\neg$). The *satisfiability problem* is to determine if a given formula is *satisfiable*, i.e., if there exists an assignment to the variables of the formula which make the formula true. A *SAT solver* is a program which solves the satisfiability problem and produces a satisfying assignment in the case that the given formula is satisfiable. In practice, SAT solvers usually accept formulae given in *conjunctive normal form* (CNF), i.e., formulae of the form $\bigwedge_i C_i$ where each C_i is a *clause* (of the form $\bigvee_j l_{ij}$ where each l_{ij} is either a Boolean variable or its negation).

Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. At its core this algorithm performs a depth-first search through the space of possible assignments, stopping when a satisfying assignment has been reached or when the space has been exhausted. Naturally, there are many details and optimizations which make this idea more practical, such as employing conflict-driven clause learning. In more detail, the DPLL algorithm repeatedly performs the following steps:

[1]From Doron Zeilberger's talk at the Fields institute in Toronto, December 2015 (`http://www.fields.utoronto.ca/video-archive/static/2015/12/379-5401/mergedvideo.ogv`, minute 44)

- Decide: Choose an unassigned variable and assign it a value. If all variables have been assigned without a conflict, return the satisfying assignment.
- Deduce: Perform simplifications on the clauses in the given formula to detect conflicts and infer values of variables. The inference process (*Boolean Constant Propagation*) is used repeatedly until no new inferences are made.
- Resolve: If a conflict occurs, learn a clause prohibiting the current assignment and perform a "backjump" by undoing the variable choices leading to the conflict and continuing with another assignment. If the conflict occurs when there are no variable assignments to undo, return UNSAT.

It is also possible to consider the satisfiability problem for the formulas of first-order logic over various theories. Some theories which are commonly used include equality logic with uninterpreted functions, array theory, bitvector theory, and various theories of arithmetic such as integer or real arithmetic (or their linear arithmetic fragments). Programs which are able to solve the satisfiability problem in this context are known as *SAT-modulo-theories* (SMT) solvers.

Modern SMT solvers typically combine a SAT solver with appropriate theory solver(s) to determine satisfiability of first-order formulas. Given a existentially-quantified first-order formula in conjunctive normal form it is possible to abstract the formula into a pure propositional formula by replacing each theory constraint with a fresh propositional variable. A SAT solver is then used to determine if the abstracted formula is satisfiable; if so, the assignment is given to the theory solver to determine if the assignment yields a genuine solution of the original formula. If the assignment does not give rise to a solution then the theory solver determines a clause which encodes a reason why the assignment is prohibited by the theory and passes that to the SAT solver so the search can be resumed.

The above procedure forms the basis of the DPLL(T) algorithm (here T represents the theory under consideration). The procedure is called *lazy* if it waits until all propositional variables have been assigned until querying the theory solver. *Less lazy* variants are possible which also query the theory solver with partial assignments and therefore pass theory lemmas to the SAT solver more frequently.

2 Proof-of-concept: The MathCheck system

In this section we outline a proof-of-concept system we call MathCheck which uses SAT and CAS functionalities to finitely verify conjectures in mathematics. The first version of MathCheck [ZGC15] was developed by Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki and was successfully used to verify two conjectures from graph theory up to bounds which had previously been unobtainable. A SAT solver was the primary search tool used, but the system would periodically make queries to a CAS to learn facts about the remaining search space. These facts were then translated into a form the SAT solver could use and the search was resumed with these additional clauses.

The second version of MathCheck was developed by the author with Vijay Ganesh, Albert Heinle, Ilias Kotsireas, Saeed Nejati, and Krzysztof Czarnecki. It was used to study conjectures in combinatorial design theory, as we describe in Section 2.2. The system can be viewed as a parallel systematic generator of combinatorial structures referred to by the conjecture-under-verification C. It uses a CAS to prune away structures that do not satisfy C, while the SAT solver is used to verify whether any of the remaining structures satisfy C. In addition, if the SAT solver used supports the generation of UNSAT cores (concise encodings of why a formula is unsatisfiable) we use them to further prune the search in a CDCL-style learning feedback loop.

2.1 Architecture of MathCheck2

The architecture of the MathCheck2 system is outlined in Figure 1. At its heart is a generator of combinatorial structures, written in Python, which uses data provided to it by CAS functions to prune the search space and interfaces with SAT solvers to verify the conjecture-in-question. The generation script

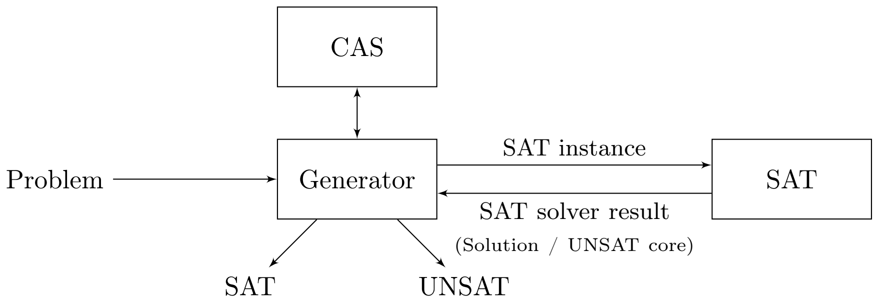

Figure 1: Outline of the architecture of MathCheck2.

contains functions useful for translating combinatorial conditions into clauses which can be read by a SAT solver. In particular, the generator is currently optimized to deal with conjectures which concern Hadamard matrices from coding and combinatorial design theory. The generation script can also be tailored to search for specific classes of Hadamard matrices such as those generated by Williamson matrices (see Theorem 1 in Section 3.2).

Once the class of combinatorial objects has been determined, the script accepts a parameter n which determines the size of the object to search for. For example, when searching for Hadamard matrices, the parameter n denotes the order (i.e., the number of rows) of the matrix. The generation script then queries the CAS it is interfaced with for properties that any order n instance of the combinatorial object in question must satisfy. The result returned by the CAS is read by the generator and then used to prune the space which will be searched by the SAT solver.

Once the generator determines the space to be searched it splits the space into distinct subspaces in a divide-and-conquer fashion. Once the partitioning of the search space has been completed, the script generates two types of files:

1. A single "master" file in DIMACS CNF (conjunctive normal form) format which contains the conditions specifying the combinatorial object being searched for. These are encoded as propositional formulas in conjunctive normal form. An assignment to the variables which makes all of them true would give a valid instance of the object being searched for (and a proof that no such assignment exists proves that no instance of the object in question exists).

2. A set of files which contain partial assignments of the variables in the master file. Each file corresponds to exactly one subspace of the search space produced by the generator.

There are at least 2 advantages of splitting up the problem in such a way:

1. It easily facilitates parallelization. For example, once the instances are generated each instance can be given to a cluster of SAT solvers running in parallel.

2. It allows domain-specific knowledge to be used in the splitting process; partitioning the space in a fortuitous manner can considerably speed up the search, as the SAT solver executes its search without using such domain-specific knowledge.

Furthermore, in cases that an instance is found to be unsatisfiable, some SAT solvers such as MapleSAT [LGPC16], that support the generation of a so-called *UNSAT core*, can be used to further prune away other similar structures that do not satisfy the conjecture-under-verification. Given an unsatisfiable instance ϕ, its UNSAT core is a set of clauses that pithily characterizes the reason why ϕ is unsatisfiable and thus encodes an unsatisfying subspace of the search space.

2.2 Application to the Hadamard conjecture

As a case study of the MathCheck2 system we applied MathCheck2 to the *Hadamard conjecture* from combinatorial design theory. This conjecture states that for any natural number n, there exists a $4n \times 4n$ matrix H with ± 1 entries for which HH^{T} is a diagonal matrix with each diagonal entry equal to $4n$. In particular, we specialize in Hadamard matrices generated by the so-called Williamson method. As described in Section 5.1, we were able to verify that such Hadamard matrices do not exist in order $4 \cdot 35$, a result previously computed using a different methodology by D. Đoković [Đok93]. However, due to the nature of the problem and the techniques used, no short certificate of the computations could be produced, making it difficult to check the work short of re-implementing the approach from scratch. In fact, the author specifically states that

> In the case $n = 35$ our computer search did not produce any solutions [...] Although we are confident about the correctness of this claim, an independent verification of it is highly desirable since this is the first odd integer, found so far, with this property.

Because MathCheck2 was written completely independently, uses different techniques internally, and makes use of well-tested SAT solvers and CAS functions, our system is able to provide an independent verification as requested by Đoković. (The above notes equally apply to the verification in [HKTR08].)

MathCheck2 was also used to show that $n = 35$ is the true smallest number for which no Williamson matrices of order n exist, not merely the smallest *odd* number with this property. This was done by constructing Hadamard matrices using the Williamson method for all orders $n < 35$. We submitted the Hadamard matrices generated by our system (the largest of which has order 156) to the Magma database of Hadamard matrices. There were 160 matrices that we generated which were not previously included in this database.

3 Background on Hadamard matrices and Combinatorial Mathematics

In this section we discuss the mathematical preliminaries necessary to understand MathCheck2 and its application to the Hadamard conjecture.

3.1 Hadamard matrices

First, we define the combinatorial objects known as Hadamard matrices and present some of their properties.

Definition 1. A matrix $H \in \{\pm 1\}^{n \times n}$, $n \in \mathbb{N}$, is called a **Hadamard matrix**, if for all $i \neq j \in \{1, \ldots, n\}$, the dot product between row i and row j in H is equal to zero. We call n the **order** of the Hadamard matrix.

First studied by Hadamard [Had93], he showed that if n is the order of a Hadamard matrix, then either $n = 1$, $n = 2$ or n is a multiple of 4. In other words, he gave a *necessary* condition on n for there to exist a Hadamard matrix of order n. The Hadamard conjecture is that this condition is also *sufficient*, so that there exists a Hadamard matrix of order n for all $n \in \mathbb{N}$ where n is a multiple of 4.

Hadamard matrices play an important role in many widespread branches of mathematics, for example in coding theory [Mul54, Ree54, Wal23] and statistics [HW+78]. Because of this, there is a high interest in the discovery of different Hadamard matrices up to equivalence. Two Hadamard matrices H_1 and H_2 are said to be *equivalent* if H_2 can be generated from H_1 by applying a sequence of negations/permutations to the rows/columns of H_1, i.e., if there exist signed permutation matrices U and V such that $U \cdot H_1 \cdot V = H_2$.

There are several known ways to construct sequences of Hadamard matrices. One of the simplest such constructions is by Sylvester [Syl67]: given a known Hadamard matrix H of order n, $\left[\begin{smallmatrix} H & H \\ H & -H \end{smallmatrix}\right]$ is a Hadamard matrix of order $2n$. This process can of course be iterated, and hence one can construct Hadamard matrices of order $2^k n$ for all $k \in \mathbb{N}$ from H.

There are other methods which produce infinite classes of Hadamard matrices such as those by Paley [Pal33]. However, no general method is known which can construct a Hadamard matrix of order n for arbitrary multiples of 4. The smallest unknown order is currently $n = 4 \cdot 167 = 668$ [CD07]. A database with many known matrices is included in the computer algebra system MAGMA [BCP97]. Further collections are available online [Slo, Seb].

Because there are 2^{n^2} matrices of order n with ±1 entries, the search space of possible Hadamard matrices grows extremely quickly as n increases, and brute-force search is not feasible. Because of this, researchers have defined special types of Hadamard matrices which can be searched for more efficiently because they lie in a small subset of the entire space of Hadamard matrices.

3.2 Williamson matrices

One prominent class of special Hadamard matrices are those generated by so-called Williamson matrices. These are described in this section.

Theorem 1 (cf. [Wil44])**.** *Let $n \in \mathbb{N}$ and let A, B, C, $D \in \{\pm1\}^{n\times n}$. Further, suppose that*

1. *A, B, C, and D are symmetric;*
2. *A, B, C, and D commute pairwise (i.e., $AB = BA$, $AC = CA$, etc.);*
3. *$A^2 + B^2 + C^2 + D^2 = 4nI_n$, where I_n is the identity matrix of order n.*

Then the matrix

$$\begin{bmatrix} A & B & C & D \\ -B & A & -D & C \\ -C & D & A & -B \\ -D & -C & B & A \end{bmatrix}$$

is a Hadamard matrix of order $4n$.

For practical purposes, one considers A, B, C, and D in the Williamson construction to be *circulant* matrices, i.e., those matrices in which every row is the previous row shifted by one entry to the right (with wrap-around, so that the first entry of each row is the last entry of the previous row). Such matrices are completely defined by their first row $[x_0, \dots, x_{n-1}]$ and always satisfy the commutativity property. If the matrix is also symmetric then we must further have $x_1 = x_{n-1}$, $x_2 = x_{n-2}$, and in general $x_i = x_{n-i}$ for $i = 1, \dots, n-1$. Therefore, if a matrix is both symmetric and circulant its first row must be of the form

$$\begin{array}{ll} [x_0, x_1, x_2, \dots, x_{(n-1)/2}, x_{(n-1)/2}, \dots, x_2, x_1] & \text{if } n \text{ is odd} \\ [x_0, x_1, x_2, \dots, x_{n/2-1}, x_{n/2}, x_{n/2-1}, \dots, x_2, x_1] & \text{if } n \text{ is even.} \end{array} \tag{1}$$

Definition 2. A **symmetric** sequence of length n is one of the form (1), i.e., one which satisfies $x_i = x_{n-i}$ for $i = 1, \dots, n-1$.

Williamson matrices are circulant matrices A, B, C, and D which satisfy the conditions of Theorem 1. Since they must be circulant, they are completely defined by their first row. (In light of this, we may simply refer to them as if they were sequences.) Furthermore, since they are symmetric the Hadamard matrix generated by these matrices is completely specified by the $4\lceil\frac{n+1}{2}\rceil$ variables

$$a_0, a_1, \dots, a_{\lceil(n-1)/2\rceil}, b_0, \dots, b_{\lceil(n-1)/2\rceil}, c_0, \dots, c_{\lceil(n-1)/2\rceil}, d_0, \dots, d_{\lceil(n-1)/2\rceil}.$$

Given an assignment of these variables, the rest of the entries of the matrices A, B, C, and D may be chosen in such a way that conditions 1 and 2 of Theorem 1 always hold. There is no trivial way of enforcing condition 3, but we will later derive consequences of this condition which will simplify the search for matrices which satisfy it.

There are three types of operations which, when applied to the Williamson matrices, produce different but essentially equivalent matrices. For our purposes, generating just one of the equivalent matrices will be sufficient, so we impose additional constraints on the search space to cut down on extraneous solutions and hence speed up the search.

1. Ordering: Note that the conditions on the Williamson matrices are symmetric with respect to A, B, C, and D. In other words, those four matrices can be permuted amongst themselves and they will still generate a valid Hadamard matrix. Given this, we enforce the constraint that

$$|\text{rowsum}(A)| \leq |\text{rowsum}(B)| \leq |\text{rowsum}(C)| \leq |\text{rowsum}(D)|,$$

where rowsum(X) denotes the sum of the entries of the first (or any) row of X. Any A, B, C, and D can be permuted so that this condition holds.

2. Negation: The entries in the sequences defining any of A, B, C, or D can be negated and the sequences will still generate a Hadamard matrix. Given this, we do not need to try both possibilities for the sign of the rowsum of A, B, C, and D. For example, we can choose to enforce that the rowsum of each of the generating matrices is nonnegative. Alternatively, when n is odd we can choose the signs so they satisfy rowsum$(X) \equiv n \pmod 4$ for $X \in \{A, B, C, D\}$. In this case, a result of Williamson [Wil44] says that $a_i b_i c_i d_i = -1$ for all $1 \leq i \leq (n-1)/2$.

3. Permuting entries: We can reorder the entries of the generating sequences with the rule $a_i \mapsto a_{ki \bmod n}$ where k is any number coprime with n, and similarly for b_i, c_i, d_i (the *same* reordering must be applied to each sequence for the result to still be equivalent). Such a rule effectively applies an automorphism of $\mathbb{Z}_n$ to the generating sequences.

3.3 Power spectral density

Because the search space for Hadamard matrices is so large, it is advantageous to focus on a specific construction method and describe properties which any Hadamard matrix generated by this specific method must satisfy; such properties can speed up a search by significantly reducing the size of the necessary space. One such set of properties for Williamson matrices is derived using the discrete Fourier transform from Fourier analysis. The *discrete Fourier transform* of a sequence $A = [a_0, a_1, \dots, a_{n-1}]$ is the periodic function

$$\text{DFT}_A(s) \coloneqq \sum_{k=0}^{n-1} a_k \omega^{ks} \qquad \text{for } s \in \mathbb{Z},$$

where $\omega \coloneqq e^{2\pi i/n}$ is a primitive nth root of unity. Because $\omega^{ks} = \omega^{ks \bmod n}$ one has that $\text{DFT}_A(s) = \text{DFT}_A(s \bmod n)$, so that only n values of DFT_A need to be computed and the remaining values are determined through periodicity. In fact, when A consists of real entries, it is well-known that $\text{DFT}_A(s)$ is equal to the complex conjugate of $\text{DFT}_A(n-s)$. Hence only $\lfloor \frac{n+1}{2} \rfloor$ values of DFT_A need to be computed.

The *power spectral density* of the sequence A is given by

$$\text{PSD}_A(s) \coloneqq |\text{DFT}_A(s)|^2 \qquad \text{for } s \in \mathbb{Z}.$$

Note that $\text{PSD}_A(s)$ will always be a nonnegative real number.

Example 1. *Let $\omega = e^{2\pi i/5}$. The discrete Fourier transform and power spectral density of the sequence $A = [1, 1, -1, -1, 1]$ are given by:*

$$\begin{aligned}
\text{DFT}_A(0) &= 1 + 1 - 1 - 1 + 1 = 1 & \text{PSD}_A(0) &= 1 \\
\text{DFT}_A(1) &= 1 + \omega - \omega^2 - \omega^3 + \omega^4 \approx 3.236 & \text{PSD}_A(1) &\approx 10.472 \\
\text{DFT}_A(2) &= 1 + \omega^2 - \omega^4 - \omega^6 + \omega^8 & & \\
&= 1 - \omega + \omega^2 + \omega^3 - \omega^4 \approx -1.236 & \text{PSD}_A(2) &\approx 1.528
\end{aligned}$$

3.4 Periodic autocorrelation

As we will see, the defining properties of Williamson matrices (in particular, condition 3 of Theorem 1) can be re-cast using a function known as the periodic autocorrelation function (PAF). Re-casting the equations in this way is advantageous because many other combinatorial conjectures can also be stated in terms of the PAF. Hence, code which is used to counter-example or finitely verify one such conjecture can be re-applied to many other conjectures.

Definition 3. The **periodic autocorrelation function** of the sequence A is the periodic function given by

$$\mathrm{PAF}_A(s) \coloneqq \sum_{k=0}^{n-1} a_k a_{(k+s) \bmod n} \qquad \text{for } s \in \mathbb{Z}.$$

Similar to the discrete Fourier transform, one has $\mathrm{PAF}_A(s) = \mathrm{PAF}_A(s \bmod n)$ and $\mathrm{PAF}_A(s) = \mathrm{PAF}_A(n-s)$ (see [Kot13a]), so that the PAF_A only needs to be computed for $s = 0, \dots, \lfloor \frac{n-1}{2} \rfloor$; the other values can be computed through symmetry and periodicity.

Example 2. *The values of the periodic autocorrelation function of the sequence* $A = [1, 1, -1, -1, 1]$ *are given by:*

$$\begin{aligned}
\mathrm{PAF}_A(0) &= 1^2 + 1^2 + (-1)^2 + (-1)^2 + 1^2 = 5 \\
\mathrm{PAF}_A(1) &= 1^2 + (-1) + (-1)^2 + (-1) + 1^2 = 1 \\
\mathrm{PAF}_A(2) &= (-1) + (-1) + (-1) + (-1) + 1^2 = -3
\end{aligned}$$

By symmetry, $\mathrm{PAF}_A(3) = \mathrm{PAF}_A(2)$ *and* $\mathrm{PAF}_A(4) = \mathrm{PAF}_A(1)$.

Now we will see how to rewrite condition 3 of Theorem 1 using PAF values. Note that the sth entry in the first row of $A^2 + B^2 + C^2 + D^2$ is

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s).$$

Condition 3 requires that this entry should be $4n$ when $s = 0$ and it should be 0 when $s = 1, \dots, n-1$. The condition when s is 0 does not need to be explicitly checked because in that case the sum will always be $4n$, as $\mathrm{PAF}_A(0) = \sum_{k=0}^{n-1} (\pm 1)^2 = n$ and similarly for B, C, and D.

Additionally, the first row of $A^2 + B^2 + C^2 + D^2$ will be symmetric as each matrix in the sum has a symmetric first row. Thus ensuring that

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = 0 \quad \text{for } s = 1, \dots, \lceil \tfrac{n-1}{2} \rceil \tag{2}$$

guarantees that every entry in the first row of $A^2 + B^2 + C^2 + D^2$ is 0 besides the first. Since $A^2 + B^2 + C^2 + D^2$ will also be circulant, ensuring that (2) holds will ensure condition 3 of Theorem 1.

3.5 Compression

Because the space in which a combinatorial object lies is proportional to the size of the object, it is advantageous to instead search for *smaller* objects when possible. Recent theorems on so-called "compressed" sequences allow us to do that when searching for Williamson matrices.

Definition 4 (cf. [ĐK15])**.** Let $A = [a_0, a_1, \dots, a_{n-1}]$ be a sequence of length $n = dm$ and set

$$a_j^{(d)} = a_j + a_{j+d} + \dots + a_{j+(m-1)d}, \qquad j = 0, \dots, d-1.$$

Then we say that the sequence $A^{(d)} = [a_0^{(d)}, a_1^{(d)}, \dots, a_{d-1}^{(d)}]$ is the ***m*-compression** of A.

Example 3. *The sequence* $A = [1, 1, -1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, 1]$ *of length* 15 *has the following two compressions:*

$$A^{(3)} = [-3, 1, 1] \qquad \textit{and} \qquad A^{(5)} = [3, -1, -1, -1, -1].$$

As we will see, the space of the compressed sequences that we are interested in will be much smaller than the space of the uncompressed sequences. What makes compressed sequences especially useful is that we can derive conditions that the compressed sequences must satisfy using our known conditions on the uncompressed sequences. To do this, we utilize the following theorem which is a special case of a result from [ĐK15].

Theorem 2. *Let* A, B, C, *and* D *be sequences of length* $n = dm$ *which satisfy*

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = \begin{cases} 4n & \text{if } s = 0 \\ 0 & \text{if } 1 \le s < \mathrm{len}(A). \end{cases} \tag{3}$$

Then for all $s \in \mathbb{Z}$ *we have*

$$\mathrm{PSD}_A(s) + \mathrm{PSD}_B(s) + \mathrm{PSD}_C(s) + \mathrm{PSD}_D(s) = 4n. \tag{4}$$

Furthermore, both (3) *and* (4) *hold if the sequences* A, B, C, D *are replaced with their compressions* $A^{(d)}$, $B^{(d)}$, $C^{(d)}$, $D^{(d)}$.

Since $\mathrm{PSD}_X(s)$ is always nonnegative, equation (4) implies that $\mathrm{PSD}_{A^{(d)}}(s) \le 4n$ (and similarly for B, C, D). Therefore if a candidate compressed sequence $A^{(d)}$ satisfies $\mathrm{PSD}_{A^{(d)}}(s) > 4n$ for some $s \in \mathbb{Z}$ then we know that the uncompressed sequence A can never be one of the sequences which satisfies the preconditions of Theorem 2.

Useful properties: Lastly, we derive some properties that the compressed sequences which arise in our context must satisfy. For a concrete example, note that the compressed sequences of Example 3 fulfill these properties.

Lemma 1. *If* A *is a sequence of length* $n = dm$ *with* ± 1 *entries, then the entries* $a_i^{(d)}$, $i \in \{0, \dots, d-1\}$, *have absolute value at most* m *and* $a_i^{(d)} \equiv m \pmod 2$.

Proof. For all $0 \le j < d$ we have, using the triangle inequality, that

$$\left|a_j^{(d)}\right| = \left|\sum_{k=0}^{m-1} a_{j+kd}\right| \le \sum_{k=0}^{m-1} |a_{j+kd}| = m.$$

Additionally, $a_j^{(d)} \equiv \sum_{k=0}^{m-1} 1 \equiv m \pmod 2$ since $a_{j+kd} \equiv 1 \pmod 2$. □

Lemma 2. *The compression of a symmetric sequence is also symmetric.*

Proof. Suppose that A is a symmetric sequence of length $n = dm$. We want to show that $a_j^{(d)} = a_{d-j}^{(d)}$ for $j = 1, \dots, d-1$. By reversing the sum defining $a_j^{(d)}$ and then using the fact that $n = md$, we have

$$\sum_{k=0}^{m-1} a_{j+kd} = \sum_{k=0}^{m-1} a_{j+(m-1-k)d} = \sum_{k=0}^{m-1} a_{n+j-d(k+1)}.$$

By the symmetry of A, $a_{n+j-d(k+1)} = a_{d(k+1)-j}$, which equals a_{d-j+dk}. The sum in question is therefore equal to $\sum_{k=0}^{m-1} a_{d-j+dk} = a_{d-j}^{(d)}$, as required. □

4 Encoding and Search Space Pruning Techniques

An attractive property of Hadamard matrices when encoding them in a SAT context is that each of their entries is one of two possible values, namely ± 1. We choose the encoding that 1 is represented by true and -1 is represented by false. We call this the *Boolean value* or *BV* encoding. Under this encoding, the multiplication function of two x, $y \in \{\pm 1\}$ becomes the XNOR function in the SAT setting, i.e., $\mathrm{BV}(x \cdot y) = \mathrm{XNOR}(\mathrm{BV}(x), \mathrm{BV}(y))$.

4.1 Naive encoding of Hadamard matrices in SAT

In order to check if a matrix $H \in \{\pm 1\}^{n \times n}$ with rows $H_0, \ldots, H_{n-1}$ is Hadamard, it is necessary to verify that $H_i \cdot H_j = 0$ for all $0 \le i, j < n$ with $i \ne j$. In other words, we want to verify that the component-wise product

$$H_i * H_j = \begin{bmatrix} h_{i,0} \cdot h_{j,0} & h_{i,1} \cdot h_{j,1} & \cdots & h_{i,n-1} \cdot h_{j,n-1} \end{bmatrix}$$

has a row sum of 0. To compute $h_{ik} \cdot h_{jk}$ in the SAT setting we define the new 'product' variables $p_{ijk} \coloneqq \mathrm{XNOR}(\mathrm{BV}(h_{ik}), \mathrm{BV}(h_{jk}))$ for all $0 \le i, j, k < n$ with $i < j$; these variables store the Boolean values of the entries of $H_i * H_j$. In order to add together the entries of $H_i * H_j$ as Boolean values, we employ a network of full and half bit adders. A half adder consumes two Boolean values and produces two Boolean values; when thought of as bits, the two outputs store the binary representation of the sum of the inputs. (A full adder does the same thing, but consumes three inputs.)

Repeatedly using the adders on the set of variables p_{ijk} for $k = 0, \ldots, n-1$ yields $\lfloor \log_2 n \rfloor + 1$ new variables which store the binary representation of $\sum_{k=0}^{n-1} p_{ijk}$. We want there to be $n/2$ true BVs and $n/2$ false BVs in this sum so that the rowsum of $H_i * H_j$ is zero. Therefore we want to ensure the binary representation of $\sum_{k=0}^{n-1} p_{ijk}$ is exactly $n/2$, because false BVs count for 0 and true BVs count for 1 in the adder network.

4.2 Williamson autocorrelation encoding

The Williamson encoding is very similar to the general encoding but with fewer variables; we merely have the $4\lceil \frac{n+1}{2} \rceil$ variables

$$a_0, a_1, \ldots, a_{\lceil (n-1)/2 \rceil}, b_0, \ldots, b_{\lceil (n-1)/2 \rceil}, c_0, \ldots, c_{\lceil (n-1)/2 \rceil}, d_0, \ldots, d_{\lceil (n-1)/2 \rceil}.$$

Also, instead of the conditions $\mathrm{rowsum}(H_i * H_j) = 0$ for $i \ne j$ we must enforce the conditions

$$\mathrm{rowsum}(A_i * A_j + B_i * B_j + C_i * C_j + D_i * D_j) = 0 \quad \text{for } i \ne j.$$

Like in Section 4.1 this is done by defining new variables to represent the entries of the component-wise products. Also, note that because of the circulant property most of the conditions to enforce will be identical. As previously mentioned in Section 3.4, it is only necessary to encode the $\lceil \frac{n-1}{2} \rceil$ autocorrelation equations given in (2) to ensure that such matrices generate a valid Hadamard matrix.

4.3 Technique 1: Sum-of-squares decomposition

As a special case of compression, consider what happens when $d = 1$ and $m = n$. In this case, the compression of A is a sequence with a single entry whose value is $\sum_{k=0}^{n-1} a_k = \mathrm{rowsum}(A)$. If A, B, C, and D are $\{\pm 1\}$-sequences which satisfy the conditions of Theorem 2, then the theorem applied to this m-compression says that

$$\mathrm{PAF}_{A^{(1)}}(0) + \mathrm{PAF}_{B^{(1)}}(0) + \mathrm{PAF}_{C^{(1)}}(0) + \mathrm{PAF}_{D^{(1)}}(0) = 4n$$

which simplifies to

$$\mathrm{rowsum}(A)^2 + \mathrm{rowsum}(B)^2 + \mathrm{rowsum}(C)^2 + \mathrm{rowsum}(D)^2 = 4n,$$

and by Lemma 1 each rowsum must have the same parity as n.

In other words, the rowsums of the sequences A, B, C, and D decompose $4n$ into the sum of four perfect squares whose parity matches the parity of n. Since there are usually only a few ways of writing $4n$ as a sum of four perfect squares this severely limits the number of sequences which could satisfy the hypotheses of Theorem 2. Furthermore, some computer algebra systems contain functions for explicitly computing what the possible decompositions are (e.g., `PowersRepresentations` in Mathematica and `nsoks` by Joe Riel of Maplesoft [Rie]). We can query such CAS functions to determine all possible values that the rowsums of A, B, C, and D could possibly take. For example, when $n = 35$ we find that there are exactly three ways to write $4n$ as a sum of four positive odd squares, namely,

$$1^2 + 3^2 + 3^2 + 11^2 = 1^2 + 3^2 + 7^2 + 9^2 = 3^2 + 5^2 + 5^2 + 9^2 = 4 \cdot 35.$$

When using this technique it is necessary to encode constraints on the rowsum of the generating matrices, e.g., $\operatorname{rowsum}(A) = 1$. This may be simply done by using a binary adder network on the variables $a_0, \dots, a_{\lceil (n-1)/2 \rceil}$. We give the variables which appear twice in the first row of A (due to symmetry) a weight of 2 in the binary adder network so that the rowsum is computed correctly.

4.4 Technique 2: Divide-and-conquer

Because each instance can take a significant amount of time to solve, it is beneficial to divide instances into multiple partitions, each instance encoding a subset of the search space. In our case, we found that an effective splitting method was to split by compressions, i.e., to have each instance contain one possibility of the compressions of A, B, C, and D. To do this, we first need to know all possible compressions of A, B, C, and D. These can be generated by applying Lemmas 1 and 2. For example, when $n = 35$ and $d = 5$ there are 28 possible compressions of A with $\operatorname{rowsum}(A) = 1$. Of those, only 12 satisfy $\operatorname{PSD}_A(s) \leq 4n$ for all $s \in \mathbb{Z}$. There are also 12 possible compressions for each of B, C, and D with $\operatorname{rowsum}(B) = \operatorname{rowsum}(C) = 3$ and $\operatorname{rowsum}(D) = 11$. Thus there are 12^4 total instances which would need to be generated for this selection of rowsums, however, only 41 of them satisfy the conditions given by Theorem 2.

Furthermore, if n has two nontrivial divisors m and d then we can find all possible m-compressions and d-compressions of A, B, C, and D. In this case, each instance can set *both* the m-compression and the d-compression of each of A, B, C, and D. Since there are more combinations to check when dealing with two types of compression this causes an increase in the number of instances generated, but each instance has more constraints and a smaller subspace to search through.

4.5 Technique 3: UNSAT core

After using the divide-and-conquer technique one obtains a collection of instances which are almost identical. For example, the instances will contain variables which encode the rowsums of A, B, C, and D. Since there are multiple possibilities of the rowsums (as discussed in Section 4.3), not all instances will set those variables to the same values. However, since the instances are the same except for those variables, it is sometimes possible to use an *UNSAT core* result from one instance to learn that other instances are unsatisfiable.

MapleSAT is one SAT solver which supports UNSAT core generation. Provided a master instance and a set of assumptions (variables which are set either true or false), the UNSAT core contains a subset of the assumptions which make the master instance unsatisfiable. Thus, any other instance which sets the variables in the UNSAT core in the same way must also be unsatisfiable.

5 Experimental Results

5.1 Nonexistence of Williamson matrices of order 35

We searched for Williamson matrices of order 35 using the techniques described in Section 4 with both 5 and 7-compression. Despite the exponential growth of possible first rows of the matrices A, B, C, and D, the described pruning results in 21,674 SAT instances of three possible forms, as described in Figure 2. Each

rowsum(A)	rowsum(B)	rowsum(C)	rowsum(D)	Number of Instances
1	3	3	11	6960
1	3	7	9	8424
3	5	5	9	6290

Figure 2: The number of instances of each type generated in the process of searching for Williamson matrices of order 35.

Order	Base Encoding (Sec. 4.2)	Technique 1 (Sec. 4.3)	Technique 2 (Sec. 4.4)	Technique 3 (Sec. 4.5)
25	317s (1)	1702s (4)	408s (179)	408s (179)
26	865s (1)	3818s (3)	61s (3136)	34s (1592)
27	5340s (1)	8593s (3)	1518s (14994)	1439s (689)
28	7674s (1)	2104s (2)	234s (13360)	158s (439)
29	-	21304s (1)	N/A	N/A
30	1684s (1)	36804s (1)	139s (370)	139s (370)
31	-	83010s (1)	N/A	N/A
32	-	-	96011s (13824)	95891s (348)
33	-	-	693s (8724)	683s (7603)
34	-	-	854s (732)	854s (732)
35	-	-	31816s (21674)	31792s (19356)

Figure 3: The numbers in parentheses denote how many MapleSAT calls successfully returned a result for the given Williamson order. The timings refer to the total amount of time used during those calls. A hyphen denotes a timeout after 24 hours.

instance has subsequently been checked with several SAT solvers, and each one has been discovered to be unsatisfiable. Using MapleSAT with UNSAT core generation, 19,356 SAT solver calls were necessary to determine that all instances were unsatisfiable.

5.2 Experimental setup and methodology

Timings were run on the high-performance computing cluster SHARCNET. Specifically, the cluster we used ran CentOS 5.4 and used 64-bit AMD Opteron processors running at 2.2 GHz. Each SAT instance was generated using MathCheck2 with the appropriate parameters and the instance was submitted to SHARCNET to solve by running MapleSAT on a single core (with a timeout of 24 hours).

Figure 3 contains a summary of the performance of our encoding and pruning techniques. The timings are for searching for Williamson matrices of order n with $25 \leq n \leq 35$ and for each of the techniques discussed in Section 4. We did not use Techniques 2 and 3 for orders 29 and 31 as they have no nontrivial divisors to perform compression with, but they were otherwise very effective at partitioning the search space in an efficient way. Technique 3 was effective at cutting down the number of instances generated in certain orders. Although the instances pruned tended to be those which would have been quickly solved, this technique would be especially valuable in a situation where few cores are available, as it allows many SAT solver calls (which have a fixed overhead) to be avoided.

6 Proposal for Future Work

We propose to extend our work in the following directions:

1. Scale our current MathCheck2 system to Hadamard matrices of higher order and find new inequivalent

Hadamard matrices. For this, we plan to refine and generalize our method to apply to other Hadamard construction types. As a concrete example, some types we plan to consider include *Hadamard matrices with circulant core* [KKS06b] and *Hadamard matrices with two circulant cores* [KKS06a]. Both of these Hadamard matrix types can be defined using a similar number of variables (i.e., $O(n)$) as the Williamson construction.

The latter type is especially interesting to consider because of the *structured Hadamard conjecture* [Kot13b] of Kotsireas, Koukouvinos, and Seberry. This states that there exists a Hadamard matrix with two circulant cores of order $4n$ for all $n > 0$. (In contrast, most other types of Hadamard matrices are either known or expected to not exist for all orders.)

2. Use our system to search for other combinatorial objects which can be defined by conditions using the autocorrelation function. For example, in *Algorithms and Metaheuristics for Combinatorial Matrices* [Kot13a] Kotsireas lists at least eleven different types of combinatorial objects which are simply defined using the autocorrelation function (either periodic or aperiodic). The periodic autocorrelation function has been previously defined, and the aperiodic autocorrelation function (AAF) is defined by

$$\mathrm{AAF}_A(s) \coloneqq \sum_{k=0}^{n-s-1} a_k \bar{a}_{(k+s) \bmod n} \qquad \text{for } s = 0, \dots, n-1.$$

for a complex sequence $A = [a_0, \dots, a_{n-1}]$, where the bar represents complex conjugation. (In contrast to the PAF sum which has n terms, the AAF sum contains only $n - s$ terms.)

A concrete example of a combinatorial object we plan to consider are *complex Golay sequences.* These are pairs A, B of sequences of length n whose entries are from $\{\pm 1, \pm i\}$ and which satisfy

$$\mathrm{AAF}_A(s) + \mathrm{AAF}_B(s) = 0 \qquad \text{for } s = 1, \dots, n-1.$$

The smallest open case for this problem is $n = 23$ [CHK02]. Since the entries of the sequences can take 4 possible values, it will be necessary to use two Boolean variables to encode each entry. This will necessarily make the encoding of multiplication more complicated, however, preliminary experimentation shows that it is possible to construct a logic circuit expressed in conjunctive normal form which encodes $x \cdot \bar{y}$ for two numbers $x, y \in \{\pm 1, \pm i\}$ with sixteen clauses.

3. Make custom modifications to the SAT solvers used; this allows domain-specific code to be used which is specifically tailored to each conjecture under verification. The idea of modifying a SAT solver in this way to make a *"programmatic SAT solver"* was introduced by Ganesh et al. [GOS$^+$12]. Using a special API they allow user-provided code to examine partial solutions as the search progresses and programmatically influence the behaviour of the solver by generating problem-specific learned clauses. Additionally, they showed that the programmatic SAT approach was up to 100 times more efficient than a standard SAT approach in the context of RNA folding problems.

In our context, the programmatic SAT idea has many potential benefits:

(a) Increased expressiveness. It is often simpler to express a condition in a programmatic way rather than as a CNF propositional formula.

(b) Efficiency. The fact that some conditions require many additional new variables and clauses to be added to the SAT instances can be detrimental to the SAT solver's performance.

For example, the MathCheck2-generated Williamson instances required an encoding of constraints like $\mathrm{rowsum}(A) = 1$. From this we can determine the number of the variables $a_0, \dots, a_{n-1}$ which must be true and the number which must be false. Instead of encoding this constraint using a network of binary adders, we can have the SAT solver keep track of how many $a_0, \dots, a_{n-1}$ are true and false and stop the search when any partial solution has counts which are inconsistent with the given constraint.

(c) Better leverage of CAS functionality. The current version of MathCheck2 uses the results from a CAS to derive conditions which limit the search space, but does not use a CAS while the search is in progress. However, calling CAS functions from inside the SAT solver will allow theory-specific lemmas to be learned as the search progresses (similar to the DPLL(T) algorithm).

This was found to be useful in the first version of MathCheck [ZGC15], which for example made use of a function in Sage to find the shortest path between two points. In our context, we plan to use CAS functions to compute automorphisms which would allow the detection of symmetries and removal of isomorphic solutions. For example, the last equivalence operation of Section 3.2 (permuting entries) is difficult to encode using propositional formulae but should be tractable using a CAS.

7 Related Work

The idea of combining the capabilities of SAT/SMT solvers and computer algebra systems or domain-specific knowledge has been examined by various research groups. Junges et al. [JLCÁ13] studied an integration of Gröbner basis theory in the context of SMT solvers. Although they implemented their own version of Buchberger's algorithm, they describe that it is possible to "plug in an off-the-shelf GB procedure implementation such as the one in Singular" as the core procedure. Singular [DGPS15] is a computer algebra system with specialized algorithms for polynomial systems. Ábrahám later highlights the potentials of combining symbolic computation and SMT solving in [Á15]. The veriT SMT solver [BDODF09] uses the computer algebra system Reduce [Hea04] to support non-linear arithmetic. The Lean theorem prover [dMKA+15] combines domain-specific knowledge with SMT solvers. SAT-based results on the Erdős discrepancy conjecture [KL14] inspired the previous version of MathCheck [ZGC15].

8 References

[Á15] Erika Ábrahám. Building bridges between symbolic computation and satisfiability checking. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 1–6. ACM, 2015.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265, 1997.

[BDODF09] Thomas Bouton, Diego Caminha B De Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185.* ios Press, 2009.

[CD07] Charles J. Colbourn and Jeffrey H. Dinitz, editors. *Handbook of combinatorial designs.* Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, second edition, 2007.

[CFG+86] Bruce W Char, Gregory J Fee, Keith O Geddes, Gaston H Gonnet, and Michael B Monagan. A tutorial introduction to Maple. *Journal of Symbolic Computation*, 2(2):179–200, 1986.

[CHK02] Robert Craigen, W Holzmann, and Hadi Kharaghani. Complex Golay sequences: structure and applications. *Discrete mathematics*, 252(1):73–89, 2002.

[DGPS15] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. Singular 4-0-2 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2015.

[dMKA$^+$15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The LEAN theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer International Publishing, 2015.

[GOS$^+$12] Vijay Ganesh, Charles W O'Donnell, Mate Soos, Srinivas Devadas, Martin C Rinard, and Armando Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 143–156. Springer, 2012.

[Had93] Jacques Hadamard. Résolution d'une question relative aux déterminants. *Bull. sci. math*, 17(1):240–246, 1893.

[Hea04] AC Hearn. REDUCE user's manual, version 3.8, 2004.

[HKTR08] Wolf H Holzmann, Hadi Kharaghani, and Behruz Tayfeh-Rezaie. Williamson matrices up to order 59. *Designs, Codes and Cryptography*, 46(3):343–352, 2008.

[HW$^+$78] A Hedayat, WD Wallis, et al. Hadamard matrices and their applications. *The Annals of Statistics*, 6(6):1184–1238, 1978.

[JLCÁ13] Sebastian Junges, Ulrich Loup, Florian Corzilius, and Erika Ábrahám. On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. In *Algebraic Informatics*, pages 186–198. Springer, 2013.

[ĐK15] Dragomir Ž Đoković and Ilias S Kotsireas. Compression of periodic complementary sequences and applications. *Designs, Codes and Cryptography*, 74(2):365–377, 2015.

[KKS06a] Ilias S Kotsireas, Christos Koukouvinos, and Jennifer Seberry. Hadamard ideals and Hadamard matrices with two circulant cores. *European Journal of Combinatorics*, 27(5):658–668, 2006.

[KKS06b] Ilias S Kotsireas, Christos Koukouvinos, and Jennifer Seberry. Hadamard ideas and Hadamard matrices with circulant core. *JOURNAL OF COMBINATORIAL MATHEMATICS AND COMBINATORIAL COMPUTING*, 57:47, 2006.

[KL14] Boris Konev and Alexei Lisitsa. A SAT attack on the erdős discrepancy conjecture. In *Theory and Applications of Satisfiability Testing–SAT 2014*, pages 219–226. Springer, 2014.

[Kot13a] Ilias S Kotsireas. Algorithms and metaheuristics for combinatorial matrices. In *Handbook of Combinatorial Optimization*, pages 283–309. Springer, 2013.

[Kot13b] Ilias S Kotsireas. Structured Hadamard conjecture. In *Number Theory and Related Fields*, pages 215–227. Springer, 2013.

[LGPC16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average branching heuristic for SAT solvers. In *Proceedings of AAAI-16*, 2016.

[MMZ$^+$01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[MSS$^+$99] João P Marques-Silva, Karem Sakallah, et al. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[Mul54] David E Muller. Application of boolean algebra to switching circuit design and to error detection. *Electronic Computers, Transactions of the IRE Professional Group on*, (3):6–12, 1954.

[Đok93] Dragomir Ž Đoković. Williamson matrices of order $4n$ for $n = 33, 35, 39$. *Discrete mathematics*, 115(1):267–271, 1993.

[Pal33] Raymond EAC Paley. On orthogonal matrices. *J. Math. Phys.*, pages 311–320, 1933.

[Ree54] Irving Reed. A class of multiple-error-correcting codes and the decoding scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49, 1954.

[Rie] Joe Riel. `nsoks`: A Maple script for writing n as a sum of k squares. `https://sites.google.com/site/uwmathcheck/nsoks.mpl`.

[Seb] Jennifer Seberry. Library of Williamson matrices. `http://www.uow.edu.au/~jennie/WILLIAMSON/williamson.html`.

[Slo] Neil Sloane. A library of Hadamard matrices. `http://neilsloane.com/hadamard/`.

[Syl67] James Joseph Sylvester. Thoughts on inverse orthogonal matrices, simultaneous signsuccessions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 34(232):461–475, 1867.

[Wal23] Joseph L Walsh. A closed set of normal orthogonal functions. *American Journal of Mathematics*, pages 5–24, 1923.

[Wil44] John Williamson. Hadamard's determinant theorem and the sum of four squares. *Duke Math. J*, 11(1):65–81, 1944.

[Wol99] Stephen Wolfram. *The Mathematica book, version 4*. Cambridge university press, 1999.

[ZGC15] Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki. MathCheck: A math assistant via a combination of computer algebra systems and SAT solvers. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 607–622. Springer International Publishing, 2015.