MATHCHECK2: A SAT+CAS Verifier for Combinatorial Conjectures

Curtis Bright*, Vijay Ganesh*, Albert Heinle*, Ilias Kotsireas†, Saeed Nejati*, and Krzysztof Czarnecki*
*University of Waterloo
†Wilfrid Laurier University

***Abstract*—In this paper, we outline MATHCHECK2, a combination of a SAT solver and a computer algebra system (CAS) aimed at finitely verifying or counterexampling mathematical conjectures. Using MATHCHECK2 we verified the Hadamard conjecture from design theory for matrices up to order 144 and many additional orders up to 168. Also, we provide independent verification of the claim that Williamson matrices of order 35 do not exist, and demonstrate for the first time that 35 is the smallest number with this property. In the course of our work, we discovered over 500 Hadamard matrices which were not equivalent to any matrices in the comprehensive MAGMA Hadamard database.**

The crucial insight behind MATHCHECK2 is that a combination of an efficient search procedure (like those in SAT solvers) with a domain-specific knowledge base (à la CAS) can be a very effective way to verify, counterexample, and learn deeper properties of mathematical conjectures (especially in combinatorics) and the structures they refer to. MATHCHECK2 can be seen as a systematic parallel generator of structures referred to by the conjecture-under-verification C, and these conjectures are typically of the form "for all natural numbers n, some combinatorial object exists". MATHCHECK2 uses a divide-and-conquer approach to parallelize the search, and a CAS to prune away classes whose members are guaranteed to not satisfy the conjecture C. The SAT solver is used to verify whether any of the remaining structures for each number n satisfy C, and in addition learn UNSAT cores in a conflict-driven clause-learning style feedback loop to further prune the search space. Finally, our latest version of MATHCHECK2 (updated from our previously presented version [5]) uses programmatic methods to learn clauses which are generated on-the-fly by domain-specific constraints.

I. INTRODUCTION

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

Many conjectures in combinatorial mathematics are simple to state but very hard to verify. For example, a conjecture like the Hadamard [7] 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 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 up to some finite bound is very difficult, 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, and also ansatz-driven methods do not scale well enough in general.

In recent years, conflict-driven clause-learning (CDCL) Boolean SAT solvers [3, 18, 19] have become very efficient general-purpose search procedures for a large variety of applications. Despite this remarkable progress these algorithms have worst-case exponential time complexity, and may not perform well by themselves for many search applications. Put differently, SAT solvers are probably the best general-purpose search procedures we currently have, and can become more efficient with appropriately encoded domain-specific knowledge. By contrast, computer algebra systems (CAS) such as MAPLE [6], MATHEMATICA [27], and SAGE [3] are often a rich storehouse of algorithms to obtain domain-specific knowledge, but do not 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.

The success of the SAT and CAS combination has been demonstrated in papers on MATHCHECK [28] and MATHCHECK2 [5]. In the first paper, Ganesh et al. explored one way of combining these two classes of systems wherein the CAS was used as a theory solver, à la DPLL(T), to add theory lemmas to the SAT solvers that was the primary driver of the search. They used MATHCHECK to finitely verify (i.e., verify up to some finite bound) conjectures from graph theory. In the second paper, the method was generalized to work with combinatorial structures, in particular Hadamard matrices. *Hadamard matrices* are $4n \times 4n$ matrices H with ± 1 entries for which HH^{T} is a diagonal matrix with each diagonal entry equal to $4n$. The *Hadamard conjecture* states that such a matrix exists for any natural number n. In particular, we specialize in Hadamard matrices generated by the so-called Williamson method. MATHCHECK was restructured to contain a generator, which divides the seach space a priori into equivalence classes using CAS systems. The additional information about each of these equivalence classes enabled us to ignore many of the classes in our search, since we were able to prove with a CAS that these cannot contain a solution. We were able to construct Hadamard matrices – using the Williamson construction method

[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)

– for orders as large as 168, and verified that Williamson matrices do not exist in order 35. This was a result which was previously computed using a different methodology by D. Đoković [21], who requested an independent verification.

In this paper, we present a refinement of our methods and an improved implementation of MATHCHECK2. Previously, we observed that many of the generated SAT instances used a common set of clauses to help prune away the search space. With more and more of these pruning results included the number of these clauses increased and could harm the performance of the SAT solver. To assist the SAT solver, we chose to encode some constraints using a programmatic approach as presented by Ganesh et al. [10]. Using this approach we were able to generate Williamson matrices of all even orders up to 42 in a feasible amount of time. Note that previously done searches for Williamson matrices assumed that the order was odd, leveraging additional symmetry in these cases.

A. Contributions

This paper makes the following contributions:

1) An improvement of our prototype MATHCHECK2, a combination of CAS and SAT solvers for finitely verifying or counterexampling math conjectures. We discuss three techniques in Section IV that dramatically improve the search capabilities of the basic MATHCHECK2 methodology. We further show how we assist the SAT solver by performing certain checks programmatically using a CAS and thereby prune away parts of the search space significantly. All three techniques can be adapted for other conjectures, beyond the ones considered in this paper. https://sites.google.com/site/uwmathcheck/.
2) Description of a novel algorithm for finding Williamson matrices of a given order (or showing that none exist). This algorithm makes use of recent theorems about the properties of compressed sequences [22] and invariants which significantly limit the number of compressed sequences to search.
3) Submission of the Hadamard matrices generated by our system (up to order 168) to the MAGMA database of Hadamard matrices. Up until now, more than 500 matrices were generated by us which are not equivalent to any matrix previously included in this database. (We keep a list that is regularly updated at https://sites.google.com/site/uwmathcheck/hadamard-conjecture.) Such matrices that are not equivalent to any previously known Hadamard matrix are very useful in practical applications of coding theory.

II. ARCHITECTURE OF MATHCHECK2

The architecture of our proposed MATHCHECK2 system is outlined in Figure 1. At its heart is a generator of combinatorial structures, 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 generator contains functions useful for translating combinatorial conditions into clauses which can be read by a SAT solver. It is possible to substitute the SAT solver with an SMT solver to simplify the encoding process, but this resulted in a too large overhead in our computations. The generator is currently optimized to deal with conjectures which concern Hadamard matrices from coding and combinatorial design theory.

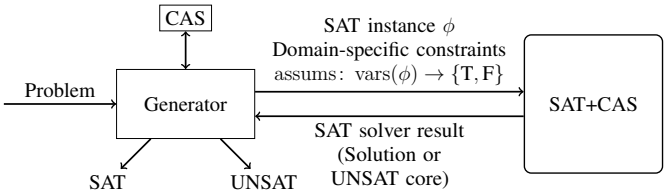

Fig. 1. High-level overview of an updated MATHCHECK2 architecture which was found useful for generating Hadamard matrices. The generator script splits the search space into many instances, each instance defined by a set of assumptions for the variables in ϕ and domain-specific constraints (which can be generated and interpreted by the CAS). Some instances are pruned away based on a previous UNSAT core result or by filtering theorems which require the usage of a CAS to apply. The SAT+CAS box contains a DPLL(CAS) style combination.

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 generator then queries the CAS (we chose MAPLE in our calculations) it is interfaced with for properties that any order n instance of the combinatorial object in question must satisfy. However, since our generator is written in Python, many CAS functionality is provided by certain modules such as NUMPY [25]; in order to avoid overhead in calling a CAS specifically, we tried to use these module functions whenever possible. 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[2] format which contains the conditions specifying the combinatorial object being searched for. These are encoded as propositional formulae 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.

The main advantage of splitting up the problem in such a way is that it easily facilitates parallelization. Using domain

[2]For more information on this format, please refer to http://www.satcompetition.org/2009/format-benchmarks2009.html

specific knowledge, we partition the search space into different classes of the same mathematical structure, and since these classes are independent of each other, a cluster of SAT solvers can search the space for each partition in parallel.

As a new feature, the generator does not include all clauses for sanity checks into the files. We rather perform the most common checks programmatically in a feedback loop inside the SAT solver. In this way, we reduce the number of needed clauses in each file, and we observed a speedup of MATHCHECK2 when searching for Hadamard matrices.

Furthermore, an *UNSAT core* (generated by SAT solvers such as MAPLESAT [17, 16]) can often be used to further prune away other similar instances. The UNSAT core of an unsatisfiable formula ϕ is a set of clauses that pithily characterizes the reason why ϕ is unsatisfiable and thus encodes an unsatisfying subspace of the search space.

III. MATHEMATICAL BACKGROUND

In this section we discuss the definitions and theorems used in this work.

A. Hadamard and Williamson matrices

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.*

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 $U \cdot H_1 \cdot V = H_2$.

One prominent class of special Hadamard matrices are those generated by so-called Williamson matrices.

Theorem 1 (cf. [26])**.** *Let $n \in \mathbb{N}$ and let A, B, C, $D \in \{\pm 1\}^{n\times n}$. 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 a Hadamard matrix of order $4n$ can be constructed (for details see [26]).

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 $[x_0, \ldots, x_{n-1}]$ 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 $i = 1, \ldots, n-1$. Therefore, if a matrix is both symmetric and circulant its first row must be of one of the two forms

$$\begin{aligned}&[x_0, x_1, x_2, \ldots, x_{(n-1)/2}, x_{(n-1)/2}, \ldots, x_2, x_1]\\ &[x_0, x_1, x_2, \ldots, x_{n/2-1}, x_{n/2}, x_{n/2-1}, \ldots, x_2, x_1]\end{aligned} \tag{1}$$

according to if n is odd or even.

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 $i = 1, \ldots, n-1$.*

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.)

B. Williamson equivalences

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

$$|\mathrm{rsum}(A)| \leq |\mathrm{rsum}(B)| \leq |\mathrm{rsum}(C)| \leq |\mathrm{rsum}(D)|,$$

where $\mathrm{rsum}(X)$ 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 $\mathrm{rsum}(X) \equiv n \pmod 4$ for $X \in \{A, B, C, D\}$. In this case, a result of Williamson [26] 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.

C. Filtering theorem

Because the search space for Hadamard matrices is so large, it is advantageous to focus on a specific construction method and use known properties of that construction type to filter the search space. We now list the definitions we need to state the main such filtering theorem we used.

Definition 3. *The **power spectral density** of the sequence A is the nonnegative real sequence*

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

where DFT_A is the discrete Fourier transform of A.

Definition 4. *The **periodic autocorrelation function** of the sequence A is the periodic function given by*

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

Definition 5 (cf. [22])**.** *Let $A = [a_0, a_1, \ldots, a_{n-1}]$ be a sequence of length $n = dm$ and set*

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

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

The following theorem is a special case of a result from [22].

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

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = \begin{cases} 4n & \text{if } s = 0 \\ 0 & \text{if } 1 \le s < \mathrm{len}(A). \end{cases} \tag{2}$$

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

$$\mathrm{PSD}_A(s) + \mathrm{PSD}_B(s) + \mathrm{PSD}_C(s) + \mathrm{PSD}_D(s) = 4n. \tag{3}$$

Furthermore, both (2) and (3) 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 (3) 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.

D. Useful lemmas

In the course of our research we discovered the following useful properties of the compressed sequences which arise in our context. These lemmas significantly reduce the number of SAT instances we need to generate.

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

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

IV. 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., $\mathrm{BV}(x \cdot y) = \mathrm{XNOR}(\mathrm{BV}(x), \mathrm{BV}(y))$.

A. 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 componentwise 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 $p_{ijk} \coloneqq \mathrm{BV}(h_{ik} \cdot h_{jk})$ 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 these values (when thought of as ± 1), 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 $\{0, 1\}$ values the two outputs store the binary representation of the sum of the inputs. (A full adder does the same thing, but consumes three inputs.)

B. Williamson autocorrelation encoding

The Williamson encoding is very similar to the general encoding but with fewer variables; we merely have the $4\left\lceil \frac{n+1}{2} \right\rceil$ 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 $\mathrm{rsum}(H_i * H_j) = 0$ for $i \ne j$ we must enforce the conditions

$$\mathrm{rsum}(A_i * A_j + B_i * B_j + C_i * C_j + D_i * D_j) = 0 \quad \text{for } i \ne j. \tag{4}$$

Like in Section IV-A this is done by defining new variables to represent the entries of the componentwise products. Also, note that because of the symmetry and circulant properties most of the conditions to enforce will be identical. In fact, it is only necessary to encode (4) for $i = 0$ and $j = 1, \ldots, \left\lceil \frac{n-1}{2} \right\rceil$ to ensure that such matrices generate a valid Hadamard matrix.

C. 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 $\sum_{k=0}^{n-1} a_k = \mathrm{rsum}(A)$. 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

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

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 [24]). 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 in ascending order, namely,

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

Any Williamson quadruple is equivalent to another quadruple whose rowsum sum-of-squares decomposition is of one of the above three types.

When using this technique it is necessary to encode constraints on the rowsum of the generating matrices, e.g., $\mathrm{rsum}(A) = 1$. This may be simply done by using a binary adder network on the variables $a_0, \ldots, a_{\lceil (n-1)/2 \rceil}$. 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.

D. 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 $\mathrm{rsum}(A) = 1$. Of those, only 12 satisfy $\mathrm{PSD}_A(s) \leq 4n$ for all $s \in \mathbb{Z}$. The calculation of $\mathrm{PSD}_A(s)$ was possible to be performed within Python using NUMPY instead of directly querying a CAS. There are also 12 possible compressions for each of B, C, and D with $\mathrm{rsum}(B) = \mathrm{rsum}(C) = 3$ and $\mathrm{rsum}(D) = 11$. 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.

E. Technique 3: UNSAT core

After using the divide-and-conquer technique one obtains a collection of instances which are almost identical. For example, the order 40 instances contained 4185 variables and only at most 184 variables differed between instances. Because of this similarity, a short reason why one instance is unsatisfiable may also apply to other instances which are similar. When this is the case, those instances can immediately be pruned away.

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.

$\mathrm{rsum}(A)$	$\mathrm{rsum}(B)$	$\mathrm{rsum}(C)$	$\mathrm{rsum}(D)$	# of Instances
1	3	3	11	6960
1	3	7	9	8424
3	5	5	9	6290

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

F. Technique 4: Programmatic SAT

There are several constraints shared among the different SAT instances which help prune away large regions of the search space. However, the more constraints we include the more the size of each file increases; the SAT solver has more clauses to handle and this causes an increase of its runtime. In the latest version of MATHCHECK2, we oursourced these checks into a DPLL(CAS) style feedback loop which generates clauses on the fly whenever it encounters that the SAT solver is searching in a space where no solution is provably possible.

Specifically, the possible compressions which were not filtered out by the generator were translated into a set of linear constraints which were passed to the SAT+CAS solver. The CAS would then use these constraints to generate learned clauses as the search progressed.

V. VERIFICATION OF THE NONEXISTENCE OF WILLIAMSON MATRICES OF ORDER 35

We searched for Williamson matrices of order 35 using the techniques described in Section IV 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 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.

Our practice was to have people that were not involved in writing the respective code verify its correctness, and to have domain experts verify the application of the theorems used. Furthermore, our confidence of the correctness of our code was strengthened by the successful discovery of Williamson-generated Hadamard matrices for all the orders $4n$ with $n < 35$. These have been determined to be valid Hadamard matrices by the computer algebra system MAGMA.

VI. Experimental Results on the Hadamard Conjecture

We checked all of the Hadamard matrices we computed for equivalence against those in MAGMA's Hadamard matrix database. In total, our methods generated more than 500 pairwise inequivalent Hadamard matrices which were also not equivalent to any matrices in this database. We submitted these to the MAGMA team and one can download these on our project website (URL in Section I-A).

Experimental Setup and Methodology: The 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 IV. 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 would allow the overhead of many SAT solver calls to be avoided.

The timings given in Figure 3 refer to the total amount of time used by MAPLESAT across all the jobs run on SHARCNET for each order and technique. The numbers in parentheses in Figure 3 denote how many MAPLESAT calls returned a result and did not time out. The jobs using the base encoding and Technique 1 which did not time out all returned SAT. All of the jobs using Technique 2 completed without timing out and most the instances were found to be UNSAT. The Technique 3 results were the same as the Technique 2 results except with fewer calls to MAPLESAT as some instances could immediately be determined to be UNSAT.

Figure 4 contains the average runtimes for the SAT instances generated by our script using a compression factor of 2 (and therefore only for even orders). For comparision purposes, MAPLESAT was run both with and without the programmatic functionality and all other techniques enabled.

Order	Base Encoding (Sec. IV-B)	Technique 1 (Sec. IV-C)	Technique 2 (Sec. IV-D)	Technique 3 (Sec. IV-E)
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)

Fig. 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.

Order	Original	Programmatic
24	0.01	0.01
26	0.09	0.08
28	0.06	0.05
30	0.48	0.28
32	0.04	0.05
34	2.69	1.51
36	0.83	0.75
38	10.62	6.08
40	1.02	1.08
42	112.51	42.21

Fig. 4. Average runtimes, in seconds, of running MAPLESAT on our SAT instances of even order for both our original and programmatic implementations. Each instance was generated using all the filtering theorems previously described and a compression factor of 2.

VII. 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. Recently, Heule et al. [12] used a SAT solver to solve the Boolean Pythagorean triples problem from Ramsey theory. Junges et al. [14] 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 [9] is a computer algebra system with specialized algorithms for polynomial systems. Ábrahám later highlights the potentials of combining symbolic computation and SMT solving in [1]. The VERIT SMT solver [4] uses the computer algebra system REDUCE [11] to support non-linear arithmetic. The LEAN theorem prover [8] combines domain-specific knowledge with SMT solvers. Combining SAT and SMT with theorem proving has been done in the automated theorem prover COQ as well [2]. The idea of using equivalences in satisfiability problems to prune the search space has also been exploited by symmetry breaking [13, 23]. SAT-based results on the Erdős discrepancy conjecture [15] inspired the previous version of MATHCHECK [28]. This version also combined SAT with computer algebra systems but specialized in graph theory and used the CAS to uncover theory lemmas as the search progressed. Work related to finding Hadamard matrices has been referenced in Section III.

VIII. Conclusions and Future Work

We have successfully presented the advantages of utilizing the power of SAT solvers in combination with domain specific knowledge and algorithms provided by computer algebra systems. Our main mathematical problem was the verification of the Hadamard conjecture for some orders by using MATHCHECK2 to search for and discover Williamson matrices.

We verified independently, as requested by D. Đoković, that there is no Hadamard Matrix of order $4 \cdot 35$ which is generated by Williamson matrices. Moreover, we discovered more than 500 Hadamard matrices that are not equivalent to any matrix in the MAGMA Hadamard database.

A future direction is to scale to Hadamard matrices of higher order. For this, we plan to refine the methods (e.g., by examining other construction types). We also want to analyze the UNSAT cores generated by Technique 3 to explain their effectiveness in certain cases, as well as exploring the usage of incremental SAT solvers [3, 20]. Finally, we plan to use MATHCHECK2 and our newly acquired knowledge to consider other combinatorial problems. There are many problems which can be expressed as a search for objects which satisfy certain autocorrelation equations (as just one example, those involving complex Golay sequences). Since the ability to work with autocorrelation is already built-in to MATHCHECK2, we should be able to execute such searches with minor modifications.

REFERENCES

[1] 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.

[2] Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in COQ for a fully automated decision procedure. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, 2011.

[3] 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.

[4] 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.

[5] Curtis Bright, Vijay Ganesh, Albert Heinle, lias Kotsireas, Saeed Nejati, and Krzysztof Czarnecki. Mathcheck2: A SAT+CAS verifier for combinatorial conjectures. In *Computer Algebra in Scientific Computing (to appear)*. Springer Berlin Heidelberg, 2016.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. *arXiv preprint arXiv:1605.00723*, 2016.

[13] Brahim Hnich, Steven D. Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint Models for the Covering Test Problem. *Constraints*, 11(2):199–219, 2006.

[14] 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.

[15] 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.

[16] 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.

[17] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. *To appear in the proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, 2016.

[18] João P Marques-Silva, Karem Sakallah, et al. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[19] 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.

[20] Alexander Nadel and Vadim Ryvchin. Efficient SAT Solving under Assumptions. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 242–255. Springer, 2012.

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

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

[23] Steve D Prestwich, Brahim Hnich, Helmut Simonis, Roberto Rossi, and S Armagan Tarim. Partial Symmetry Breaking by Local Search in the Group. *Constraints*, 17(2):148–171, 2012.

[24] Joe Riel. `nsoks`: A MAPLE script for writing n as a sum of k squares.

[25] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NUMPY array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[26] John Williamson. Hadamard's Determinant Theorem and the Sum of Four Squares. *Duke Math. J*, 11(1):65–81, 1944.

[27] Stephen Wolfram. *The* MATHEMATICA *Book, version 4*. Cambridge University Press, 1999.

[28] 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.