Noname manuscript No.
(will be inserted by the editor)

Combining SAT Solvers with Computer Algebra Systems to Verify Combinatorial Conjectures

Edward Zulkoski · Curtis Bright · Albert Heinle · Ilias Kotsireas · Krzysztof Czarnecki · Vijay Ganesh

the date of receipt and acceptance should be inserted later

Abstract We present a method and an associated system, called MATHCHECK, that embeds the functionality of a computer algebra system (CAS) within the inner loop of a conflict-driven clause-learning SAT solver. SAT+CAS systems, à la MATHCHECK, can be used as an assistant by mathematicians to either find counterexamples or finitely verify open universal conjectures on any mathematical topic (e.g., graph and number theory, algebra, geometry, etc.) supported by the underlying CAS. Such a SAT+CAS system combines the efficient search routines of modern SAT solvers, with the expressive power of CAS, thus complementing both. The key insight behind

E. Zulkoski
University of Waterloo
200 University Ave. West, Waterloo, Ontario, Canada
E-mail: ezulkosk@gsd.uwaterloo.ca

C. Bright
University of Waterloo
200 University Ave. West, Waterloo, Ontario, Canada
E-mail: cbright@uwaterloo.ca

A. Heinle
University of Waterloo
200 University Ave. West, Waterloo, Ontario, Canada
E-mail: aheinle@uwaterloo.ca

I. Kotsireas
Wilfrid Laurier University
75 University Ave. West, Waterloo, Ontario, Canada
E-mail: ikotsire@wlu.ca

K. Czarnecki
University of Waterloo
200 University Ave. West, Waterloo, Ontario, Canada
E-mail: kczarnec@gsd.uwaterloo.ca

V. Ganesh
University of Waterloo
200 University Ave. West, Waterloo, Ontario, Canada
E-mail: vganesh@uwaterloo.ca

the power of the SAT+CAS combination is that the CAS system can help cut down the search-space of the SAT solver, by providing learned clauses that encode theory-specific lemmas, as it searches for a counterexample to the input conjecture (just like the T in DPLL(T)). In addition, the combination enables a more efficient encoding of problems than a pure Boolean representation.

In this paper, we leverage the capabilities of several different CAS, namely the SAGE, MAPLE, and MAGMA systems. As case studies, we study three long-standing open mathematical conjectures, two from graph theory regarding properties of hypercubes, and one from combinatorics about Hadamard matrices. The first conjecture states that any matching of any d-dimensional hypercube can be extended to a Hamiltonian cycle; the second states that given an edge-antipodal coloring of a hypercube there always exists a monochromatic path between two antipodal vertices; the third states that Hadamard matrices exist for all orders divisible by 4. Previous results on the graph theory conjectures have shown the conjectures true up to certain low-dimensional hypercubes, and attempts to extend them have failed until now. Using our SAT+CAS system, MATHCHECK, we extend these two conjectures to higher-dimensional hypercubes. Regarding Hadamard matrices, we demonstrate the advantages of SAT+CAS by constructing Williamson matrices up to order 42 (equivalently, Hadamard up to order $4 \cdot 42 = 168$), improving the bounds up to which Williamson matrices of even order have been constructed. Prior state-of-the-art construction was only feasibly performed for odd numbers, where possible.

1 Introduction

Boolean conflict-driven clause-learning (CDCL) SAT and satisfiability modulo theories (SMT) solvers have become some of the leading tools for solving complex problems expressed as logical constraints [7]. This is particularly true in software engineering, broadly construed to include testing, verification, analysis, synthesis, and security. Modern SMT solvers such as Z3 [15], CVC4 [4], STP [25], and VERIT [9] contain efficient decision procedures for a variety of first-order theories, such as uninterpreted functions, quantified linear integer arithmetic, bitvectors, and arrays. However, even with the expressiveness of SMT, many constraints, particularly ones stemming from mathematical domains such as graph theory, topology, algebra, or number theory are non-trivial to solve using today's state-of-the-art SAT and SMT solvers.

Computer algebra systems (e.g., MAPLE [11], MATHEMATICA [69], MAGMA [8] and SAGE [59]), on the other hand, are powerful tools that have been used for decades by mathematicians to perform symbolic computation over problems in graph theory, topology, algebra, number theory, etc. However, when applied to prove or disprove a certain statement, computer algebra systems (CAS) lack the search capabilities of SAT/SMT solvers, which are a central aspect of the latter tools.

In this paper, we present a method and a prototype tool, called MATHCHECK, that combines the search capability of SAT solvers with powerful domain knowledge of CAS systems (i.e., a toolbox of algorithms to solve a broad range of mathematical problems). The tool MATHCHECK can solve problems that are too difficult or inef-

ficient to encode as SAT problems. MATHCHECK can be used by mathematicians to finitely check or find counterexamples to open conjectures. It can also be used by engineers who want to readily leverage the joint capabilities of both CAS systems and SAT solvers to model and solve problems that are otherwise too difficult with either class of tools alone.

The key concept behind MATHCHECK is that it embeds the functionality of a computer algebra system (CAS) within the inner loop of a CDCL SAT solver. Computer algebra systems contain state-of-the-art algorithms from a broad range of mathematical areas, many of which can be used as subroutines to easily encode predicates relevant both in mathematics and engineering. The users of MATHCHECK write predicates in the language of the CAS, which then interacts with the SAT solver through a controlled SAT+CAS interface. The user's goal is to finitely check or find counterexamples to a Boolean combination of predicates (somewhat akin to a quantifier-free SMT formula). The SAT solver searches for counterexamples in the domain over which the predicates are defined, and invokes the CAS to learn clauses that help cut down the search space (akin to the "T" in DPLL(T)).

In this work, we focus on constraints from the domain of combinatorics, although our approach is equally applicable to other areas of mathematics. Constraints in graph theory such as connectivity, Hamiltonicity, acyclicity, etc. are non-trivial to encode with standard solvers, and can lend themselves to many possible encodings of widely ranging performance [64]. The same holds for encoding specifically structured matrices with entries coming from a finite domain. We believe that the method described in this paper is a step in the right direction towards making SAT/SMT solvers useful to a broader class of mathematicians and engineers than before.

Most CAS's additionally support methods for computing symmetries of a group and automorphisms of graph objects, sometimes via interfacing fast graph automorphism tools such as SAUCY [14] or BLISS [37]. Symmetry breaking has been applied to SAT instances through tools such as SHATTER [1], which converts the conjunctive normal form (CNF) of the input to a graph automorphism problem that is then solved with an off-the-shelf tool such as SAUCY. We use these methods to define symmetry breaking routines for our graph theoretic case studies, which significantly reduce solving times.

While we believe that our method is probably the first such combination of SAT+CAS systems, there has been previous work in attempting to extend SAT solvers with graph reasoning [18,27,58]. These works can loosely be divided into two categories: constraint-specific extensions, and general graph encodings. As an example of the first case, efficient SAT-based solvers have been designed to ensure that synthesized graphs contain no cycles [27]. In [58], Hamiltonicity checks are reduced to *native* Boolean cardinality constraints and lazy connectivity constraints. While more efficient than standard encodings of acyclicity and Hamiltonicity constraints, these approaches lack generality. On the other hand, approaches such as in CP(Graph) [18], a constraint satisfaction problem (CSP) solver extension, encode a core set of graph operations with which complicated predicates (such as Hamiltonicity) can be expressed. *Global constraints* [18] can be tailored to handle predicate-specific optimizations. Although it can be non-trivial to efficiently encode global constraints, previous work has defined efficient procedures which enforce graph constraints, such as connectivity, incremen-

tally during search [35]. Our approach is more general than the above approaches, because CAS systems are not restricted to graph theory. One might also consider a general SMT theory-plugin for graph theory. However given the diverse array of predicates and functions within the domain, a monolithic theory-plugin (other than a CAS system) seems impractical at this time.

Prior to our work, the construction of Hadamard matrices [31] was mostly done via utilizing the power of CAS [40,41,42,49,50]. One could also use SAT solvers to directly search for Hadamard matrices, but unfortunately SAT solvers perform poorly over straightforward encodings of the Hadamard problem. For example, experiments we conducted show that it is very costly to compute Hadamard matrices of order larger than 20. However, our experiments also showed that SAT solvers scale much better than CAS when it comes to combinatorial search, while CAS can solve complex set of constraints that may be difficult or even impossible to encode as a SAT instance. Based on these observations and using the knowledge obtained from our work on the original MATHCHECK system [71], we wrote a follow-up tool (called MATHCHECK2 [10]) specifically designed to construct Hadamard matrices using the Williamson construction method [68] for circulant matrices. Using this system we provided an independent verification of a result by Đoković that there are no Williamson matrices for order 35 [49]. Furthermore, we discovered 885 matrices that were formerly not included in the comprehensive database of Hadamard matrices in the MAGMA computer algebra system (ranging with orders up to $4 \cdot 42$). We also constructed Williamson matrices for all even orders up to 42; it should be noted that classical purely CAS driven methods focused solely on odd orders since additional symmetry results are available in those cases.

Main Contributions:[1]

Analysis of a SAT+CAS Combination Method and the MATHCHECK tool. In Section 3, we present a method and tool that combines a CAS with SAT, denoted as SAT+CAS, facilitating the creation of user-defined CAS predicates. Such tools can be used by mathematicians to finitely search or find counterexamples to universal sentences in the language of the underlying CAS. MATHCHECK allows users to easily specify and solve complex combinatorial questions using the simple interface provided. The system can easily be extended to other domains, although we currently focus on problems coming from graph- and matrix-theory.

Results on Two Open Graph-Theoretic Conjectures over Hypercubes. In Section 4, we use our system to extend results on two long-standing open conjectures related to hypercubes. Conjecture 1 states that any matching of any d-dimensional hypercube can extend to a Hamiltonian cycle. Conjecture 2 states that given an edge-antipodal coloring of a hypercube, there always exists a monochromatic path between two antipodal vertices. Previous results have shown Conjecture 1 (resp. Conjecture 2) true up to $d = 4$ [22] (resp. $d = 5$ [21]); we extend these two conjectures to $d = 5$ (resp. $d = 6$). We discuss symmetry breaking optimizations, in which we learn many symmetric clauses during solving, which result in an order of magnitude performance improvement for MATHCHECK on the two case studies.

[1] All code and data is available at `https://sites.google.com/site/uwmathcheck/`.

Results Regarding Construction of Hadamard Matrices. In Section 5, we use MATHCHECK to construct new Hadamard matrices via the Williamson construction. In [10], we constructed Williamson matrices for all even orders up to 34, while we are now able to construct Williamson matrices for all even orders up to 42. We furthermore report on 309 newly identified Williamson matrices of order 40, which can be used to construct Hadamard matrices of order 160. For comparison, there is only one Hadamard matrix of order 160 currently listed in the comprehensive database of the computer algebra system MAGMA. Our constructed matrices will be made available in a future version of MAGMA (as well as on our website given above).

Performance Analysis of MATHCHECK. In Section 6, we provide detailed performance analysis of MATHCHECK in terms of how much search space reduction is achieved relative to finite brute-force search, as well as how much time is consumed by each component of the system. Improvements from symmetry breaking techniques are also discussed. We additionally compare MATHCHECK to ALLOY*, a higher-order relational logic solver built on top of a SAT solver on our two graph-theoretic case studies. We chose to compare against ALLOY* as it was 1) SAT-based; and 2) expressive enough to support our two graph theoretic case studies. Results over the two case studies favor MATHCHECK.

Verification of Results. We provide details on the techniques used to check the results of our case studies in Section 7. In addition to checking that the input formula to the SAT solver and its output are correct, we must also ensure that the learned clauses generated by the CAS follow from the input specification. We discuss the analyses we performed and certificates generated by our tool in order to check its correctness after solving.

2 Background

2.1 Graph Theory

We assume standard definitions for propositional logic, basic mathematical logic concepts such as satisfiability, and solvers (for a detailed overview, see [7]). We denote a graph $G = \langle V, E \rangle$ as a set of vertices V and edges E, where an edge e_{ij} connects the pair of vertices v_i and v_j. We only consider undirected graphs in this work. The *order* of a graph is the number of vertices it contains. For a given vertex v, we denote its neighbors – vertices that share an edge with v – as $N(v)$.

The hypercube of dimension d, denoted Q_d, consists of 2^d vertices and $2^{d-1} \cdot d$ edges, and can be constructed in the following way (see Figure 3a): label each vertex with a unique binary string of length d, and connect two vertices with an edge if and only if the Hamming distance of their labels is 1. A *matching* of a graph is a subset of its edges that mutually share no vertices. A vertex is *matched* (by a matching) if it is incident to an edge in the matching, else it is *unmatched*. A *maximal matching* M is a matching such that adding any additional edge to M violates the matching property. A *perfect matching* (resp. *imperfect matching*) M is a matching such that all (resp. not all) vertices in the graph are incident with an edge in M. A *forbidden matching* is a matching such that some unmatched vertex v exists and every $v' \in N(v)$

is matched. Intuitively, no superset of the matching can match v. Vertices in Q_d are *antipodal* if their binary strings differ in all positions (i.e., opposite "corners" of the cube). Edges e_{ij} and e_{kl} are antipodal if $\{v_i, v_k\}$ and $\{v_j, v_l\}$ are pairs of antipodal vertices. A *2-edge-coloring* of a graph is a labeling of the edges with either red or blue. A 2-edge-coloring is *edge-antipodal* if the color of every edge differs from the color of the edge antipodal to it.

A symmetry/automorphism of a graph is a permutation of its vertices that preserves edges and non-edges. The set of all automorphisms of a graph is called its automorphism group.

2.2 Hadamard Matrices

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. In other words, a matrix H is a Hadamard matrix, if $HH^t = n \cdot I_n$, where H^t denotes the transpose of H and I_n is the identity matrix of size n.

First studied by Hadamard [31], 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 for the existence of 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 [46, 52, 65], statistics [32], and aeronautics[2]. 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 [60, 51, 68]. We will utilize the Williamson construction in this paper, introduced in the next subsection.

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 $n = 4 \cdot 167 = 668$ [13]. A database with many known matrices is included in the computer algebra system MAGMA [8]. Further collections are available online [56, 57].

2.2.1 Williamson matrices

Theorem 1 (cf. [68]) *Let $n \in \mathbb{N}$ and let A, B, C, D $\in \{\pm 1\}^{n \times n}$. Further, suppose that*

[2] `http://www.jpl.nasa.gov/blog/2013/8/hadamard-matrix`

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

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

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 $[x_0, \ldots, x_{n-1}]$ 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 the form

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

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

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.) Furthermore, since they are symmetric the Hadamard matrix generated by these matrices is completely specified by the $4\left\lceil \frac{n+1}{2} \right\rceil$ variables

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

3 SAT+CAS Combination Architecture

This section describes the combination architecture of a CAS system with a SAT solver, the method underpinning the MATHCHECK tool. Figure 1 provides a schematic of MATHCHECK. The key idea behind such combinations is that the CAS system is integrated in the inner loop of a conflict-driven clause-learning SAT solver, akin to how a theory solver T is integrated into a DPLL(T) system [47]. MATHCHECK allows the user to define predicates in the language of CAS that express some mathematical conjecture. The input mathematical conjecture can be expressed as a set of *assertions* and *queries*, such that a satisfying assignment to the conjunction of the assertions and **negated** queries constitute a counterexample to the conjecture. We refer to this conjunction simply as the input formula in the remainder of the paper. First, the formula is translated into Boolean constraints that describes the set of structures (e.g.,

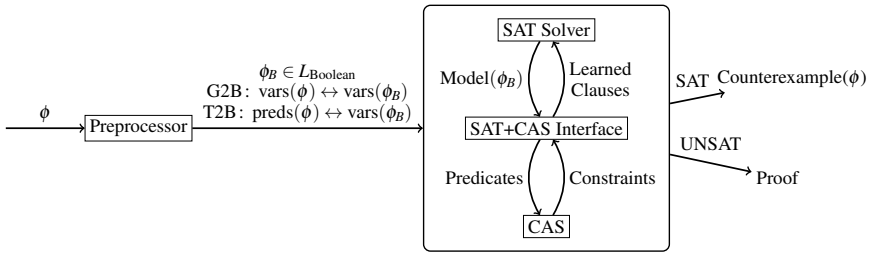


Fig. 1: High-level overview of the MATHCHECK architecture, which is similar to DPLL(T)-style SMT solvers. MATHCHECK takes as input a formula ϕ over fragments of mathematics supported by the underlying CAS system, and produces either a counterexample or a proof that no counterexample exists.

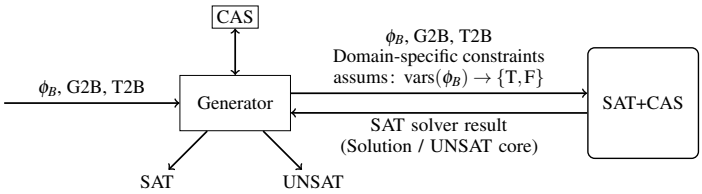


Fig. 2: High-level overview of an updated MATHCHECK architecture which was found useful for generating Hadamard matrices. Following the preprocessing step a generator script splits the search space into many instances, each instance defined by a set of assumptions for the variables in ϕ_B and domain-specific 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 as in Figure 1.

graphs or numbers) referred to in the conjecture. Second, the SAT solver enumerates these structures in an attempt to counterexample the input conjecture.

During our work on Hadamard matrices, we added an additional feature to MATHCHECK which appeared to be useful: We made use of the domain specific knowledge provided by CAS systems to partition the search space. This is done by introducing a **generator**. This generator (with the help of CAS systems) generates SAT instances which contain additional assumptions causing a solution (if existent) to lie in a specific partition. The partition in the case of the Williamson construction refers to sequences that have a certain compression, which we will introduce in subsection 5.3. Figure 2 illustrates this latest design change of MATHCHECK.

The solver, solving each generated SAT instances, routinely queries the CAS system during its search to learn clauses (akin to callback plugins in programmatic SAT solvers [26] or theory plugins in DPLL(T) [47]). Clauses thus learned can dramatically cut down the search space of the SAT solver.

Combining the solver with CAS extends each of the individual tools in the following ways. First, off-the-shelf SAT (or SMT) solvers contain efficient search techniques and decision procedures, but lack the expressiveness to easily encode many complex mathematical predicates. Even if a problem can be easily reduced to SAT/SMT, the

choice of encoding can be very important in terms of performance, which is typically non-trivial to determine, especially for non-experts on solvers. For example, Velev et al. [64] investigated 416 ways to encode Hamiltonian cycles to SAT as permutation problems to determine which encodings were the most effective. Further, such a system can take advantage of many built-in common structures in a CAS (e.g., graph families such as hypercubes), which can greatly simplify specifying structures and complex predicates. On the other side, CAS's contain many efficient functions for a broad range of mathematical properties, but often lack the robust search routines available in SAT.

3.1 MATHCHECK for Graph Theoretic Problems

The input to MATHCHECK is a tuple $\langle S, \phi \rangle$, where S is a propositional formula extended to allow predicates over graph variables. A graph variable $G = \langle G_V, G_E \rangle$ indicates the vertices and edges that can potentially occur in its instantiation, denoted G_I. A graph variable G consists of a set of $|V|$ Boolean variables (one for each vertex), and $|E|$ Boolean variables for edges. Setting an edge e_{ij} (resp. vertex v_i) to True means that e_{ij} (resp. v_i) is a part of the graph instantiation G_I. Through a slight abuse of notation, we often define a graph variable $G = Q_d$, indicating that the sets of Booleans in G_V and G_E correspond to the vertices and edges in the hypercube Q_d, respectively.

Predicates can be defined by the user, and are classified as either *SAT predicates* or *CAS predicates*. SAT predicates are blasted to propositional logic, using the mapping from graph components (i.e., vertices and edges) to Boolean variables.[3] As an example, for any graph variable G used in an input formula, we add an EdgeImpliesVertices(G) constraint, indicating that an edge cannot exist without its corresponding vertices:

$$\textbf{EdgeImpliesVertices(G):} \bigwedge \{e_{ij} \Rightarrow (v_i \wedge v_j) \mid e_{ij} \in G_E\}. \tag{2}$$

CAS predicates, defined as pieces of code in the language of the CAS, check properties of instantiated graphs and add learned clauses to the SAT solver when not satisfied. In our case, we use the SAGE CAS [59], which we essentially use as a collection of Python modules for mathematics.

Here we provide a very high-level overview, with more details in Section 3.4 below. Please refer to Figure 1, which depicts the SAT+CAS combination. Given a formula ϕ over graph variables and predicates, we conjoin the assertions with the negated queries, and preprocess it as described below. When the SAT solver finds a partial model, additional checks are performed by the CAS using the defined CAS predicates. The potential solution is either deemed a valid counterexample to the conjecture and returned to the user, or the SAT search is refined with learned clauses. Output is either SAT and a counterexample to the conjecture, or UNSAT along with a proof certificate. Although similar to the DPLL(T) approach of SMT solvers in many aspects, we note several important differences in terms extensibility, power, and flexibility: 1) rather than a monolithic theory plugin for graphs, we opt for a more *extensible* approach by incorporating the CAS, allowing new predicates (say, over numbers, geometry, algebra,

[3] For notational convenience, we often use existential quantifiers when defining constraints; these are unrolled in the implementation. We only deal with finite graphs.

etc.) to be easily defined via the CAS functionality; 2) the CAS predicates are defined as pieces of code interpreted by the CAS. This gives considerable *additional power* to the SAT+CAS combination; 3) the user may *flexibly* decide that certain predicates may be encoded directly to Boolean logic via bit-blasting, and thus take advantage of the efficiency of CDCL solvers in certain cases.

3.2 MATHCHECK for Hadamard Matrices

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

For each multiplication of two entries in a given matrix, one can store one additional variable representing the result of the multiplication. The sum of variables (when thought of as ± 1 values) can be encoded using a network of binary adders. Both of these encodings add polynomially many extra variables to a given SAT instance.

3.3 Williamson encoding

As mentioned before, we have $4\left\lceil \frac{n+1}{2} \right\rceil$ variables to solve for in the Williamson construction with circulant matrices. We enforce the conditions

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

where $*$ denotes componentwise multiplication of sequences of the same length. This is done by defining new variables to represent the entries of the componentwise products.

3.4 Architecture of MATHCHECK

The architecture of MATHCHECK specifically used for the graph problems is given in Figure 1. Figure 2 depicts the addition of the aforementioned generator, which partitions the search space with the help of domain specific knowledge provided by a CAS system.

For the graph problems, the **Preprocessor** prepares ϕ for the inner CAS-DPLL loop using standard techniques. First, we create necessary Boolean variables that correspond to graph components (vertices and edges) as described above. We replace each SAT predicate via bit-blasting with its propositional representation in situ (with respect to ϕ's overall propositional structure), such that any assignment found by the SAT solver can be encoded into graphs adhering to the SAT predicates. Finally, Tseitin-encoding and a Boolean abstraction of ϕ is performed such that CAS predicates are abstracted away by new Boolean variables; since these techniques are well-known, we do not discuss them further. This phase produces three main outputs: the CNF Boolean

abstraction ϕ_B of the SAT predicates, a mapping from graph components to Booleans G2B, and a mapping T2B from CAS predicate definitions to Boolean variables. The CAS predicates themselves are fed into the CAS.

For the search for Williamson matrices, we experimentally discovered that the best approach is to separate the search space in advance, which lead to the introduction of the generator. The generator is provided by a problem-specific script, which queries the CAS system while partitioning the search space. Optimally, the partitioning is chosen in a way such that the CAS can determine in advance, if the existence of a solution can be ruled out in some of the partitions by applying domain specific knowledge. Only SAT instances that are not ruled out by the method are forwarded to the SAT solver, which itself has again a CAS interface. The instances are specified by a set of assumptions (i.e., values for the variables in ϕ_B) and a set of domain-specific constraints (which can be interpreted by the CAS). For example, our Williamson instances contained linear equations in terms of the variables in the original formula ϕ.

The **SAT+CAS** interface acts similar to the DPLL(T) interface between the DPLL loop and theory-plugins, ensuring that partial assignments from the SAT solver satisfy theory-specific CAS predicates. After an assignment is found, literals corresponding to abstracted CAS predicates are checked. The SAT+CAS interface provides an API that allows CAS predicates to interact with the SAT solver, which modifies the API from the programmatic SAT solver LYNX [26]. In particular, the interface allows learned clauses to be added to the SAT solver based on the CAS computations. We discuss concrete examples in Section 4.

3.5 Implementation

We have prototyped our system adopting the lazy-SMT solver approach (as in [55]), specifically combining the GLUCOSE SAT solver [3] with the SAGE CAS [59]. Minor modifications to GLUCOSE were made to call out to SAGE whenever an assignment was found (of the Boolean abstraction). The SAT+CAS interface extends the existing SAT interface in SAGE. When the solver determines that the formula is UNSAT, we return two types of certificates. The first is a clausal proof of the final unsatisfiable call to the SAT solver in the DRUP-TRIM format [33], which can be checked using the DRUP-TRIM tool. Note however that, unlike a pure SAT solving run, many clauses were added due to checks of CAS predicates. In order to check the correctness of the added clauses, we return a second proof certificate, which consists of the mapping from graph components to Boolean variables, the mapping of abstracted CAS predicates to Boolean variables, and the set of clauses learned from CAS predicate invocations. We discuss how we utilize these certificates in greater detail in Section 7.

For the Hadamard matrix construction, we used structural results (outlined in subsection 5) to partition the search space. The partitioning process is assisted by computer algebra systems, and for each partition a custom SAT instance is generated. Different from our implementation in [10], we checked for certain properties which our sequences must fulfill programmatically, instead of hard-coding the properties into the SAT instances. In this case study we used a version of MAPLESAT [43] extended with a SAT+CAS programmatic interface. MAPLESAT is a modification of

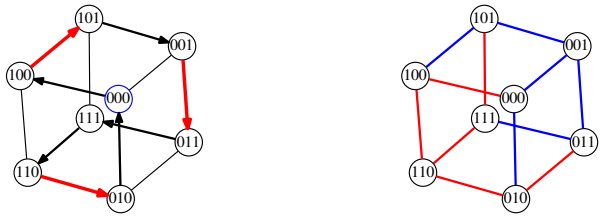


Fig. 3: (a) The red edges denote a generated matching, where the blue vertex 000 is restricted to be unmatched, as discussed in Section 4. A Hamiltonian cycle that includes the matching is indicated by the arrows. (b) An edge-antipodal 2-edge-coloring of the cube Q_3. Not a counterexample to Conjecture 2 due to the red (or blue) path from 000 to 111.

MINISAT [20] which uses a learning rate branching heuristic. From the programmatic interface it is possible to make calls to a CAS such as SAGE or C libraries such as FFTW [24] but we observed the best performance using custom written C functions instead of library functions. (The graph theory case studies may also have benefited from custom written C functions but as the CAS functions used were more complex in those cases we did not attempt to replace them.)

4 Two Results regarding Open Conjectures over Hypercubes

We use our system to prove two long-standing open conjectures up to a certain parameter (dimension) related to hypercubes, which have not been previously shown. Hypercubes have been studied for theoretical interest, due to their nice properties such as regularity and symmetry, but also for practical uses, such as in networks and parallel systems [12].

4.1 Matchings Extend to Hamiltonian Cycles

The first conjecture we look at was posed by Ruskey and Savage on matchings of hypercubes in 1993 [53]; although it has inspired multiple partial results [22,30] and extensions [23], the general statement remains open:

Conjecture 1 (Ruskey and Savage, [53]) For every dimension d, any matching of the hypercube Q_d can be extended to a Hamiltonian cycle.

Consider Figure 3a. The red edges correspond to a matching and the arrows depict a Hamiltonian cycle extending the matching. Intuitively, the conjecture states that for any d-dimensional hypercube Q_d, no matter which matching M we choose, we can find a Hamiltonian cycle of Q_d that goes through M. Our encoding searches for

matchings, and checks a sufficient subset of the full set of matchings of Q_d to ensure that the conjecture holds for a given dimension (by returning UNSAT and a proof). As we will show, constraints such as ensuring that a potential model is a matching are easily encoded with SAT predicates, while constraints such as "extending to a Hamiltonian cycle" are expressed easily as CAS predicates.

Previous results have shown this conjecture true for $d \leq 4$,[4] however the combinatorial explosion of matchings on higher dimensional hypercubes makes analysis increasingly challenging, and a general proof has been evasive. We demonstrate using our approach the first result that Conjecture 1 holds for Q_5 – the 5-dimensional hypercube. We use a conjunction of SAT predicates to generate a sufficient set of matchings of the hypercube, which are further verified by a CAS predicate to check if the matching can **not** be extended to a Hamiltonian cycle (such that a satisfying model would counterexample the conjecture).

Note that the simple approach of generating *all* matching of Q_d does not scale (see Table 1 below), and the approach would take too long, even for $d = 5$. We prove several lemmas to reduce the number of matchings analyzed. In the following, we use the graph variable $G = Q_d$, such that its vertex and edge variables correspond to the vertices and edges in Q_d.

It is straightforward to encode matching constraints as a SAT predicate. For every pair of incident edges e_1, e_2, we ensure that only one can be in the matching (i.e., at most one of the two Booleans may be True), which can be encoded as:

$$\textbf{Matching(G):} \bigwedge \{(\neg e_1 \vee \neg e_2) \mid e_1, e_2 \in G_E \wedge \text{incident?}(e_1, e_2)\}. \tag{3}$$

The number of clauses generated by the above translation is $2^d \cdot \binom{d}{2}$, which can be understood as: for each of the 2^d vertices in Q_d, ensure that each of the d incident edges to that vertex are pairwise not both in the matching.

A previous result from Fink [22] demonstrated that any perfect matching of the hypercube for $d \geq 2$ can be extended to a Hamiltonian cycle. Our search for a counterexample to Conjecture 1 should therefore only consider imperfect matchings, and even further, only maximal forbidden matchings as shown below. To encode this, we ensure that at least one vertex is not matched by any generated matching. Since all vertices are symmetric in a hypercube, we can, without loss of generality, choose a single vertex v_0 that we ensure is not matched. We encode that all edges incident to v_0 cannot be in the matching:

$$\textbf{Forbidden(G):} \bigwedge \{\neg e \mid e \in G_E \wedge \text{incident?}(v_0, e)\}. \tag{4}$$

A further key observation to reduce the matchings search space is that, if a matching M extends to a Hamiltonian cycle, then any matching M' such that $M' \subseteq M$ can also be extended to a Hamiltonian cycle.

Observation 1. All matchings can be extended to a Hamiltonian cycle if and only if all maximal forbidden matchings can be extended to a Hamiltonian cycle.

Proof. The forward direction is straightforward. For the reverse, suppose all maximal forbidden matchings can be extended to a Hamiltonian cycle. For any non-maximal

[4] We were unable to find the original source of the results for $d \leq 4$, however the result is asserted in [22]. We also verified these results using our system.

matching M, we can always greedily add edges to M to make it maximal. Call the maximal matching M'. If M' is perfect, Fink's result on perfect matchings can be applied. If not, then it is a maximal forbidden matching, and by assumption it can be extended to a Hamiltonian cycle. In either case, the resulting Hamiltonian cycle must pass through the original matching M. □

We encode this by adding the following constraints to MATHCHECK:

$$\textbf{EdgeOn(G):} \bigwedge \left\{ v \Rightarrow \bigvee \{ e \mid e \in G_E \wedge \text{incident?}(v,e) \} \mid v \in G_V \right\} \tag{5}$$

$$\textbf{Maximal(G):} \bigwedge \{ (v_i \vee v_j) \mid e_{ij} \in G_E \}. \tag{6}$$

Equation 5 states that if a vertex is on, then one of its incident edges must be in the matching. Equation 6 ensures that we only generate maximal matchings.

Proposition 1. The conjunction of Constraints 2 – 6 encode exactly the set of maximal forbidden matchings of the hypercube in which a designated vertex v_0 is prevented from being matched.

Proof. It is clear from above that any model generated will be a forbidden matching by Constraints 3 and 4 – we prove that Equations 5 and 6 ensure maximality. Suppose M is a non-maximal matching. Then there exists an edge e such that the matching does not match either of its endpoints. By Constraints 2 and 5, no edge is incident with either endpoint. But then edge e could be added without violating the matching constraints, and Constraint 6 is violated. Thus, any matching generated must be maximal. It remains to show that *all* forbidden maximal matchings that exclude v_0 can be generated. Let M be a forbidden maximal matching such that v_0 is unmatched. We construct a satisfying variable assignment over Constraints 2 – 6 which encodes M as follows:

$$\begin{aligned} &\{ e \mid e \in M \} \cup \{ \neg e \mid e \in G_E \backslash M \} \cup \\ &\{ v \mid \exists_{e \in M} \text{incident?}(v,e) \} \cup \{ \neg v \mid \nexists_{e \in M} \text{incident?}(v,e) \}. \end{aligned} \tag{7}$$

Constraint 3 holds since M is a matching, and therefore no two incident edges can both be in M. Constraint 4 holds since it is assumed that v_0 is not matched, and therefore no edge incident to v_0 can be in M. Constraints 2 and 5 hold simply because they encode the definition of a matched vertex, and the second line of Equation 7 ensures that only matched vertices are in the satisfying assignment. Constraint 6 holds since M is maximal. □

To check if each matching extends to a Hamiltonian cycle, we create the CAS predicate EXTENDSTOHAMILTONIAN (see Figure 4), which reduces the formula to an instance of the traveling salesman problem (TSP). Let M be a matching of Q_d. We create a TSP instance $\langle Q_d, W \rangle$, where Q_d is our hypercube, and W are the edge weights, such that edges in the matching (red edges in Figure 3a) have weight 1, and otherwise weight 2 (black edges).

Proposition 2. Let $|V|$ be the number of vertices in Q_d. A Hamiltonian cycle exists through M in Q_d if and only if $\text{TSP}(\langle Q_d, W \rangle) = 2|V| - |M|$.

```
1:  EXTENDSTOHAMILTONIAN()
2:      g ← s.getGraph(G)
3:      q ← CubeGraph(5)
4:      for e in q.edges() do
5:          if e in g
6:              q.setEdgeLabel(e, 1)
7:          else
8:              q.setEdgeLabel(e, 2)
9:          ⟨cycle, weight⟩ ← TSP(q)
10:         return weight == 2 · q.order() − |g|
```

```
1:  ANTIPODALMONOCHROMATIC()
2:      g ← s.getGraph(G)
3:      q ← CubeGraph(6)
4:      pairs ← getAntipodalPairs(q)
5:      for ⟨v1, v2⟩ in pairs do
6:          if shortestPath(g, v1, v2) ≠ ∅
7:              return True              ▷ a path exists
8:      return False
```

Fig. 4: CAS-defined predicates from each graph theoretic case study. In EXTENDSTOHAMILTONIAN, g corresponds to the matching found by the SAT solver. In ANTIPODALMONOCHROMATIC, g refers to the graph induced by a single color in the 2-edge-coloring.

Proof. Since Q_d has $|V|$ vertices, any Hamiltonian cycle must contain $|V|$ edges. ($\Leftarrow$) From our encoding, it is clear that $2|V| - |M|$ is the minimum weight that could possibly be outputted by TSP, and this can only be achieved by including all edges in the matching and $|V| - |M|$ edges not in the matching. ($\Rightarrow$) The Hamiltonian cycle through M has $|M|$ edges contributing a weight of 1, and $|V| - |M|$ edges contributing a weight of 2. The total weight is therefore $|M| + 2(|V| - |M|) = 2|V| - |M|$. From above, this is also the minimum weight cycle that TSP could produce. □

Finally, after each check of EXTENDSTOHAMILTONIAN that evaluates to True, we add a learned clause, based on computations performed in the predicate, to prune the search space. Since a TSP instance is solved we obtain a Hamiltonian cycle C of the cube. Clearly, any future matchings that are subsets of C can be extended to a Hamiltonian cycle; our learned constraint prevents these subsets (below h refers to the Boolean variable abstracting the CAS predicate):

$$\bigvee \{e \mid e \in Q_{dE} \backslash C\} \cup \{h\}, \text{ where } C \text{ is the learned Hamiltonian cycle.} \tag{8}$$

Our full formula for Conjecture 1 is therefore:

$$\begin{aligned} &\textbf{assert } \text{EdgeImpliesVertices}(G) \wedge \text{Matching}(G) \wedge \\ &\qquad \text{Forbidden}(G) \wedge \text{EdgeOn}(G) \wedge \text{Maximal}(G) \\ &\textbf{query } \text{ExtendsToHamiltonian}(G) \end{aligned} \tag{9}$$

4.2 Connected Antipodal Vertices in Edge-antipodal Colorings

The second conjecture deals with edge-antipodal colorings of the hypercube:

Conjecture 2 ([17]) For every dimension d, in every edge-antipodal 2-edge-coloring of Q_d, there exists a monochromatic path between two antipodal vertices.

Consider the 2-edge-coloring of the cube in Figure 3b. Although the coloring is edge-antipodal, it is not a counterexample, since there is a monochromatic (red) path

from 000 to 111, namely $\langle 000, 100, 110, 111 \rangle$. In this case, constraints such as edge-antipodal-ness are expressed with SAT predicates. We ensure that no monochromatic path exists between two antipodal vertices with a CAS predicate. Previous work has shown that the conjecture holds up to dimension 5 [21] – we show that the conjecture holds up to dimension 6.

We begin with a graph variable $G = Q_6$, and constrain it such that its instantiation corresponds to a 2-edge-coloring of the hypercube. More specifically, since there are only two colors, we associate edges in G's instantiation G_I (i.e., edges evaluated to True) with the color red, and the edges in $Q_d \backslash G_I$ with blue. An important known result is that for a given coloring, the graph induced by edges of one color is isomorphic to the other. It is therefore sufficient to check only one of the color-induced graphs for a monochromatic antipodal path.

We first ensure that any coloring generated is edge-antipodal.

$$\begin{aligned} \textbf{EdgeAntipodal(G):} \bigwedge \{ & (\neg e_1 \wedge e_2) \vee (e_1 \wedge \neg e_2) \\ & \mid e_1, e_2 \in G_E \wedge \text{isAntipodal?}(e_1, e_2) \}. \end{aligned} \tag{10}$$

Note that for every edge there is exactly one unique antipodal edge to it. Since there are $2^{d-1} \cdot d$ edges in Q_d, and therefore $2^{d-2} \cdot d$ pairs of antipodal edges, there are $2^{2^{d-2} \cdot d}$ possible 2-edge-colorings that are antipodal. We can reduce the search space by using a recent result from Feder and Suber [21]:

Theorem 1 ([21])**.** Call a labeling of Q_d *simple* if there is no square $\langle x, y, z, t \rangle$ such that e_{xy} and e_{zt} are one color, and e_{yz} and e_{tx} are the other. Every simple coloring has a pair of antipodal vertices joined by a monochromatic path.

We therefore prevent simple colorings by ensuring that such a square exists:

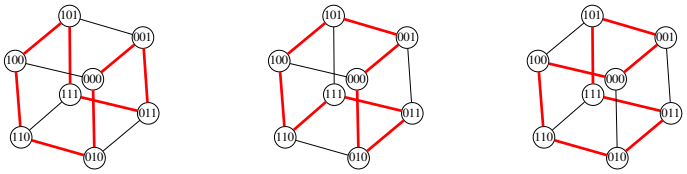
$$\begin{aligned} \textbf{NonSimple(G):} \bigvee \{ & (\neg e_{xy} \wedge e_{yz} \wedge \neg e_{zt} \wedge e_{tx}) \vee (e_{xy} \wedge \neg e_{yz} \wedge e_{zt} \wedge \neg e_{tx}) \\ & \mid e_{xy}, e_{yz}, e_{zt}, e_{tx} \in G_E \wedge \text{isSquare?}(e_{xy}, e_{yz}, e_{zt}, e_{tx}) \}. \end{aligned} \tag{11}$$

It remains to check whether an antipodal monochromatic path exists, which is checked by the CAS predicate ANTIPODALMONOCHROMATIC in Figure 4. Given a graph G, which contains only the red colored edges, we first compute the pairs of antipodal vertices in Q_d. Using the built-in shortest path algorithm of the CAS, we check whether or not any of the pairs are connected, indicating that an antipodal monochromatic path exists. In the case when predicate returns True, we learn the constraint that all future colorings should not include the found antipodal path P (m abstracts the CAS predicate):

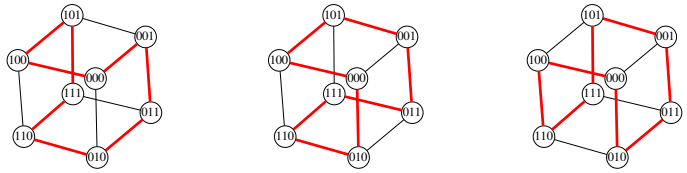
$$\bigvee \{\neg e \mid e \in P\} \cup \{m\}, \text{ where } P \text{ is the learned path.} \tag{12}$$

The full formula for Conjecture 2 is then:

$$\begin{aligned} &\textbf{assert } \text{EdgeImpliesVertices}(G) \wedge \text{EdgeAntipodal}(G) \wedge \text{NonSimple}(G) \\ &\textbf{query } \text{AntipodalMonochromatic}(G) \end{aligned} \tag{13}$$



(a) Learned Cycle (b) (1,2)(5,6) (c) (2,4)(3,5)



(d) (1,4,2)(3,5,6) (e) (1,2,4)(3,6,5) (f) (1,4)(3,6)

Fig. 5: The six unique Hamiltonian cycles of Q_3. The cycle in part (a) is the initially learned cycle, and all are others are derived from (a) using the respective permutations written in cyclic notation.

4.3 Symmetry Breaking

In each case study, a learned clause is added to the solver whenever the respective CAS predicate is not satisfied by the current model. While the learned clauses described above prune many non-satisfying models from being returned by the solver (e.g., any matching that is a subset of the Hamiltonian cycle in the first case study), many similar learned clauses can be obtained through symmetry breaking techniques, due to the highly symmetric nature of hypercubes. Our approach to symmetry breaking is loosely inspired by the work of Benhamou et. al [6], which proposes an enhanced version of clause learning where all symmetric clauses are learned during conflict analysis, rather than a single conflict clause.

Consider Figure 5. From the first case study, if we discover the Hamiltonian cycle in Figure 5a, then we learn a clause preventing any model that corresponds to a subset of the cycle. Informally, if we fix the vertices of the cube but rotate the Hamiltonian cycle to different orientations, we can learn clauses for each found cycle. Similarly, in the Antipodal case study we can learn many antipodal monochromatic paths through such rotations.

In order to compute such clauses, prior to solving, we simply compute the automorphism group of the hypercube using the CAS. In our case, the SAGE CAS interfaces the BLISS graph automorphism tool [37]. Then, whenever a Hamiltonian cycle C is learned, for every symmetry π in the automorphism group, we compute $C^\pi = \{(u^\pi, v^\pi) \mid (u,v) \in C_E\}$. It can easily be seen that C^π is also a Hamiltonian cycle

of the hypercube due to the properties of symmetries, and we only briefly outline the intuition for this. Suppose $C = \langle c_1, c_2, \ldots, c_n, c_1 \rangle$. For any edge in the cycle (c_i, c_{i+1}), we know that (c_i^π, c_{i+1}^π) is an edge in the cube since symmetries preserve the set of edges and non-edges. Finally, since π is a permutation of the vertices, $c_1^\pi, \ldots, c_n^\pi$ are unique, so C^π is a Hamiltonian cycle.

The Antipodal case study is handled analogously. We note that performing this operation over the entire automorphism group can generate many redundant clauses; we ensure that duplicates are not added. This can be optimized by considering only the proper symmetry group, which omits any symmetries from reflection.

5 Search for Williamson Generated Hadamard Matrices using SAT+CAS

We impose additional constraints on the search space of Williamson matrices to cut down on extraneous solutions and hence speed up the search.
1. Ordering. Without loss of generality, we can assume

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

where $\text{rowsum}(X)$ denotes the sum of the entries of the first (or any) row of X. Any choices 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 $\text{rowsum}(X) \equiv n \pmod 4$ for $X \in \{A, B, C, D\}$. In this case, a result of Williamson [68] says that $a_i b_i c_i d_i = -1$ for all $1 \leq i \leq (n-1)/2$. This additional symmetry structure tremendously cuts down the search space, and is the reason why mainly Williamson matrices for odd n are studied.
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).

5.1 Power spectral density

One set of properties specific for Williamson matrices is derived using the *discrete Fourier transform* from Fourier analysis, i.e., the periodic function $\text{DFT}_A(s) := \sum_{k=0}^{n-1} a_k \omega^{ks}$ for a sequence $A = [a_0, a_1, \ldots, a_{n-1}]$, where $s \in \mathbb{Z}$ and $\omega := e^{2\pi i/n}$ is a primitive nth root of unity.

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

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

The importance of the power spectral density in the search for Hadamard matrices will become apparent in Theorem 2.

5.2 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 combinatorial conjectures can be stated in terms of the PAF, allowing code used for one conjecture to be applied to 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 [39]), so that the PAF_A only needs to be computed for $s = 0, \ldots, \lfloor \frac{n-1}{2} \rfloor$; the other values can be computed through symmetry and periodicity.

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 $s = 1, \ldots, n-1$. 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

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = 0 \quad \text{for } s = 1, \ldots, \lceil \tfrac{n-1}{2} \rceil \tag{14}$$

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 (14) holds will ensure condition 3 of Theorem 1.

5.3 Compression

Because the size of the space in which a combinatorial object lies is generally exponential in 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. [50]) 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 PAF and PSD can be still applied to compressed sequences. We utilize the following theorem which is a special case of a result from [50].

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

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = \begin{cases} 4n, & s = 0 \\ 0, & 1 \leq s < \mathrm{len}(A). \end{cases} \tag{15}$$

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

$$\mathrm{PSD}_A(s) + \mathrm{PSD}_B(s) + \mathrm{PSD}_C(s) + \mathrm{PSD}_D(s) = 4n. \tag{16}$$

Furthermore, both (15) *and* (16) *hold if the sequences A, B, C, D are replaced with their compressions $A^{(d)}$, $B^{(d)}$, $C^{(d)}$, $D^{(d)}$.*

Proofs to the below stated lemmas can be found in [10].

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.*

5.4 Encoding and Search Space Pruning Techniques

Additional techniques which we used to break down the problem and assist our SAT solver are given as follows.

1. **Sum-of-squares decomposition**: Theorem 2 states that for $d = 1$ and $m = n$, we have the identity
$$\mathrm{PAF}_{A^{(1)}}(0) + \mathrm{PAF}_{B^{(1)}}(0) + \mathrm{PAF}_{C^{(1)}}(0) + \mathrm{PAF}_{D^{(1)}}(0) = 4n,$$
simplifying to $\mathrm{rowsum}(A)^2 + \mathrm{rowsum}(B)^2 + \mathrm{rowsum}(C)^2 + \mathrm{rowsum}(D)^2 = 4n$. In other words, we know that the row-sums of A, B, C and D form a sum-of-squares decomposition of $4n$.
2. **Divide-and-Conquer via compression**: We can divide the search space with respect to different possible compressions. Since many properties are stable under compression, we can also rule out certain compressions right away using filtering theorems.
3. **UNSAT core**: The instances generated by the divide-and-conquer process process are very similar (e.g., 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 similar instances. Some SAT solvers such as MAPLESAT [43] support the generation of such a reason in the form of an UNSAT core; if the reason why an instance is unsatisfiable applies to other instances those instances can immediately be pruned away.
4. **Programmatic SAT**: In our previous publication [10], we encoded all knowledge about the matrices A, B, C, and D directly in each SAT instance we created. In this updated version of MATHCHECK, we chose to not directly encode all conditions in the SAT instances, but instead let the SAT+CAS solver check some conditions programmatically. Specifically, the possible compressions which were not removed

Dimensions	Matchings	Forbidden Matchings	Maximal Forbidden Matchings
2	7	3	0
3	108	42	2
4	41,025	14,721	240
5	13,803,794,944	4,619,529,024	6,911,604

Table 1: The number of matchings of the hypercube were computed using our tool in conjunction with sharpSAT [62]: a tool for the #SAT problem. Note that the numbers for forbidden matchings are only lower bounds, since we only ensure that the *origin* vertex is unmatched. However, any unfound matchings are isomorphic to found ones.

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.

With all the mentioned techniques, we have successfully found 309 Williamson matrices of order 40. These lead to 309 new, pairwise inequivalent, Hadamard matrices of order 160.

6 Performance Analysis of MATHCHECK

For the two graph theoretic conjectures, we ran Formula 9 with $d = 5$ and Formula 13 with $d = 6$ until completion. Since both runs returned UNSAT, we conclude that both conjectures hold for these dimensions, which improves upon known results for both conjectures.

The experiments were performed on a 2.4 GHz 4-core Lenovo Thinkpad laptop with 8GB of RAM, running 64-bit Linux Mint 17. We used SAGE version 6.3 and GLUCOSE version 3.0. Formula 9 required 348,150 checks of the EXTENDSTOHAMILTONIAN predicate, thus learning an equal number of Hamiltonian cycles in the process, and took just under 8 hours. Formula 13 required 86,612 checks of the ANTIPODALMONOCHROMATIC predicate (learning the same number of monochromatic paths), requiring 1 hour 35 minutes of runtime. We note that for lower dimensional cubes solving time was far less (< 20 seconds for either case study). Adding symmetry breaking greatly reduced the solving time and number of CAS predicate checks: the first case study required 1 hour and 5 minutes, and 2441 CAS predicate checks, while the second took only 3 minutes and 122 predicate checks.

The approach we have described significantly dominates naïve brute-force approaches for both conjectures; learned clauses greatly reduce the search space and cut the number of necessary CAS predicate checks. Given the data in Table 1 and the number of calls to EXTENDSTOHAMILTONIAN for Q_5, a brute-force check of all matchings (resp. forbidden matchings) of Q_5 would require 39,649 (resp. 20) times more checks of the predicate (i.e., that many more TSP calls) than our approach. Similar comparisons can be made for the second case study.

Figure 6 depicts how much time is consumed by the SAT solver and CAS predicates in both case studies, and Figure 7 indicates the same but with symmetry breaking enabled. The lines denote the cumulative time, such that the right most point of each

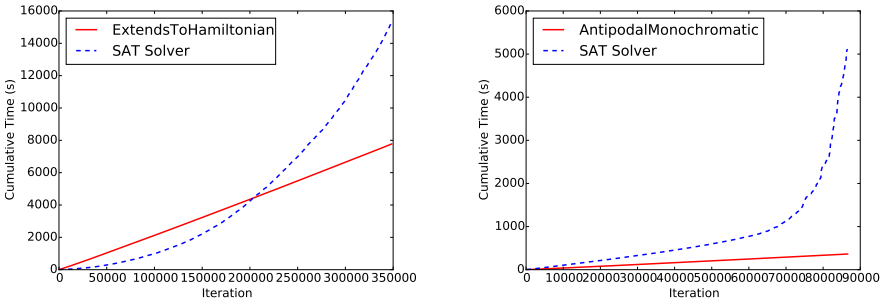


Fig. 6: Cumulative times spent in the SAT solver and CAS predicates during the two graph theory case studies. SAT solver performance degrades during solving (as indicated by the increasing slope of the line), due to the extra learned clauses and more constrained search space.

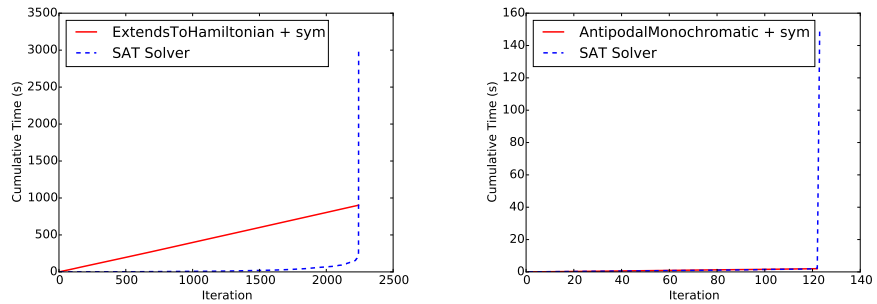


Fig. 7: Cumulative times spent in the SAT solver and CAS predicates during the two graph theory case studies with symmetry breaking. Note the differing axes from Figure 6. Interestingly, solving time is now dominated by the last UNSAT call.

line is the total time consumed by the respective system component. The near-linear lines for the CAS predicate calls indicate that each check consumed roughly the same amount of time. SAT solving ultimately dominates the runtime in both case studies, particularly due to later calls to the solver when many learned clauses have been added by CAS predicates, and the search space is highly constrained. Interestingly, the final UNSAT call to the SAT solver when symmetry breaking was enabled required significantly more time than any other calls. We did not experience this behaviour in the non-symmetry breaking experiments.

One of our motivations for this work was to allow complicated predicates to be easily expressed, so it is worth commenting on the size of the actual predicates. Since predicates were written using SAGE (which is built on top of Python), the pseudocode written in Figure 4 matches almost exactly with the actual code, with small exceptions such as computing the antipodal pairs in the second one. All other function calls correspond to built-in functions of the CAS. Learn-functions were also short, requiring less than 10 lines of code each.

To compute the new Hadamards of order 160 we ran our generation script for 300 hours on a 3.3 GHz Intel Xeon CPU E5-2667 v2 processor under 64-bit Ubuntu Linux 14.04. We used MAPLE for computing the possible sum-of-squares decompositions and NUMPY [66] for computing the power spectral densities. 171,910 total possible compressions were computed, of which 139,761 were eliminated using UNSAT cores. Of the remaining instances, 31,793 were found to be UNSAT and 356 were found to be SAT; this process took a total of 26,193 seconds. The SAT solver used was a version of MAPLESAT [43] modified to support the programmatic interface as described in Section 3.4.

6.1 Analysis of Hypercube Case Studies with Existing SAT-based Approaches

We were interested in finding previously existing SAT-based tools capable of efficiently solving and expressing the problems in our two graph theoretic case studies. Our criteria for selecting a tool were that both case studies could be succinctly expressed, and solved for at least lower-dimensional cubes with reasonable efficiency. We excluded standard SMT solvers from this evaluation due to poor support for higher-order logic. Since we are dealing with finite cases, one could in theory compare against "bitblasting" approaches, akin to how Hamiltonian cycle constraints are expressed in SAT solvers [64]. However, since our formula requires that *no* Hamiltonian cycle exists through the matching, encoding techniques from [64], which check for the *existence* of a Hamiltonian cycle, cannot be succinctly used to encode our formula.[5] We discuss other related tools in Section 8.

One such tool that met these criteria was ALLOY*, a relational finite model finder for higher-order logic [45], which extends its first-order predecessor [36]. Alloy (for first-order logic) translates input to the constraint solver KODKOD [63], which performs an efficient translation to SAT. ALLOY* extends this approach with a CEGAR loop on top of KODKOD; abstraction avoids solving higher-order constraints directly, and refinement ensures that models that do not satisfy the elided higher-order constraints are avoided in future iterations of the CEGAR loop.

For our experiments, we used default options for ALLOY*, however we changed the underlying SAT solver to GLUCOSE (which is generally considered faster than the default solver SAT4J), to match our experiments, and increased maximum memory to 4GB, which was the maximum allowed by their user interface. We also increased the maximum CEGAR loop iterations to 100,000, although this limit was never reached. For encoding Hamiltonicity, we used the monadic second-order logic encoding from [19, p. 247], with slight modifications due to ALLOY*'s concise syntax and for performance improvements. For ensuring that a monochromatic path exists between two antipodal vertices, we used a previously encoded connected constraint that is

[5] In [64], the Hamiltonian cycle detection problem is reduced to SAT by encoding it as a permutation problem, such that if a Hamiltonian cycle exists, then the permutation extracted from the model corresponds to the cycle. In order to negate this existential check (as is needed by Formula 9), one must ensure that all permutations of the vertices do not correspond to a Hamiltonian cycle. While this can be succinctly expressed a quantified Boolean formula (QBF), a substantial number of universally quantified variables must be introduced. Hamiltonian cycle detection can also be succinctly expressed in answer set programming (ASP) [44], but similar search-space explosion problems still exist.

Case Study	Translation (s)	Solving (s)	#Vars	#Clauses	MATHCHECK Time (s)
Matchings(3)	1.583	3.051	8037	27370	0.160
Matchings(4)	N/A	N/A	N/A	N/A	0.961
Matchings(5)	N/A	N/A	N/A	N/A	28800.000
Antipodal(3)	0.511	0.054	18366	65927	0.463
Antipodal(4)	4.201	0.268	203066	838251	2.211
Antipodal(5)	92.627	4.091	2221834	10682619	28.035
Antipodal(6)	N/A	N/A	N/A	N/A	3900.000

Table 2: Translation and solving times for ALLOY* on the two graph theory case studies (hypercube dimension indicated in parentheses). The number of variables and clauses produced by ALLOY*'s translation to SAT are likely the reason for long translation times. Times for MATHCHECK indicate total time; translation times were negligible compared to solving.

available with the ALLOY* implementation. We also made use of ALLOY*'s `when` construct, which improves performance of the CEGAR loop on quantified implications. All encodings are available at [70].

Table 2 displays the time taken by ALLOY* to translate its input to SAT and then perform solving. Recall that solving may require many calls to the SAT solver due to the CEGAR loop. We also included the total number of variables and clauses in the initial translation to SAT. ALLOY* produces an error during the translation phase for the Matching case study for $d = 4$, and the Antipodal case study for $d = 6$, presumably due to memory constraints and the large CNF formulas generated. Interestingly, running time is completely dominated by the translation for the Antipodal case study. The long translation time is due to the large increase in problem size when converting from relational first order logic to SAT, for these particular problems. In addition, their approach does not take advantage of predicate-specific learning opportunities, such as preventing any future matchings that are subsets of found cycles in the Matchings case study.

7 Verification of Results

Given the mathematical nature of our results, it is important to have a high degree of confidence in their correctness. This is especially true when trying to disprove a statement. In the Hadamard case, we tried to find instances of Hadamard matrices for order 40 in this paper. Once found, it is easy to verify their validity by independently checking the properties in different computer algebra systems. However, if an UNSAT is expected, as in the graph-theoretic problems we dealt with in this paper, one has to rely on the correctness of the encodings, the theory and the tools that have been used. Two main issues typically arise when verifying SAT-based analysis: 1) one must ensure that input to the SAT solver is correct, i.e., the tool which generates the DIMACS file correctly encodes the problem; 2) the computation of the SAT solver is correct. Other SAT-based analyses of mathematical problems, such as the "SAT attack" on the Erdős-discrepancy problem by Konev et al. [38] or the work of Heule et al. [34] on solving the Boolean Pythagorean triples problem, mitigate these concerns

in primarily two ways. First, in each of these works, the input DIMACS files could be generated by a very small program, which could be checked manually. In both cases, these generators were publicly released in order to be independently validated. Second, SAT solver proofs were verified with off-the-shelf clausal proof verifiers, such as DRUP-TRIM [33] or the more recent DRAT-TRIM [67].

Several issues arise when trying to take similar steps for our results. First, our tool has grown to several thousands of lines of code, and relies on multiple other software systems such as SAGE [59], GLUCOSE [3], and various Python libraries. As such, manually verifying the correctness of such a system is a non-trivial task. In order to strengthen the confidence in our results, we instead provide separate SAT generators for our two graph theory case studies, independent from the rest of our tool's codebase, that are small enough to be manually checked (approximately 100 lines of Python code each). Second, since clauses are periodically added to the solver via external calls to the CAS, merely checking the proof produced by the final UNSAT call to the SAT solver is insufficient. We must additionally ensure that clauses returned by the CAS predicates adhere to their specifications, i.e., Formulas 8 and 12. We discuss the independent checkers and certificates in more detail in the following sections. All code is available in [70].

7.1 UNSAT Proof Certificates

When MATHCHECK returns UNSAT, two types of proof certificates are produced. The first is a DRUP-TRIM certificate [33] from the final [unsatisfiable] call to the SAT solver. This is then checked with DRUP-TRIM to verify the correctness of the SAT solver's resolution proof; since this approach is commonly used we do not elaborate further. The proof for our Matchings case study was 927 MB in size, and for the Antipodal case study it was 1.4 GB (for the highest dimension cubes checked). In both of our case studies, DRUP-TRIM verified the proofs produced by the SAT solver. We also verified the results for lower dimensional cubes.

The second certificate is used to check the clauses produced by the CAS. The certificate is a triple $\langle M,P,C\rangle$, where M is a bijection between graph components (i.e., vertices and edges) and DIMACS variables, P is a similar mapping from abstracted CAS predicates and their corresponding DIMACS variables, and C is the set of learned clauses produced by the CAS predicates. We additionally annotate which CAS predicate produced each clause. The purpose of this certificate is to verify that the learned clauses produced by the CAS predicates adhere to their specification. This involves creating specialized checkers for each predicate. For example, consider a certificate $\langle M,P,C\rangle$ produced by the Matchings case study. It may be useful to refer to Formula 8. For a given learned clause, we first ensure that all literals occur positively, and then lift all DIMACS variables to their associated graph components/CAS predicates (e.g., the abstraction h of EXTENDSTOHAMILTONIAN) using M and P. We ensure that, for example, h exists in the learned clause, and that all remaining variables correspond to edges in the graph (as opposed to vertices). Finally, we check that the set of edges *not* represented in the clause correspond to a Hamiltonian cycle of Q_d. We repeat this process for every learned clause produced during solving.

7.2 Correctness of Specification

As discussed, ensuring the correctness of a large system is non-trivial, and testing that the tool correctly encodes the problem to SAT may not be sufficient. For our two graph theory case studies, we opted to create separate DIMACS generators that are much more concise than MATHCHECK's code base (approximately 100 lines of code each). These generators however only directly generate clauses related to the SAT predicates, and rely upon the certificate produced by MATHCHECK (which is also checked) to add the clauses generated by the CAS predicates (e.g., clauses associated with learned Hamiltonian cycles). One additional complication is that since these learned clauses adhere to the mapping between graph components and DIMACS used when MATHCHECK solved the formula, we must use the same mapping when generating DIMACS. We therefore ensure that the graph components used in our generators correctly adhere to the mapping specified in the first field M of the certificate, as discussed previously. Finally, before adding the learned clauses to the DIMACS file, we check that they correspond to Formulas 8 and 12, using specialized checkers as described previously. In both case studies, the generated SAT formulas were unsatisfiable, as expected. We again verified these results with DRUP-TRIM.

7.3 Verification of Discovered Hadamard Matrices

The generator produces SAT instances for each partition of the search space with respect to the possible compressions of fixed length sequences, as introduced in section 5.3. When a solver can find a solution to one of these instances, it produces a certificate in form of the encoded first lines of the found circulant matrices A, B, C and D in the Williamson construction. A custom program then translates these certificates into a Hadamard matrix using Theorem 1. The same program verifies the pairwise orthogonality of the rows of the produced matrix. As an independent verification, we used the computer algebra system MAGMA, which has a mechanism implemented to check if a matrix is in fact a Hadamard matrix.

The independent check if the found matrices are coming from different equivalence classes (cf. section 2.2) is also performed using the equivalence testing functionality in MAGMA.

All these separate checks contribute to our confidence that our construction methods are correct. However, besides our independent code review and testing practices, the generator script and specifically its CAS interface has not been formally verified. This will be subject of future work.

7.4 Further Threats to Correctness

While we strive to ensure correctness of as much of our tool as possible, since it has not been formally verified, we do make some assumptions regarding correctness. Specifically, we do not check that communication between the SAT solver and CAS is correct, in the sense that the mapping between graph components and DIMACS

variables remains constant. Second, we base our checks on the assumption that the human-derived proofs from Section 4 are correct. Ideally, these proofs would be verified with a theorem prover such as COQ [61] or ISABELLE [48]. Nonetheless, we believe that our current approach gives a high-degree of confidence in the correctness of our results.

8 Related Work

As already noted, our approach of combining a CAS system within the inner-loop of a SAT solver most closely resembles and is inspired by DPLL(T) [47]. There are also similarities with the idea of programmatic SAT solver LYNX [26], which is an instance-specific version of DPLL(T). Also, our tool MATHCHECK is inspired by the recent SAT-based results on the Erdős discrepancy conjecture [38]. Other works [18, 27,58] have extended solvers to handle graph constraints, as discussed in Section 1, by either creating solvers for specific graph predicates [27,58], or by defining a core set of constraints with which to build complex predicates [18]. Our approach contains positive aspects from both: state-of-the-art algorithms from the CAS can be used to define new predicates easily, and the methodology is general, in that new predicates can be defined using the CAS. A recent solver called MONOSAT is capable of efficiently solving problems involving monotonic theories [5]; in particular it supports many graph properties such as shortest path, connectedness, minimum spanning tree, etc. An efficient encoding for the edge-antipodal colorings conjecture may be possible using their approach, however the Ruskey-Savage conjecture violates the monotonic theory requirement. ALLOY* [45] is capable of solving many bounded higher-order relational logic specifications, and can therefore support the types of problems addressed in our case studies. We showed in Section 6.1 that the encodings of the two graph theory case studies do not seem to scale in ALLOY*, partly due to the time needed to translate the problem to the large SAT formulas generated during solving.

Several tools have combined a CAS with SMT solvers for various purposes, mainly focusing on the non-linear arithmetic algorithms provided by many CAS's. For example, the VERIT SMT solver [9] also uses functionality of the REDUCE CAS[6] for non-linear arithmetic support. Our work is more in the spirit of DPLL(T), rather than modifying the decision procedure for a single theory.

Symmetry breaking has been a widely studied topic in the context of SAT [54,6, 1], constraint solving [28,29,63], and more recently SMT [16,2]. Symmetry breaking approaches are either static – constraints are added before solving to prevent isomorphic models, as in [63], or dynamic – symmetries are detected during search and appropriate clauses are added, as in [29,6]. Our approach is most inspired by [6] – rather than learning a single learned clause from an unsatisfied CAS predicate, many are learned that correspond to graphs isomorphic to the one found (e.g., the Hamiltonian cycle).

[6] `http://www.reduce-algebra.com/index.htm`

9 Conclusions and Future Work

In this paper, we present MATHCHECK, a combination of a CAS in the inner-loop of a conflict-driven clause-learning SAT solver, and we show that this combination allows for highly expressive predicates that are otherwise non-trivial/infeasible to encode as purely Boolean formulas. Our approach combines the well-known domain-specific abilities of CAS with the search capabilities of SAT solvers thus enabling us to both construct Hadamard matrices of high orders and finitely verify two long-standing open mathematical conjectures over hypercubes up to to particular dimension, not feasible by either kind of tool alone. We further discussed how our system greatly dominates naïve brute-force search techniques for the case studies. Known symmetry breaking techniques further drastically reduced solving times. We stress that the approach is not limited to the domain of combinatorics. In fact, the ideas behind MATHCHECK can be applied to any conjecture, which can be potentially disproved by finding an object in a finite domain. We intend to extend our work to other branches of mathematics supported by CAS's, such as number theory. Another direction we plan to investigate is integration with a proof-producing SMT solver, such as VERIT. In addition to taking advantage of the extra power of an SMT solver, the integration with VERIT will allow us to more easily produce proof certificates.

References

1. Fadi A Aloul, Igor L Markov, and Karem A Sakallah. SHATTER: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th annual Design Automation Conference*, pages 836–839. ACM, 2003.
2. Carlos Areces, David Déharbe, Pascal Fontaine, and Ezequiel Orbe. SYMT: finding symmetries in SMT formulas. In *SMT Workshop 2013 11th International Workshop on Satisfiability Modulo Theories*, 2013.
3. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.
4. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011.
5. Sam Bayless, Noah Bayless, Holger H Hoos, and Alan J Hu. SAT modulo monotonic theories. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
6. Belaïd Benhamou, Tarek Nabhani, Richard Ostrowski, and Mohamed Réda Saïdi. Enhancing clause learning by symmetry in SAT solvers. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 329–335. IEEE, 2010.
7. Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
8. Wieb Bosma, John Cannon, and Catherine Playoust. The MAGMA algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265, 1997.
9. Thomas Bouton, Diego Caminha B De Oliveira, David Déharbe, and Pascal Fontaine. VERIT: an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *LNCS*, pages 151–156. Springer Berlin Heidelberg, 2009.
10. 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.
11. 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.

12. Y-Chuang Chen and Kun-Lung Li. Matchings extend to perfect matchings on hypercube networks. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1. Citeseer, 2010.
13. 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.
14. Paul T Darga, Karem A Sakallah, and Igor L Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th annual Design Automation Conference*, pages 149–154. ACM, 2008.
15. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
16. David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Automated Deduction–CADE-23*, pages 222–236. Springer, 2011.
17. S. Devos, M., Norine. Edge-antipodal Colorings of Cubes. Open Problems Garden., 2008.
18. Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming-CP 2005*, pages 211–225. Springer, 2005.
19. Rodney G Downey and Michael R Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.
20. Niklas Een and Niklas Sörensson. MINISAT: A SAT solver with conflict-clause minimization. *Sat*, 5:8th, 2005.
21. Tomás Feder and Carlos Subi. On hypercube labellings and antipodal monochromatic paths. *Discrete Applied Mathematics*, 161(10):1421–1426, 2013.
22. Jiří Fink. Perfect matchings extend to Hamilton cycles in hypercubes. *Journal of Combinatorial Theory, Series B*, 97(6):1074–1076, 2007.
23. Jiří Fink. Connectivity of matching graph of hypercube. *SIAM Journal on Discrete Mathematics*, 23(2):1100–1109, 2009.
24. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".
25. Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.
26. 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.
27. Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer, 2014.
28. Ian P Gent, Karen E Petrie, and Jean-Francois Puget. Symmetry in Constraint Programming. *Handbook of Constraint Programming*, pages 329–376, 2006.
29. Ian P Gent and Barbara Smith. *Symmetry breaking during search in constraint programming*. Citeseer, 1999.
30. Petr Gregor. Perfect matchings extending on subcubes to Hamiltonian cycles of hypercubes. *Discrete Mathematics*, 309(6):1711–1713, 2009.
31. Jacques Hadamard. Résolution d'une question relative aux déterminants. *Bull. Sci. Math.*, 17(1):240–246, 1893.
32. A Hedayat and WD Wallis. Hadamard matrices and their applications. *The Annals of Statistics*, 6(6):1184–1238, 1978.
33. Marijn JH Heule, WA Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 181–188. IEEE, 2013.
34. 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.
35. Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.
36. Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
37. Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007.

38. Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *SAT*, 2014.
39. Ilias S Kotsireas. Algorithms and Metaheuristics for Combinatorial Matrices. In *Handbook of Combinatorial Optimization*, pages 283–309. Springer New York, 2013.
40. Ilias S. Kotsireas, Christos Koukouvinos, and Jennifer Seberry. Hadamard ideals and Hadamard matrices with circulant core. 2006.
41. 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.
42. Christos Koukouvinos and Stratis Kounias. Hadamard matrices of the Williamson type of order $4 \cdot m$, $m = p \cdot q$ an exhaustive search for $m = 33$. *Discrete mathematics*, 68(1):45–57, 1988.
43. Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 3434–3440. AAAI Press, 2016.
44. Vladimir Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.
45. Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. ALLOY*: A general-purpose higher-order relational constraint solver. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 609–619, 2015.
46. David E Muller. Application of Boolean Algebra to Switching Circuit Design and to Error Detection. *Electronic Computers, Transactions of the IRE Professional Group on Electronic Computers*, EC-3(3):6–12, 1954.
47. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2004.
48. Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. ISABELLE/HOL*: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
49. Dragomir Ž Đoković. Williamson matrices of order $4n$ for $n = 33, 35, 39$. *Discrete mathematics*, 115(1):267–271, 1993.
50. Dragomir Ž Đoković and Ilias S Kotsireas. Compression of periodic complementary sequences and applications. *Designs, Codes and Cryptography*, 74(2):365–377, 2015.
51. Raymond EAC Paley. On Orthogonal Matrices. *J. Math. Phys.*, 12(1):311–320, 1933.
52. 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.
53. Frank Ruskey and Carla Savage. Hamilton cycles that extend transposition matchings in Cayley graphs of S_n. *SIAM Journal on Discrete Mathematics*, 6(1):152–166, 1993.
54. Karem A Sakallah. Symmetry and satisfiability. *Handbook of Satisfiability*, 185:289–338, 2009.
55. Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.
56. Jennifer Seberry. Library of Williamson Matrices. `http://www.uow.edu.au/~jennie/WILLIAMSON/williamson.html`.
57. Neil Sloane. Library of Hadamard Matrices. `http://neilsloane.com/hadamard/`.
58. Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native Boolean cardinality handling for the Hamiltonian cycle problem. In *Logics in Artificial Intelligence*, pages 684–693. Springer, 2014.
59. W. A. Stein et al. Sage Mathematics Software (Version 6.3), 2010.
60. James Joseph Sylvester. Thoughts on inverse orthogonal matrices, simultaneous sign successions, 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.
61. The COQ development team. *The* COQ *proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
62. Marc Thurley. SHARPSAT–counting models with advanced component caching and implicit BCP. In *Theory and Applications of Satisfiability Testing–SAT 2006*, pages 424–429. Springer, 2006.
63. Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.
64. Miroslav N Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In *SARA*, 2009.
65. Joseph L Walsh. A Closed Set of Normal Orthogonal Functions. *American Journal of Mathematics*, 45(1):5–24, 1923.

66. 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.
67. Nathan Wetzler, Marijn JH Heule, and Warren A Hunt Jr. DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer, 2014.
68. John Williamson. Hadamard's Determinant Theorem and the Sum of Four Squares. *Duke Math. J*, 11(1):65–81, 1944.
69. Stephen Wolfram. *The* MATHEMATICA *Book, version 4*. Cambridge University Press, 1999.
70. Ed Zulkoski and Vijay Ganesh. SageSAT, 2015. `https://bitbucket.org/ezulkosk/sagesat`.
71. Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki. MATHCHECK: A math assistant based on a combination of computer algebra systems and SAT solvers. In *International Conference on Automated Deduction*, Berlin, Germany, 08/2015 2015. Springer, Springer.