MathCheck: A Math Assistant via a Combination of Computer Algebra Systems and SAT Solvers

Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki

University of Waterloo, Ontario, Canada

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, a la MathCheck, can be used as an assistant by mathematicians to either counterexample or finitely verify open universal conjectures on any mathematical topic (e.g., graph and number theory, algebra, geometry, etc.) supported by the underlying CAS system. 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 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 graph-theoretic capabilities of an open-source CAS, called SAGE. As case studies, we look at two long-standing open mathematical conjectures from graph theory regarding properties of hypercubes: the first conjecture states that any matching of any d-dimensional hypercube can be extended to a Hamiltonian cycle; and the second states that given an edge-antipodal coloring of a hypercube, there always exists a monochromatic path between two antipodal vertices. Previous results 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. We provide detailed performance analysis and show an exponential reduction in search space via the SAT+CAS combination relative to finite brute-force search.

1 Introduction

Boolean conflict-driven clause-learning (CDCL) SAT and SAT-Modulo Theories (SMT) solvers have become some of the leading tools for solving complex problems expressed as logical constraints [3]. This is particularly true in software engineering, broadly construed to include testing, verification, analysis, synthesis, and security. Modern SMT solvers such as Z3 [6], CVC4 [2], STP [12], and

VeriT [4] 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, Mathematica, and SAGE), 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, computer algebra systems (CAS) lack the search capabilities of SAT/SMT solvers.

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 inefficient to encode as SAT problems. MathCheck can be used by mathematicians to finitely check or counterexample 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. By imposing restrictions on the CAS predicates, we ensure correctness (i.e. soundness) of this SAT+CAS combination. 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 cutdown the search space (akin to the "T" in DPLL(T)).

In this work, we focus on constraints from the domain of graph theory, although our approach is equally applicable to other areas of mathematics. Constraints such as connectivity, Hamiltonicity, acyclicity, etc. are non-trivial to encode with standard solvers [25]. 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.

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 [8,14,22]. 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 [14]. In [22], 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) [8], 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* [8] 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, incrementally during search [17]. 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.

Main Contributions: [1]

Contribution I: 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 counterexample universal sentences in the language of the underlying CAS. The current version of our tool, MathCheck, allows users to easily specify and solve complex graph-theoretic questions using the simple interface provided. Although our current focus is predicates based in graph theory, the system is easily extended to other domains.

Contribution II: 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 $d = 4$ [10](resp. $d = 5$ [9]); we extend these two conjectures to $d = 5$ (resp. $d = 6$).

Contribution III: Performance Analysis of MathCheck. In Section 5, 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.

2 Background

We assume standard definitions for propositional logic, basic mathematical logic concepts such as satisfiability, and solvers. 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

[1] All code+data is available at `https://bitbucket.org/ezulkosk/sagesat`.

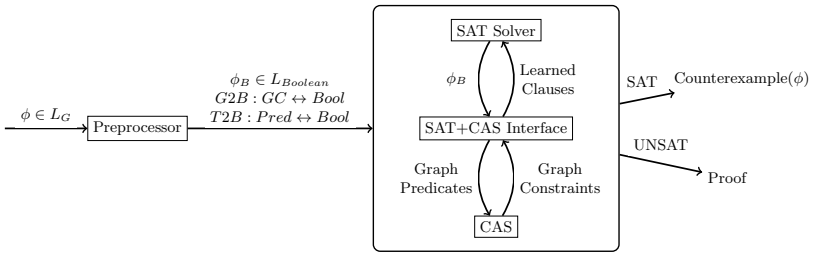

Fig. 1: High-level overview of the MathCheck architecture, which is similar to DPLL(T)-style SMT solvers. MathCheck takes as input a formula over fragments of mathematics supported by the underlying CAS system, and produces either a counterexample or a proof that none exists.

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. Inituitively, 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.

3 Contribution I: 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 [19]. The grammar of the input language of MathCheck is sketched in Figure 2. MathCheck allows the user to define predicates in the language of CAS that express some mathematical conjecture. The input mathematical conjecture is 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 a Boolean constraint that describes the set of structures (e.g., graphs or numbers) referred to in the conjecture. Second, the SAT solver enumerates these structures in an attempt to counterexample the input conjecture. The solver routinely queries the CAS system during its search (given that the CAS system is integrated into its inner loop) to learn clauses (akin to callback plugins in programmatic SAT solvers [13] or theory plugins in DPLL(T) [19]). Clauses thus learned can dramatically cutdown 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. [25] 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.

Here we provide a very high-level overview, with more details in Section 3.2 below. Please refer to Figure 1, which depicts the SAT+CAS combination. Given a formula over graph variables in the language of MathCheck (refer to Section 3.1), 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 "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 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, etc.) to be easily defined via the CAS functionality; 2) the CAS predicates are essentially defined using Python 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.

3.1 Input Language of MathCheck

The input to MathCheck is a tuple $\langle S, \phi \rangle$, where S is a set of graph variables and ϕ is a formula over S as defined by the abbreviated grammar in Figure 2. A

$$
\begin{array}{lll}
\phi & ::= & (\textbf{assert}\ \psi \mid \textbf{query}\ \psi)^+ \\
\psi & ::= & \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \mathit{Atom} \\
\mathit{Atom} & ::= & \mathit{SAT\text{-}Predicate} \mid \mathit{CAS\text{-}Predicate} \\
\mathit{SAT\text{-}Predicate} & ::= & \mathit{id}\ \text{`(`}\ \mathit{GraphVar}^+\ \text{`)'} \\
\mathit{CAS\text{-}Predicate} & ::= & \mathit{id}\ \text{`(`}\ \mathit{GraphVar}^+\ \text{`)'} \\
\mathit{GraphVar} & ::= & \textbf{graph}\ \mathit{Id}(\text{``}\texttt{Set(VertexVariables), Set(EdgeVariables)}\text{''})
\end{array}
$$

Fig. 2: Grammar L_G of MathCheck's Input Language.

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 is essentially 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.

L_G is essentially defined as propositional logic, extended to allow predicates over graph variables (as in Figure 2). 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.[2] 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:

$$\textbf{EdgeImpliesVertices(G):} \bigwedge \{e_{ij} \Rightarrow (v_i \wedge v_j) \mid e_{ij} \in G_E\}. \tag{1}$$

CAS predicates, which are essentially Python code interpreted by the CAS, check properties of instantiated (non-variable) graphs and are defined as pieces of code in the language of the CAS. In our case, we use the SAGE CAS [23], which for now can be thought of as a collection of Python modules for mathematics.

3.2 Architecture of MathCheck

The architecture of MathCheck is given in Figure 1. 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

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

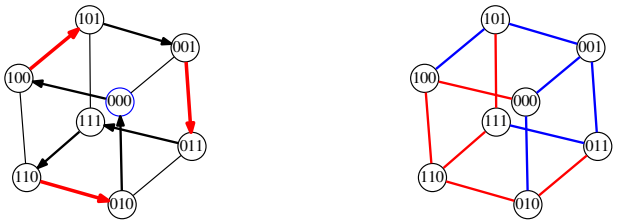

Fig. 3: (a) The red edges denote a generated matching, where the blue vertex 000 is restricted to be unmatched, as discussed in Section 4. A Hamiltonian cycle that includes the matching is indicated by the arrows. (b) An edge-antipodal 2-edge-coloring of the cube Q_3. Not a counterexample to Conjecture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

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. 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 [13].

3.3 Implementation

We have prototyped our system adopting the lazy-SMT solver approach (as in [21]), specifically combining the Glucose SAT solver [1] with the SAGE CAS [23]. 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. We further performed extensive checks on our results, including verifying the SAT solver's resolution proofs using DRUP-trim [16] as well as checking the learned clauses produced by CAS predicates, however do not elaborate now due to space constraints.

4 Contribution II: 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. 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 [5].

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 [20]; although it has inspired multiple partial results [10,15] and extensions [11], the general statement remains open:

Conjecture 1 (Ruskey and Savage, [20]). 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 hold 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 $d \leq 4$,[3] 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:

$$\textbf{Matching(G):} \bigwedge \{(\neg e_1 \vee \neg e_2) \mid e_1, e_2 \in G_E \wedge isIncident?(e_1, e_2)\}. \quad (2)$$

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 [10] 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.

[3] We were unable to find the original source of the results for $d \leq 4$, however the result is asserted in [10]. We also verified these results using our system.

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:

$$\textbf{Forbidden(G):} \bigwedge \{\neg e \mid e \in G_E \wedge isIncident?(v_0, e)\}. \tag{3}$$

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 matching M, we can always greedily add edges to M to make it maximal. Call the maximized 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:

$$\begin{aligned}\textbf{EdgeOn(G):} \bigwedge \{v \Rightarrow \exists_{e \in X}\ e | v \in G_V\},\\ s.t.\ X = \{e | e \in G_E \wedge isIncident?(v, e)\}\end{aligned} \tag{4}$$

$$\textbf{Maximal(G):} \bigwedge \{(v_i \vee v_j) \mid e_{ij} \in G_E\}. \tag{5}$$

Equation 4 states that if a vertex is on, then one of its incident edges must be in the matching. Equation 5 ensures that we only generate maximal matchings.

Proposition 1. The conjunction of Constraints 1 – 5 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 2 and 3 – we prove that Equations 4 and 5 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 1 and 4, no edge is incident with either endpoint. But then edge e could be added without violating the matching constraints, and Constraint 5 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 an forbidden maximal matching such that v_0 is unmatched. We construct a satisfying variable assignment over Constraints 1 – 5 which encodes M as follows:

$$\begin{aligned}&\{e \mid e \in M\} \cup \{\neg e \mid e \in G_E \backslash M\} \cup \\ &\{v \mid \exists_{e \in M}\ isIncident?(v, e)\} \cup \{\neg v \mid \nexists_{e \in M}\ isIncident?(v, e)\}.\end{aligned} \tag{6}$$

```
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:         if weight == 2 · q.order() − |g|
11:             return True
12:         else
13:             return False
```

```
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 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.

Constraint 2 holds since M is a matching, and therefore no two incident edges can both be in M. Constraint 3 holds since it is assumed that v_0 is not matched, and therefore no edge incident to v_0 can be in M. Constraints 1 and 4 hold simply because they encode the definition of a matched vertex, and the second line of Equation 6 ensures that only matched vertices are in the satisfying assignment. Constraint 5 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. A Hamiltonian cycle exists through M in Q_d if and only if $TSP(\langle Q_d, W\rangle) = 2 * |V| - |M|$, where $|V|$ is the number of vertices in Q_d.

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):

$$\bigvee\{e \mid e \in Q_{dE} \backslash C\} \cup \{h\}, \text{ where C is the learned Hamiltonian cycle.} \tag{7}$$

Our full formula for Conjecture 1 is therefore:

$$\begin{aligned} \textbf{assert } & EdgeImpliesVertices(G) \wedge Matching(G) \wedge \\ & Forbidden(G) \wedge EdgeOn(G) \wedge Maximal(G) \\ \textbf{query } & ExtendsToHamiltonian(G) \end{aligned} \tag{8}$$

4.2 Connected Antipodal Vertices in Edge-antipodal Colorings

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

Conjecture 2 ([7]). 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 [9] – 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.

$$\begin{aligned} \textbf{EdgeAntipodal(G):} \bigwedge\{ & (\neg e_1 \wedge e_2) \vee (e_1 \wedge \neg e_2) \\ & \mid e_1, e_2 \in G_E \wedge isAntipodal?(e_1, e_2)\}. \end{aligned} \tag{9}$$

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 [9]:

Theorem 1 ([9])**.** 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.

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 [24]: 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.

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

$$\textbf{Nonsimple(G):} \bigvee \{(\neg e_{xy} \wedge e_{yz} \wedge \neg e_{zt} \wedge e_{tx}) \vee (e_{xy} \wedge \neg e_{yz} \wedge e_{zt} \wedge \neg e_{tx}) \mid e_{xy}, e_{yz}, e_{zt}, e_{tx} \in G_E \wedge isSquare?(e_{xy}, e_{yz}, e_{zt}, e_{tx})\}. \tag{10}$$

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):

$$\bigvee \{\neg e \mid e \in P\} \cup \{m\}, \text{ where P is the learned path.} \tag{11}$$

The full formula for Conjecture 2 is then:

$$\begin{aligned} &\textbf{assert } EdgeImpliesVertices(G) \wedge EdgeAntipodal(G) \wedge NonSimple(G) \\ &\textbf{query } AntipodalMonochromatic(G) \end{aligned} \tag{12}$$

5 Contribution III: Performance Analysis of MathCheck

We ran Formula 8 with $d = 5$ and Formula 12 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.

All 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 8 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 12 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

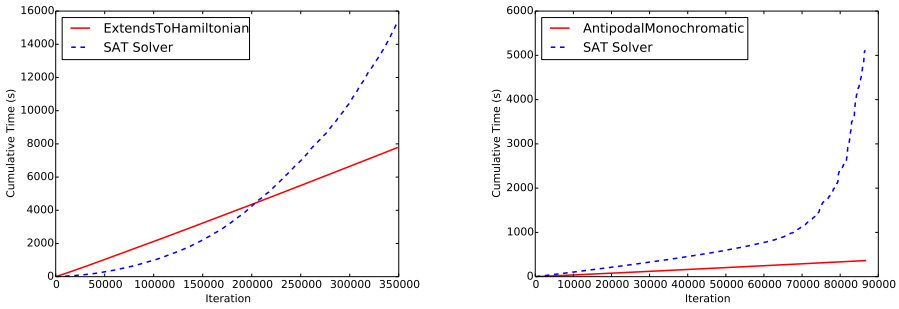

Fig. 5: Cumulative times spent in the SAT solver and CAS predicates during the two case studies. SAT solver performance degrades during solving (as indicated by the increasing slope of the line), due to the extra learned clauses and more constrained search space.

for either case study). We find it unlikely that this approach can be used for higher-dimensions, without further lemmas to reduce the search space.

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 5 depicts how much time is consumed by the SAT solver and CAS predicates in both case studies. The lines denote the cumulative time, such that the right most point of each 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. This suggests several optimizations as future work. For example, if SAT solver calls are rapidly requiring more time (e.g., around iteration 75,000 in the second plot of Figure 5), more sophisticated CAS routines can be used to produce more learned clauses per call (such as by learning constraints corresponding to all cycles *isomorphic* to the found one in case study 1), in order to reduce the number of necessary SAT calls. Alternatively, one can attempt to condense the learned clauses, which are generated independently of each other, into a more compact Boolean representation.

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.

6 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 the DPLL(T) [19]. There are also similarities with the idea of programmatic SAT solver Lynx [13], which is an instance-specific version of DPLL(T). Also, our tool MathCheck is inspired by the recent SAT-based results on the Erdős discrepency conjecture [18]. Other works [8,14,22] have extended solvers to handle graph constraints, as discussed in Section 1, by either creating solvers for specific graph predicates [14,22], or by defining a core set of constraints with which to build complex predicates [8]. 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. 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 [4] also uses functionality of the REDUCE CAS[4] 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.

7 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 verify 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. We stress that the approach is not limited to this domain, and 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. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

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

2. Clark Barrett, ChristopherL. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*. 2011.
3. Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, February 2009.
4. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In *CADE*, 2009.
5. Y-Chuang Chen and Kun-Lung Li. Matchings extend to perfect matchings on hypercube networks. In *IMECS*, volume 1. Citeseer, 2010.
6. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer, 2008.
7. S. Devos, M., Norine. Edge-antipodal Colorings of Cubes. `http://garden.irmacs.sfu.ca/?q=op/edge_antipodal_colorings_of_cubes`.
8. Grégoire Dooms, Yves Deville, and Pierre Dupont. CP (graph): Introducing a graph computation domain in constraint programming. In *CP*. 2005.
9. Tomás Feder and Carlos Subi. On hypercube labellings and antipodal monochromatic paths. *Discrete Applied Mathematics*, 161(10):1421–1426, 2013.
10. Jiří Fink. Perfect matchings extend to Hamilton cycles in hypercubes. *Journal of Combinatorial Theory, Series B*, 97(6):1074–1076, 2007.
11. Jiří Fink. Connectivity of matching graph of hypercube. *SIDMA*, 23(2), 2009.
12. Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531. Springer, 2007.
13. Vijay Ganesh, Charles W Odonnell, Mate Soos, Srinivas Devadas, Martin C Rinard, and Armando Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In *SAT*, pages 143–156. Springer, 2012.
14. Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer, 2014.
15. Petr Gregor. Perfect matchings extending on subcubes to Hamiltonian cycles of hypercubes. *Discrete Mathematics*, 309(6):1711–1713, 2009.
16. Marijn JH Heule, WA Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *FMCAD*, pages 181–188. IEEE, 2013.
17. 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.
18. Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *SAT*, 2014.
19. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *LPAR*, pages 36–50, 2004.
20. Frank Ruskey and Carla Savage. Hamilton cycles that extend transposition matchings in Cayley graphs of S_n. *SIDMA*, 6(1):152–166, 1993.
21. Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.
22. 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 *JELIA*, pages 684–693. 2014.
23. W. A. Stein and Etal. Sage Mathematics Software (Version 6.3), 2010.
24. Marc Thurley. sharpSAT–counting models with advanced component caching and implicit BCP. In *SAT*, pages 424–429. 2006.
25. Miroslav N Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In *SARA*, 2009.