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

Edward Zulkoski and **Vijay Ganesh** and **Krzysztof Czarnecki**
University of Waterloo, Waterloo, 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, à 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. We demonstrate the efficacy of our approach on a long-standing open conjecture regarding matchings of hypercubes.

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 [Biere *et al.*, 2009]. This is particularly true in software engineering, broadly construed to include testing, verification, analysis, synthesis, and security. Modern SMT solvers such as Z3 [De Moura and Bjørner, 2008], CVC4 [Barrett *et al.*, 2011], STP [Ganesh and Dill, 2007], and VeriT [Bouton *et al.*, 2009] 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. 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 cutdown the search space (akin to the "T" in DPLL(T)). 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.

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 [Velev and Gao, 2009]. 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.[1]

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 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 Fig-

[1]An extended version of this work is available in [Zulkoski *et al.*, 2015]. All code and data is available at `https://bitbucket.org/ezulkosk/sagesat`.

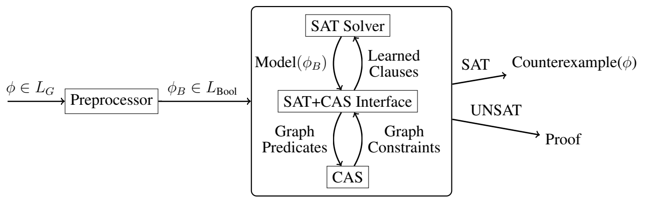

Figure 1: High-level overview of the MATHCHECK architecture. MATHCHECK takes as input a formula over fragments of mathematics supported by the underlying CAS system, and produces either a counterexample or a proof that none exists.

ure 2): 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.

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 [Nieuwenhuis *et al.*, 2004].

Here we provide a very high-level overview of the system, as displayed to Figure 1. A more detailed explanation is given in [Zulkoski *et al.*, 2015]. 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 [Ganesh *et al.*, 2012] or theory plugins in DPLL(T) [Nieuwenhuis *et al.*, 2004]). 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. [Velev and Gao, 2009] 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 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 grammar L_G described below. 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 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. 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 a 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 `EImpliesV(G)` constraint, indicating that an edge cannot exist without its corresponding vertices:

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

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 [Stein and others, 2010], which for now can be thought of as a collection of Python modules for mathematics.

3.2 Implementation

We have prototyped our system adopting the lazy-SMT solver approach (as in [Sebastiani, 2007]), specifically combining the Glucose SAT solver [Audemard and Simon, 2009] with the SAGE CAS [Stein and others, 2010]. 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 [Heule *et al.*, 2013] as well as checking the learned clauses produced by CAS predicates.

[2]For notational convenience, we often use existential quantifiers when defining constraints; these are unrolled in the implementation.

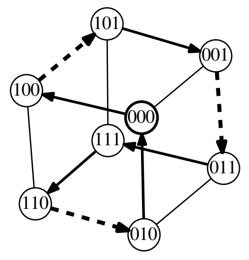

Figure 2: The dashed edges denote a generated matching, and the vertex 000 is restricted to be unmatched, as discussed in Section 4. A Hamiltonian cycle that includes the matching is indicated by the arrows.

4 Matchings Extend to Hamiltonian Cycles

We use our system to prove a long-standing open conjecture up to a certain parameter (dimension) related to hypercubes. Hypercubes have been studied for theoretical interest, due to their properties such as regularity and symmetry, but also for practical uses, such as in networks and parallel systems [Chen and Li, 2010].

The conjecture we look at was posed by Ruskey and Savage on matchings of hypercubes in 1993 [Ruskey and Savage, 1993]; although it has inspired multiple partial results [Fink, 2007; Gregor, 2009] and extensions [Fink, 2009], the general statement remains open:

Conjecture 1 (Ruskey and Savage, [Ruskey and Savage, 1993])**.** For every dimension d, any matching of the hypercube Q_d can be extended to a Hamiltonian cycle.

Consider Figure 2. The dashed 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 (specifically maximal forbidden matchings) to ensure that the conjecture holds for a given dimension. 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* matchings of Q_d does not scale, 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:

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

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 [Fink, 2007] 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:

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

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 included in [Zulkoski *et al.*, 2015]).

We encode this by adding the following constraints:

$$\textbf{EdgeOn(G):} \bigwedge\{v \Rightarrow \exists_{e \in X}\ e \mid v \in G_V\}, \quad \text{s.t. } X = \{e \mid e \in G_E \wedge \text{incident?}(v, e)\} \quad (4)$$

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

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 included in [Zulkoski *et al.*, 2015]).

To check if each matching extends to a Hamiltonian cycle, we create the CAS predicate `ExtendsToHamiltonian`

[3]We were unable to find the original source of the results for $d \leq 4$, however the result is asserted in [Fink, 2007]. We also verified these results using our system.

```
EXTENDSTOHAMILTONIAN()
    $g \leftarrow s$.getGraph($G$)
    $q \leftarrow$ CubeGraph($5$)
    for $e$ in $q$.edges() do
        if $e$ in $g$
            $q$.setEdgeLabel($e, 1$)
        else
            $q$.setEdgeLabel($e, 2$)
    $\langle$cycle, weight$\rangle \leftarrow$ TSP($q$)
    return weight $== 2 \cdot q$.order() $- |g|$
```

Figure 3: A CAS-defined predicate. Variable g corresponds to the matching found by the SAT solver.

(see Figure 3), 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 (dashed edges in Figure 2) 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 included in [Zulkoski *et al.*, 2015]).

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):

$$\bigvee\{e \mid e \in Q_{dE} \backslash C\} \cup \{h\} \tag{6}$$

Our full formula for Conjecture 1 is therefore:

$$\begin{aligned} \textbf{assert}\ & \text{EImpliesV}(G) \wedge \text{Matching}(G) \wedge \\ & \text{Forbidden}(G) \wedge \text{EdgeOn}(G) \wedge \text{Maximal}(G) \\ \textbf{query}\ & \text{ExtendsToHamiltonian}(G) \end{aligned} \tag{7}$$

5 Performance Analysis of MATHCHECK

We ran Formula 7 with $d = 5$ until completion. Since the run returned UNSAT, we conclude that the conjecture holds for the given dimension, improving upon known results.

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 7 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. We note that for lower dimensional cubes solving time was far less ($<$ 20 seconds). 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; learned clauses greatly reduce the search space and cut the number of necessary CAS predicate checks.

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 3 matches almost exactly with the actual code. 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

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) [Nieuwenhuis *et al.*, 2004]. There are also similarities with the idea of programmatic SAT solver Lynx [Ganesh *et al.*, 2012], which is an instance-specific version of DPLL(T). Our work is inspired by the recent SAT-based results on the Erdős discrepancy conjecture [Konev and Lisitsa, 2014]. Other works [Dooms *et al.*, 2005; Gebser *et al.*, 2014; Soh *et al.*, 2014] have extended solvers to handle graph constraints, by either creating solvers for specific graph predicates [Gebser *et al.*, 2014; Soh *et al.*, 2014], or by defining a core set of constraints with which to build complex predicates [Dooms *et al.*, 2005]. 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 [Bouton *et al.*, 2009] 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 study. 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. A more extensive version of this work can be found in [Zulkoski *et al.*, 2015].

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

References

[Audemard and Simon, 2009] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *International Joint Conference on Artificial Intelligence*, volume 9, pages 399–404, 2009.

[Barrett *et al.*, 2011] Clark Barrett, ChristopherL. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, 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.

[Biere *et al.*, 2009] 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.

[Bouton *et al.*, 2009] 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, *Proc. Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear.

[Chen and Li, 2010] 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.

[De Moura and Bjørner, 2008] 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.

[Dooms *et al.*, 2005] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming*, pages 211–225. Springer, 2005.

[Fink, 2007] Jiří Fink. Perfect matchings extend to Hamilton cycles in hypercubes. *Journal of Combinatorial Theory, Series B*, 97(6):1074–1076, 2007.

[Fink, 2009] Jiří Fink. Connectivity of matching graph of hypercube. *SIAM Journal on Discrete Mathematics*, 23(2):1100–1109, 2009.

[Ganesh and Dill, 2007] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[Ganesh *et al.*, 2012] 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 *Theory and Applications of Satisfiability Testing*, pages 143–156. Springer, 2012.

[Gebser *et al.*, 2014] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer, 2014.

[Gregor, 2009] Petr Gregor. Perfect matchings extending on subcubes to Hamiltonian cycles of hypercubes. *Discrete Mathematics*, 309(6):1711–1713, 2009.

[Heule *et al.*, 2013] Marijn JH Heule, WA Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[Konev and Lisitsa, 2014] Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *Theory and Applications of Satisfiability Testing*, 2014.

[Nieuwenhuis *et al.*, 2004] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2004.

[Ruskey and Savage, 1993] 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.

[Sebastiani, 2007] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[Soh *et al.*, 2014] 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.

[Stein and others, 2010] W. A. Stein et al. Sage Mathematics Software (Version 6.3), 2010.

[Velev and Gao, 2009] Miroslav N Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In *Symposium on Abstraction, Reformulation and Approximation*, 2009.

[Zulkoski *et al.*, 2015] Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki. MathCheck: A math assistant via a combination of computer algebra systems and SAT solvers. In *"Proc. Conference on Automated Deduction"*, Lecture Notes in Artificial Intelligence. Springer, 2015.