Exploiting Structure in Symmetry Detection for CNF

Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov
Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Department
The University of Michigan
{pdarga,liffiton,karem,imarkov}@eecs.umich.edu

ABSTRACT

Instances of the Boolean satisfiability problem (SAT) arise in many areas of circuit design and verification. These instances are typically constructed from some human-designed artifact, and thus are likely to possess much inherent symmetry and sparsity. Previous work [4] has shown that exploiting symmetries results in vastly reduced SAT solver run times, often with the search for the symmetries themselves dominating the total SAT solving time. Our contribution is twofold. First, we dissect the algorithms behind the venerable **nauty** [9] package, particularly the *partition refinement* procedure responsible for the majority of search space pruning as well as the majority of run time overhead. Second, we present a new symmetry-detection tool, **saucy**, which outperforms **nauty** by several orders of magnitude on the large, structured CNF formulas generated from typical EDA problems.

Categories and Subject Descriptors

G.2.2 Discrete Mathematics—Graph algorithms.

General Terms

Algorithms, Verification.

Keywords

Boolean satisfiability (SAT), symmetry, abstract algebra, backtrack search, partition refinement, graph automorphism.

1. INTRODUCTION

Boolean satisfiability instances arising in electronic design automation (EDA) applications typically are constructed from human-designed circuits, layouts, or other designs requiring the creativity and insight of a human engineer. In the interest of clarity and efficiency, designs will often exhibit significant structure. We are concerned with two types of structure: symmetry and sparsity. A design possesses *symmetry* if there is some rearrangement of the components of the design that preserves its structure. *Sparsity* is present when most design elements are directly related to only a few other elements in the whole design.

Work begun by Crawford [6], and improved by Aloul et al. [3, 4, 5], has shown that structural symmetries present in formulas, represented in conjunctive normal form (CNF), can be exploited to improve SAT solver performance. The CNF is converted into a colored undirected graph, whose symmetries are found and converted back into additional *symmetry-breaking predicates* (SBPs) concatenated to the original formula. These predicates eliminate symmetric regions of the search space explored by the SAT solver. The addition of the SBPs often reduces the search time so dramatically that the search for the symmetries themselves dominates the SAT-solving time.

In [3, 4, 5], the **nauty** [9, 10] program is used to extract the symmetries from the graph constructed from the CNF. **nauty** is the dominant publicly-available tool for symmetry extraction and graph isomorphism detection; however, it is not optimized for the large, sparse, specially-constructed graphs arising from EDA designs. Wang et al. [15] propose a custom symmetry detection engine for structural symmetries in Boolean functions. Their work exploits special properties of Boolean functions but abandons the use of partition refinement throughout the search due to its poor performance. Our new symmetry detection tool, **saucy**, makes use of the basic search structure of **nauty** but greatly improves the performance of the partition refinement procedure. Our tool outperforms **nauty** by several orders of magnitude on many benchmarks, including FPGA channel routing and microprocessor pipeline verification instances.

The remainder of this paper is outlined as follows. In Section 2, we discuss the construction of the graphs that are input to **nauty**. The partition refinement and search tree mechanisms employed in **nauty** are discussed in Section 3. In Section 4 we explore the methods of exploiting structure employed by **saucy**, and demonstrate **saucy**'s empirical success. We conclude in Section 5.

2. PREVIOUS WORK

Crawford [6] suggested a graph construction for the discovery of symmetries of formulas represented in CNF. Aloul et al. [5] extended his work; their construction is considered here. Let φ be a formula in CNF, over v variables and consisting of c clauses. We form a colored undirected graph G from φ using the following construction: (1) add a vertex for each variable and its negation, and for each clause; (2) add an edge between each variable and its negation (for *Boolean consistency*); (3) add an edge between each clause vertex

DAC 2004, June 7–11, 2004, San Diego, California, USA.

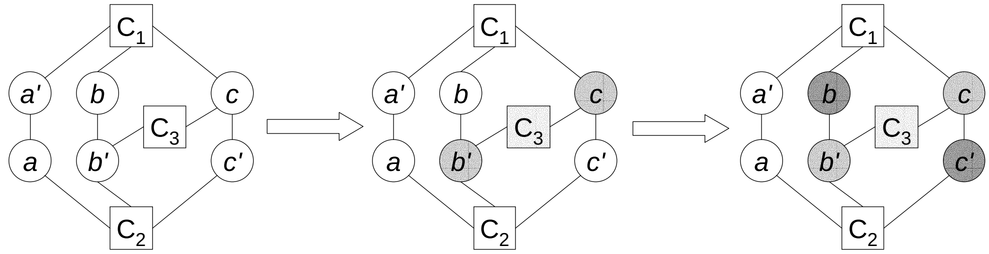

Figure 1: Application of ordered partition refinement. Graph (a) represents a graph generated from a CNF formula, with an initial coloring differentiating literals and clauses. In graph (b), some vertices have been differentiated by their degree. Refinement distinguishes two more vertices in (c), yielding a stable coloring.

Benchmark	Sym	Search	Total	% Sym
Hole-n	0.38	0.07	0.45	84.4
Urq	0.76	1.17	1.93	39.4
GRoute	38.76	5.08	43.84	88.4
FPGARoute	3.24	0.21	3.45	93.9
ChnlRoute	25.86	0.17	26.03	99.4
XOR	11.43	2.41	13.84	82.6
2pipe	23.50	8.01	31.51	74.6

Table 1: Run times in SAT solving (in seconds), taken from [4]. Each row represents a class of graphs and the average run times for symmetry detection with **nauty**, SAT solving, and total (symmetry with search) run time. The % Sym column lists the percentage of total time taken up by symmetry detection.

and its constituent literals; and (4) provide an initial coloring with variables and their negations colored differently than the clauses.

The graph constructed from the formula $(a' + b + c)(a + b' + c')(b' + c)$ is displayed in Figure 1(a); vertices C_1, C_2, and C_3 are colored differently than the literal vertices.

With this construction, there is a one-to-one correspondence between isomorphic CNF formulas and isomorphic colored graphs. Aloul et al. considered an alternative graph construction, where binary clauses are represented by an edge directly connecting their literals, saving one edge and one vertex for each such clause. They show in [5] that such graphs may have more symmetries than the original formula possesses; fortunately, the spurious symmetries are easy to detect and remove from the set returned by the symmetry detector.

Table 1 is taken from [4], using **nauty** for symmetry detection, and demonstrates the success of using SBPs to trim the search space explored by SAT solvers. In all but the synthetic Urquhart instances, the symmetry detection time dominates the SAT solving time, suggesting that the symmetry detection process must be optimized to further improve SAT solver performance.

3. OVERVIEW OF *NAUTY*

Finding the symmetries of a colored undirected graph is a complex problem in computational group theory, and thus our description of **nauty** will delve into the depths of discrete mathematics. A thorough development of the theory behind **nauty** can be found in [8, 9]; a somewhat gentler, higher-level overview is in [11].

Let G be an undirected graph with n vertices, and let $V = \{0, \ldots, n-1\}$. Each vertex in G is labeled with a unique value in V. A *permutation* on V is a bijection $\gamma : V \to V$. An *automorphism* of G is a permutation γ of the labels assigned to vertices in G such that $G^\gamma = G$; we say that γ is a structure-preserving mapping, or *symmetry*. The set of all such valid relabelings is called the *automorphism group* of G and is denoted $\mathrm{Aut}(G)$. A *coloring* is an ordered partition π of V—the cells of π form a sequence, not simply an unordered set. For example, the coloring provided to **nauty** for the graph in Figure 1 would be $[aa'bb'cc'|C_1C_2C_3]$.

nauty uses colorings to distinguish vertices that cannot possibly map into each other by any symmetry. Thus, given some coloring π, we can find some finer coloring π' that maximally distinguishes unmappable vertices. The process of creating π' from π is called *refinement*. The refinement procedure employed by **nauty** is based on Hopcroft's algorithm for minimizing the number of states in a finite automaton [2]: intuitively, if v_1 and v_2 map into each other by some symmetry, then they have the same degree, and v_1's neighbors have the same degree as v_2's neighbors, and so on.

As an example, consider the sequence of refinement steps in Figure 1. Vertices with different degree can never map into each other by any symmetry, and so can always be distinguished from each other. Thus, we can distinguish C_3 from the other clauses and b' and c from the other literals, yielding the refined coloring $\pi^{(1)} = [aa'bc'|b'c|C_3|C_1C_2]$, displayed in Figure 1(b). The process of splitting some of the cells *induces* further refinement—each vertex in a cell must have the same number of connections to vertices in every cell in the coloring, or else they can be distinguished. In the example, b and c' are connected to the second cell ($\{b', c\}$), while a and a' are not; we thus split the first cell to arrive at the further refined coloring $\pi^{(2)} = [aa'|bc'|b'c|C_3|C_1C_2]$, shown in Figure 1(c). No further refinement is induced by this split; we say this coloring is *stable*, and return $\pi' = \pi^{(2)}$.

If the refinement procedure returns a discrete coloring π', i.e. every cell of the partition is a singleton, then all vertices can be distinguished, so G must possess no symmetries besides the identity. However, if π' is not discrete, then there is some non-singleton cell in π' representing vertices that could not be distinguished based on degree—they may participate in some symmetry. **nauty** proceeds by selecting some non-singleton cell T of π', called the *target cell*, and

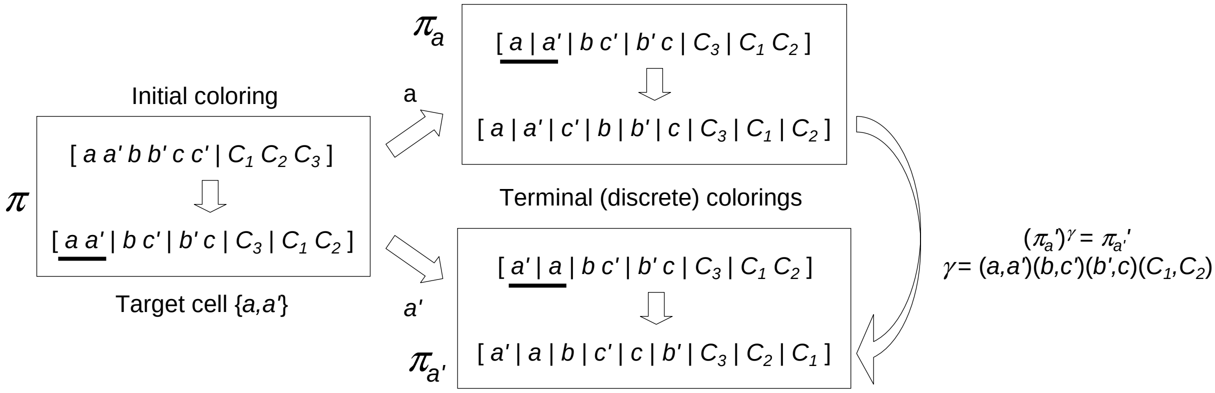

Figure 2: Search tree for finding symmetries. Each box represents a search tree node, where the top coloring is the initial coloring of the node and the bottom coloring is its stable refinement. Colorings π_a and $\pi_{a'}$ are created from π by individualizing each element of π's target cell in front of the others. Automorphisms of the graph G are permutations γ that map discrete colorings into each other, such that $G^\gamma = G$.

forms $|T|$ colorings descendant from π', each identical to π' except that one $t \in T$ is individualized in front of $T - \{t\}$. Each of these colorings is subsequently refined, and further descendant colorings are generated if the refined colorings are not discrete; this process is iterated until discrete colorings are reached. The colorings explored form a *search tree* with the discrete colorings at the leaves.

The leaves of the search tree represent possible symmetries of G. Let G^π be the relabeling of G with respect to discrete coloring π. If π_1 and π_2 are discrete colorings, and ${\pi_1}^\gamma = \pi_2$, then γ is a symmetry of G if and only if $G^{\pi_1} = G^{\pi_2}$. We can thus enumerate $\mathrm{Aut}(G)$ by fixing the first leaf encountered in the search, denoted ζ, and comparing it to every other discrete coloring: $\mathrm{Aut}(G) = \{\gamma : \pi \text{ discrete}, \zeta^\gamma = \pi, \text{and } G^\gamma = G\}$.

The search tree for our running example is shown in Figure 2. Since our stable coloring π' is not discrete, we select $(\{a, a'\})$ as our target cell, and form descendant colorings π_a and $\pi_{a'}$. After refining these new colorings, we find they are both discrete. The permutation γ mapping the stable colorings into each other is $\gamma = (a, a')(b, c')(b', c)(C_1, C_2)$; a check that $G^\gamma = G$ verifies that $\gamma \in \mathrm{Aut}(G)$. In fact, γ is the only symmetry G possesses besides the identity.

The set of automorphisms of a graph forms a permutation group under function composition. We can find some set $H \subseteq \mathrm{Aut}(G)$ such that every symmetry can be represented as a product of integer powers of elements of H. The set H *generates* $\mathrm{Aut}(G)$; we denote this by $\langle H \rangle = \mathrm{Aut}(G)$. H is *irredundant* if no $\gamma \in H$ can be generated from $H - \{\gamma\}$. The key result [5, 7] from computational group theory is that if H is irredundant, then it contains at most $\log_2 |G|$ elements, providing exponential compression of the solution, as it implicitly represents all of $\mathrm{Aut}(G)$. **nauty** produces such a generating set by performing a depth-first traversal of the search tree, and pruning away subtrees whose leaves would produce only automorphisms that can already be generated by previously discovered automorphisms. The details of determining which subtrees are uninteresting can be found in [9].

Not all leaves of the search tree are guaranteed to yield automorphisms of the graph. This occurs when the refinement procedure is unable to differentiate vertices that cannot be mapped into each other. However, such behavior only occurs in highly regular graphs, which are not common to those generated from CNF formulas from EDA applications. Even if bad leaves are encountered, backtracking can be avoided if we relax the constraint of determining generators for all of $\mathrm{Aut}(G)$—we can simply jump back to the greatest common ancestor of that leaf with ζ to skip the rest of that subtree. In the case that no bad leaves are generated or backtracked from, the number of nodes of the search tree is $O(n^3)$, but is often very small due to the considerable pruning possible. Table 2 shows various CNF formulas, the size of their corresponding graphs, and the number of nodes of the search tree explored.

4. SPARSITY AND *SAUCY*

Despite the empirical success of **nauty**, the data in Table 1 show that any further dramatic improvements in the performance of SAT solvers on symmetric instances must be made in the symmetry detector, not the SAT solver itself. **nauty**'s biggest successes have been in the mathematical domain, particularly concerning the graph isomorphism problem; for instance, **nauty** is the first program to successfully generate all isomorph-free graphs of degree 11. However, to achieve such successes **nauty** is completely general-purpose, assuming no properties of its input beyond being a colored undirected graph.

Graphs constructed from CNF formulas arising from EDA applications share some important characteristics. These graphs exhibit considerable sparsity—the average degree of a vertex is small, since most clauses do not have literals proportional to the number of variables. Improvements can be made to exploit this sparsity during partition refinement. We can extend the search data structures to include a mapping from vertices to their corresponding colorings, and use that mapping to attempt to refine only directly-connected cells. Since most cells are connected to only a few others, this has a dramatic effect on refinement performance. Other auxiliary structures can be used to annotate information regarding cell size and the position of singletons, improving the performance of target cell selection and refinement as well.

Instance		CNF		Graph		Symmetry Detection				SAT Solving	
Name	SAT?	Vars	Clauses	Vertices	Edges	Nodes	Nauty	Saucy	Speedup	zChaff	w/Sym
2pipe	N	892	6695	3575	14625	2415	2.93	0.03	97.67	0.18	0.13
3pipe	N	2468	27533	10048	58556	13041	57.53	0.16	359.56	3.20	6.44
4pipe	N	5237	80213	21547	167942	43071	523.64	0.77	680.05	228.82	153.50
5pipe	N	9471	195452	38746	403799	108345	3144.85	2.86	1099.60	347.92	122.85
6pipe	N	15800	394739	65839	812525	229503	mem	8.41	∞	time	time
7pipe	N	24065	731850	100668	1498971	431985	mem	18.82	∞	time	time
fpga11_13	N	286	1742	598	2288	1079	0.14	0.02	7.00	time	0.03
fpga11_20	N	440	4220	920	5060	1828	0.37	0.04	9.25	time	0.05
fpga13_10_sat	Y	195	905	530	1870	435	0.08	0.01	8.00	time	0.02
fpga13_12_sat	Y	234	1242	636	2556	561	0.14	0.01	14.00	time	0.02
hole11	N	132	738	276	990	253	0.02	0.01	2.00	102.25	0.02
hole12	N	156	949	325	1248	300	0.03	0.01	3.00	944.52	0.02
s3-3-3-3	Y	960	9156	2558	11700	465	1.87	0.03	62.33	22.95	0.16
s3-3-3-8	Y	912	8356	2432	10776	435	1.42	0.02	71.00	7.23	0.11
s4-4-3-1	Y	2688	33924	10354	44964	1378	88.74	0.19	467.05	441.18	218.53
s4-4-3-2	Y	2592	31736	9974	42348	1326	79.67	0.17	468.65	204.29	877.59
s4-4-3-3	Y	2592	31738	9970	42348	1326	75.98	0.17	446.94	time	884.78
s4-4-3-4	Y	2784	36176	10714	47580	1711	155.31	0.26	597.35	time	464.46
s4-4-3-5	Y	2880	38504	11072	50280	1378	101.63	0.21	483.95	time	134.09
s4-4-3-6	Y	2496	29628	9620	39888	1431	76.48	0.17	449.88	679.13	13.24
s4-4-3-7	Y	2688	33926	10362	44988	1326	78.96	0.19	415.58	831.04	18.27
s4-4-3-8	Y	1728	15320	6608	22296	1378	28.42	0.09	315.78	123.82	0.68
s4-4-3-9	Y	3360	51222	12920	64968	1596	209.52	0.38	551.37	75.21	time
Urq3_4	N	36	220	292	1208	210	0.02	0.01	2.00	0.07	0.02
Urq3_9	N	37	236	306	1289	231	0.02	0.01	2.00	3.38	0.02
x1_32	N	94	250	436	840	561	0.05	0.01	5.00	time	0.02
x1_36	N	106	282	492	948	867	0.07	0.01	7.00	time	3.47

Table 2: Statistics for various EDA-related instances. All times are in seconds. All programs were executed on a Pentium 4, 2.5 GHz machine with 1 GB memory. Cells marked "mem" represent executions of **nauty** which ran out of memory; those marked "time" are zChaff executions which timed-out after 1000 seconds. The *n-pipe* instances are microprocessor verification benchmarks [14]. The *fpga*, *s3*, and *s4* instances represent FPGA routing problems. The *hole* instances are from the DIMACS collection [1]. The *Urq* instances are from [13]. The *x1* instances are XOR-chains.

Given the particular construction discussed earlier, we can conclude that any vertex representing a clause is directly connected only to vertices representing variables or their complements; clauses are never connected to each other. In terms of partition refinement, suppose we are trying to refine cell S_1 with respect to S_2, and both S_1 and S_2 consist of clauses. Then for each $x \in S_1$, x has no connections to S_2; thus, vertices in S_1 are indistinguishable with respect to S_2, and no splitting is performed. Since this is true of every pair of clause cells, we can always use clause cells to refine only cells of literals, saving considerable work.

Motivated by these observations, we implemented a new symmetry detection tool, **saucy**[1], which capitalizes on these characteristics of graphs from CNF to improve performance. These improvements are focused around the partition refinement procedure, central to the search process. Indeed, in our experiments, typically over 80% of the execution time of symmetry detection engines is spent in refining the colorings produced as the search tree is traversed. By improving the refinement procedure, **saucy** delivers greatly improved run times over **nauty** on these specialized graphs.

Table 2 contains statistics on a variety of CNF formulas, their corresponding undirected graphs, and a performance comparison of **nauty** and **saucy**. These instances include FPGA routing (*fpga*, *s3*, *s4*) and microprocessor pipeline verification (*pipe*) problems. Clearly, **saucy** exhibits significant speedups on these structured graphs. Figure 3 demon-

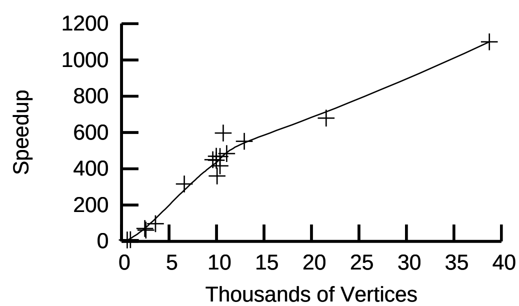

Figure 3: Relation of saucy's speedup over nauty to graph size, for graphs listed in Table 2 and requiring > 0.01 seconds for **saucy**. **saucy** exhibits a linear speedup over **nauty** on sparse EDA-related instances.

strates that **saucy**'s speedup is roughly linear in the size of the formulas given to it. The last two columns of Table 2 show the impact of adding symmetry breaking predicates [4] to the formulas input to the zChaff [12] SAT solver; the impact of symmetries should be comparable with other DPLL-based SAT solvers, as shown in [5]. Addition of the symmetry breaking predicates results in poor performance for a few of the instances; for most, however, the impact of adding the SBPs is dramatic, and the execution time of **saucy** does not dominate the SAT solving time.

In order to quantify **saucy**'s performance on dense graphs,

[1]Available on the GSRC Bookshelf for VLSI CAD at http://vlsicad.eecs.umich.edu/BK/SAUCY.

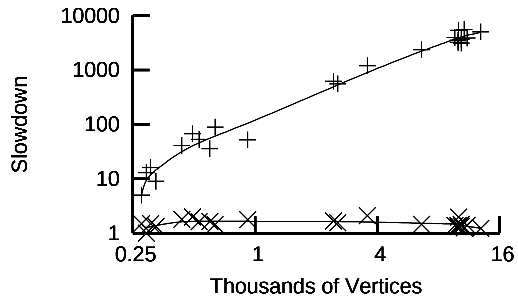

Figure 4: saucy (+) and nauty (×) slowdowns on dense graphs. The programs were executed on the complements of the graphs listed in Table 2, except 4pipe through 7pipe, for which **saucy** ran out of memory.

we constructed the *complement* of each graph in Table 2, defined as the graph with an edge wherever the original graph had none, and vice versa. Taking the complement of a graph preserves its automorphism group, and isolates run time overhead solely in the partition refinement algorithm. Figure 4 shows that **nauty** is relatively unaffected by the dense representation, while **saucy** exhibits a slowdown proportional to the size of the graph, which is expected since **saucy** is designed to take advantage of the sparsity present in realistic instances.

5. CONCLUSIONS AND FUTURE WORK

We have presented **saucy**, a new implementation of the **nauty** system specialized to the structured graphs generated from CNF formulas. By utilizing the sparsity and particular construction of these colored graphs, **saucy** achieves considerable performance improvements over **nauty**, making symmetry detection a feasible part of the satisfiability solving flow.

Future work will further utilize structure within refinement, and apply **saucy** to other discrete domains, such as constraint satisfaction problems, for which knowledge of symmetry might improve algorithmic performance.

Acknowledgments

This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center, and in part by the National Science Foundation under ITR grant No. 0205288. The authors would also like to recognize DoRon B. Motter for his work on the AutoGraph project, the precursor to **saucy**.

6. REFERENCES

[1] Dimacs challenge benchmarks. Available at ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Efficient symmetry breaking for boolean satisfiability. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 271–282, Acapulco, Mexico, August 2003.

[4] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient-symmetry breaking for boolean satisfiability. In *Proceedings of the Design Automation Conference*, pages 836–839, Anaheim, California, 2003.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *Transactions on Computer Aided Design*, 22(9):1117–1137, 2003.

[6] J. M. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*, 1992.

[7] J. B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley, 6th edition, 1999.

[8] B. D. McKay. Backtrack programming and isomorph rejection on ordered subsets. *Ars Combinatoria*, 5:65–99, 1978.

[9] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[10] B. D. McKay. nauty user's guide (version 1.5). Technical Report TR-CS-90-02, Australian National University, Department of Computer Science, 1990.

[11] T. Miyazaki. The complexity of mckay's canonical labeling algorithm. In *Groups and Computation II*, pages 239–256, 1995.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[13] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, January 1987.

[14] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the Design Automation Conference*, pages 226–231, 2001.

[15] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli. Structural detection of symmetries in boolean functions. In *International Conference on Computer Design*, pages 498–503, 2003.