Symmetry and Satisfiability: An Update

Hadi Katebi, Karem A. Sakallah, and Igor L. Markov

EECS Department, University of Michigan
{hadik,karem,imarkov}@umich.edu

Abstract. The past few years have seen significant progress in algorithms and heuristics for both SAT and symmetry detection. Additionally, the thesis that some of SAT's intractability can be explained by the presence of symmetry, and that it can be addressed by the introduction of symmetry-breaking constraints, was tested, albeit only to a rather limited extent. In this paper we explore further connections between symmetry and satisfiability and demonstrate the existence of intractable SAT instances that exhibit few or no symmetries. Specifically, we describe a highly scalable symmetry detection algorithm based on a decision tree that combines elements of group-theoretic computation and SAT-inspired backtracking search, and provide results of its application on the SAT 2009 competition benchmarks. For SAT instances with significant symmetry we also compare SAT runtimes with and without the addition of symmetry-breaking constraints.

1 Introduction

Over the past several years a fruitful interplay developed between the algorithms for graph automorphism and those of CNF satisfiability. The initial trigger was the black-box use of the **nauty** graph automorphism and canonical labeling package [12,11] to detect the symmetries in CNF formulas. This was accomplished by encoding a CNF formula as a colored graph [5,6,3] that was processed by **nauty** to produce an irredundant set of generators for the graph's automorphism group, and hence the formula's symmetries. These symmetries were subsequently used to augment the original formula with symmetry-breaking predicates that preclude a SAT solver from redundant search in symmetric portions of the solution space. It quickly became apparent, however, that the graphs of typical CNF formulas were too large (hundreds of thousands to millions of vertices) and unwieldy for **nauty** which was more geared towards small dense graphs (hundreds of vertices). The obvious remedy, changing the data structure for storing graphs from an incidence matrix to a linked list, yielded the **saucy** system which demonstrated the viability of graph automorphism detection on very large sparse graphs [7]. Unlike **nauty**, which also solved the canonical labeling problem, **saucy** was limited to just finding an irredundant set of symmetry generators. Canonical labeling seeks to assign a unique signature to a graph that captures its structure and is invariant under all possible labelings of its vertices. The **bliss** tool [10] adopted, and improved upon, **saucy**'s sparse data structures and solved both

O. Strichman and S. Szeider (Eds.): SAT 2010, LNCS 6175, pp. 113–127, 2010.

the symmetry detection and canonical labeling problems for both small dense and large sparse graphs. Close analysis of the search trees used in **nauty** and **bliss** revealed that they were primarily designed to solve the canonical labeling problem, and that symmetry generators were detected "along the way." Both tools employed sophisticated group-theoretic pruning heuristics to narrow the search for the canonical labeling of an input graph. The detection of symmetries benefited from these pruning rules, but also helped prune the "canonical labeling" tree since labelings that are related by a symmetry (i.e., a permutation of graph vertices that preserve the graph's edge relation) yield the same signature.

The next version of the **saucy** tool [8] introduced a major algorithmic change that delinked the search for symmetries from the search for a canonical labeling. This yielded a remarkable 1000-fold improvement in run time for many large sparse graphs with sparse symmetry generators, i.e., generators that "move" only a tiny fraction of the graph's vertices. This change also made the search for symmetries resemble, at least superficially, the search for satisfying assignments by a SAT solver. In this paper we further explore the connection between symmetry detection and satisfiability to better understand and improve symmetry detection algorithms. We present the **saucy** 2.1 algorithm and highlight its key feature, namely the organization of its search for symmetries along lines similar to those of CNF satisfiability. We also present and analyze the results of applying **saucy** 2.1 on the entire suite of SAT 2009 competition benchmarks. Finally, we examine the effect of static symmetry breaking on the most challenging benchmarks in this suite.

2 Preliminaries

We assume familiarity with basic notions from group theory, including such concepts as subgroups, cosets, group generators, group action, orbit partition, etc. Most of these concepts can be found in standard textbooks on abstract algebra, e.g. [9]. We will mainly focus on the automorphism group of a colored graph, i.e., the group of vertex permutations that preserve the graph's edge relation. We assume an n-vertex graph whose vertices are labeled with the integers $\{0, 1, \cdots, n-1\}$. For the rest of the paper, we will use V to denote this set. Permutations of V are bijections from V to V and are combined by functional composition. We will use γ and η to refer to permutations and employ both tabular and cycle notation to express them. The identity permutation will be denoted as ι. When clear from context $\gamma\eta$ will mean $\gamma \circ \eta$ where $\circ$ denotes functional composition. Finally, we will denote the symmetric group on the m-element set T as $S_m(T)$. The order of $S_m(T)$ is $m!$.

An *ordered partition* $\pi = [W_1|W_2|\cdots|W_m]$ of V is an ordered list of non-empty pair-wise disjoint subsets of V whose union is V. The subsets W_i are referred to as *cells* of the partition. Ordered partition π is *unit* if $m = 1$ (i.e., $W_1 = V$) and *discrete* if $m = n$ (i.e., $|W_i| = 1$ for $i = 1, \cdots, n$). An *ordered partition pair* Π is specified as

$$\Pi = \begin{bmatrix} \pi_T \\ \pi_B \end{bmatrix} = \begin{bmatrix} T_1 \,|T_2\,| \cdots |T_m \\ B_1 \,|B_2\,| \cdots |B_k \end{bmatrix}$$

with π_T and π_B referred to, respectively, as the top and bottom ordered partitions of Π. An ordered partition pair (OPP for short) Π is *isomorphic* if $m = k$ and $|T_i| = |B_i|$ for $i = 1, \cdots, m$; otherwise it is *non-isomorphic*. In other words, an OPP is isomorphic if its top and bottom partitions have the same number of cells, and corresponding cells have the same cardinality. An isomorphic OPP is *matching* if its corresponding non-singleton cells are *identical*. We will refer to an OPP as discrete (resp. unit) if its top and bottom partitions are discrete (resp. unit).

3 Implicit Representation of Permutation Sets

OPPs play a central role in the **saucy** symmetry detection algorithm we describe in this paper since they provide a compact implicit representation of sets of permutations. Specifically, a discrete OPP represents a single permutation, whereas a unit OPP represents all $n!$ permutations of V. In general, an isomorphic OPP

$$\Pi = \left[\begin{array}{c|c|c|c} T_1 & T_2 & \cdots & T_m \\ B_1 & B_2 & \cdots & B_m \end{array}\right] \tag{1}$$

represents $\prod_{1 \le i \le n} |T_i|!$ permutations. On the other hand, note that it is not possible to obtain well-defined mappings between the top and bottom partitions of a non-isomorphic OPP. Thus, non-isomorphic OPPs conveniently serve as empty sets of permutations.

Example 1. Here are several example OPPs and the permutation sets they encode.

- Discrete OPP: $\left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & 2 & 0 \end{array}\right] = \{(0\ 2\ 1)\}$
- Unit OPP: $\begin{bmatrix} 0,1,2 \\ 0,1,2 \end{bmatrix} = \{\iota, (0\ 1), (0\ 2), (1\ 2), (0\ 1\ 2), (0\ 2\ 1)\}$
- Isomorphic OPP: $\left[\begin{array}{c|c} 2 & 0,1 \\ 1 & 2,0 \end{array}\right] = \{(1\ 2), (0\ 2\ 1)\}$
- Matching OPP: $\left[\begin{array}{c|c|c} 1 & 0,2,4 & 3 \\ 3 & 0,2,4 & 1 \end{array}\right] = (1\ 3) \circ S_3(\{0,2,4\})$
- Non-isomorphic OPPs: $\left[\begin{array}{c|c|c} 0,2 & 1 \\ 1 & 2,0 \end{array}\right] = \emptyset, \left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & 2,0 \end{array}\right] = \emptyset$

4 Basic Enumeration of the Permutation Search Space

OPPs play a role similar to partial variable assignments in CNF-SAT solvers. Recall that a partial variable assignment on n Boolean variables can be encoded by an n-element array whose ith element indicates the value of the ith variable:

0, 1, or X for *unassigned*. A *complete* assignment is one in which all variables have been assigned a binary value; otherwise the assignment is *partial* and corresponds to a set of complete assignments that can be enumerated by considering all possible $0, 1$ combinations of the unassigned variables. A backtracking SAT solver extends a given partial assignment by choosing an unassigned variable and assigning to it one of the two binary values. This is referred to as a *decision* step and SAT solvers use a variety of decision heuristics to determine which variable to assign next and what value to assign to it. SAT solvers also employ *propagation* to avoid making decisions on variables whose values are implied (forced) by prior decisions. Finally, SAT solvers backtrack from "conflicts", i.e. assignments that cause the formula being checked to become unsatisfied.

As described earlier, a non-discrete OPP can be viewed as a representation of a set of permutations. The basic skeleton of a permutation enumeration algorithm can thus be patterned after a backtracking SAT algorithm that finds *all* satisfying assignments to a given CNF formula. An OPP is extended by:

- choosing a non-singleton cell (the *target* cell) from the top partition,
- choosing a vertex from the target cell (the *target* vertex), and
- mapping the target vertex to a vertex from the corresponding cell of the bottom partition.

The mapping step is accomplished by splitting the target cell so that the target vertex is in a cell of its own. The corresponding cell of the bottom partition is split similarly, placing the vertex to which the target vertex is mapped in a new singleton cell. Symbolically, given the isomorphic OPP in (1) assume that the ith cell is the target cell and let $j \in T_i$ be the target vertex. Mapping j to $k \in B_i$ refines the m-cell OPP Π to the following $(m+1)$-cell OPP Π':

$$\Pi' = \left[\begin{array}{c|c|c|c|c|c|c} T'_1 & T'_2 & \cdots & T'_i & T'_{i+1} & \cdots & T'_{m+1} \\ B'_1 & B'_2 & \cdots & B'_i & B'_{i+1} & \cdots & B'_{m+1} \end{array}\right]$$

where

$$\begin{array}{llll} T'_l = T_l & B'_l = B_l & & l = 1, \cdots, i-1 \\ T'_i = T_i - \{j\} & B'_i = B_i - \{k\} & & \\ T'_{i+1} = \{j\} & B'_{i+1} = \{k\} & & \\ T'_l = T_{l-1} & B'_l = B_{l-1} & & l = i+2, \cdots, m+1 \end{array}$$

To illustrate, consider the search tree in Figure 1(a) which enumerates all permutations of $V = \{0, 1, 2\}$ and checks which are valid symmetries of the indicated 3-vertex 2-edge graph. Each node of the search tree corresponds to an OPP which is the root of a subtree obtained by mapping a target vertex in all possible ways. For example, the unit OPP at the root of the search tree is extended into a 3-way branch by mapping target vertex 1 to 0, 1, and 2. It is important to point out that the choice of target vertex at each tree node and the order in which each of its possible mappings are processed does not affect the final set of permutations produced at the leaves of the search tree. It does, however, alter the order in which these permutations are produced. Note that valid automorphisms can be viewed as satisfying assignments whereas invalid ones are analogous to SAT

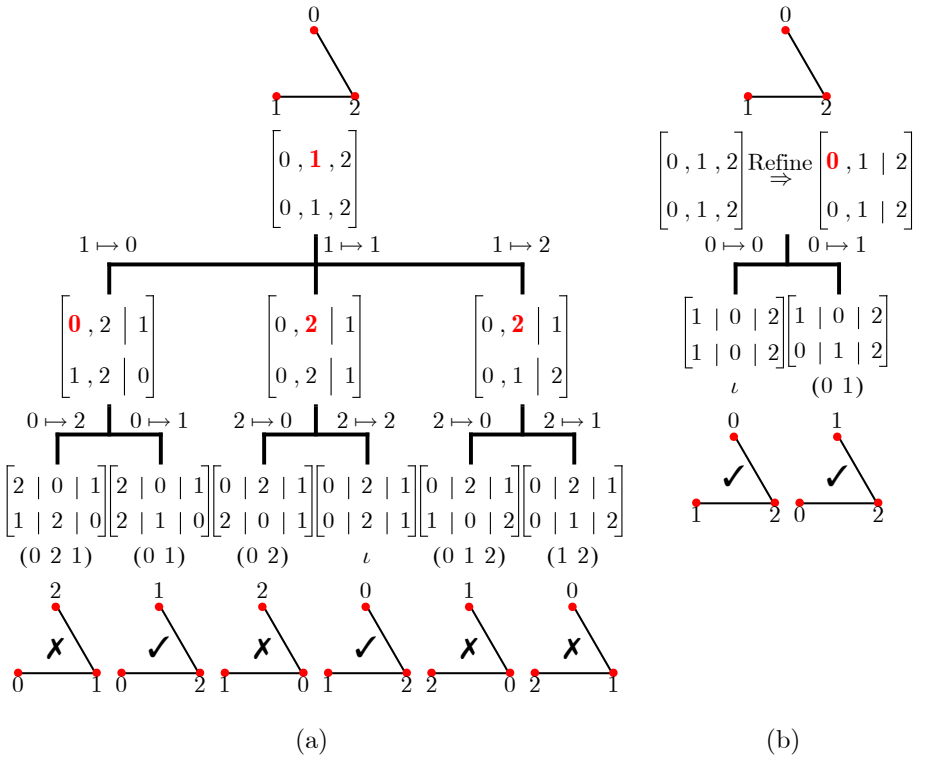
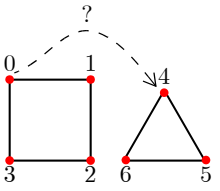


Fig. 1. Search trees for the automorphisms of a 3-vertex "line" graph. The target vertex ("decision variable") at each tree node is highlighted. (a) without partition refinement. (b) with partition refinement.

conflicts. The permutation search tree can be pruned significantly by performing *partition refinement* [1,7,12] before selecting and branching on a target vertex. This is analogous to Boolean constraint propagation in the SAT space and is standard in all algorithms for graph automorphism and canonical labeling. In the present context, partition refinement is applied *simultaneously* to the top and bottom partitions of the current OPP. This is illustrated in Figure 1(b) where vertex 2 is split from vertices 0 and 1 because it has a different degree.

As in SAT search, partition refinement is invoked after each decision assignment to determine the consequences of that decision. In some cases, this allows for the early detection of conflicts, i.e., concluding that the subtree rooted at the current tree node does not contain valid permutations. To illustrate, consider the 7-vertex graph in Figure 2 and assume that the decision to map vertex 0 to vertex 4 has just been made. This decision triggers partition refinement which causes the top and bottom partitions of the OPP to refine *non-isomorphically* proving that there are no automorphisms of this graph that map vertex 0 to vertex 4.



$$\left[\begin{array}{c|c|c} 1,2,3,4,5,6 & & 0 \\ 0,1,2,3,5,6 & & 4 \end{array}\right] \Rightarrow \left[\begin{array}{c|c|c|c} 2,4,5,6 & 1,3 & & 0 \\ 0,1,2,3 & 5,6 & & 4 \end{array}\right] \Rightarrow \left[\begin{array}{c|c|c|c} 4,5,6 & 2 & 1,3 & 0 \\ 0,1,2,3 & & 5,6 & 4 \end{array}\right]$$

Fig. 2. Example of non-isomorphic refinement. Attempting to map vertex 0 to vertex 4 causes the top and bottom partitions to split non-isomorphically into 4 and 3 cells, respectively.

5 Group-Theoretic Pruning

There are two primary pruning mechanisms anchored in group theory: *coset* pruning and *orbit* pruning. Both are routinely employed by symmetry detection and canonical labeling algorithms. The choice of OPPs to encode permutation sets introduces further opportunities to prune the search space as we show later in this section. To understand how coset and orbit pruning are employed in the search for a set of irredundant group generators requires the introduction of a few more group-theoretic concepts.

Let G be the automorphism group of our graph. The *action* of G on the graph vertices V is a map $* : G \times V \rightarrow V$ such that a) $\iota i = i$ for all $i \in V$, and b) $(\gamma\eta)(i) = \gamma(\eta i)$ for all $i \in V$ and all $\gamma, \eta \in G$. This group action induces an equivalence relation $\sim$ on the vertex set such that $i \sim j$ if and only if there exists $\gamma \in G$ with $\gamma i = j$. The resulting equivalence partition is referred to as the *orbit* partition and will be denoted by $\widehat{\pi}$. The orbit of $i \in V$ under G is the cell in $\widehat{\pi}$ that contains i and is conventionally written as Gi.

Let G_i denote the subgroup of G that "fixes" i, i.e., $G_i = \{\gamma \in G | \gamma i = i\}$. This is referred to as the *stabilizer* subgroup of i. The (left) coset of G_i in G containing η is defined as the set $\{\eta\gamma | \gamma \in G_i\}$. Note how this definition implies that *any* coset element can generate the entire coset by composing that element with the elements of G_i. The set of (left) cosets of G_i partitions G into equal-sized subsets. Now assume that Z is a set of irredundant generators for G_i. A set of generators for the parent group G can be obtained by augmenting Z with a *single* representative from each coset of G_i. This set may, however, contain redundant generators that must be eliminated with the aid of the orbit partition.

To place these pruning mechanisms in the context of the permutation search tree, consider a tree node that represents a group G and assume that the subtree under G is expanded by mapping vertex i to vertices $i, i_1, i_2, \cdots, i_k$ in that order (see Figure 3). As above, the permutation subset corresponding to mapping i to itself is G_i, the stabilizer subgroup of i. The other subsets will be denoted by $H_{i \mapsto i_j}$ and correspond to those permutations that, among other things, map i

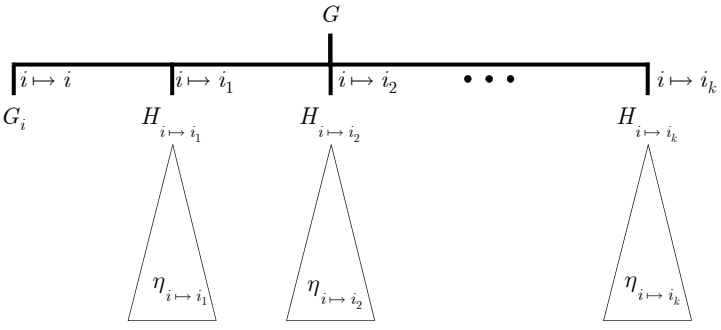


Fig. 3. Structure of the permutation search tree

to i_j. To find a set of irredundant generators for G we must now "solve" up to k independent problems where problem i_j seeks to determine whether the set of permutations $H_{i \mapsto i_j}$ is a coset of G_i. This is accomplished by searching $H_{i \mapsto i_j}$ for a *single permutation* that "satisfies" the graph edge relation, i.e., a permutation that is an automorphism of the graph. If no such permutation exists, then $H_{i \mapsto i_j}$ is "unsatisfiable", i.e., it is not a coset of G_i. This problem is remarkably similar, structurally, to the problem of finding a satisfying assignment to a CNF formula or proving that no such assignment exists.

Let permutation $\eta_{i \mapsto i_j}$ denote the "solution" to problem i_j. Clearly, $\eta_{i \mapsto i_j}$ serves as a coset representative for $H_{i \mapsto i_j}$ and can be added to the set of generators for G. Additionally, vertices i and i_j must now be in the same orbit. Thus, if the orbit of i_j contains vertex i_l with $l > j$, then problem i_l can be skipped since its corresponding coset must necessarily contain redundant generators.

A key pruning mechanism that is enabled by the OPP encoding of permutation sets is the quick discovery of candidate coset representatives. This occurs when the OPP at a given tree node is *matching*. For example, the matching OPP

$$\left[\begin{array}{c|c|c|c|c} 1 & 0,2 & 4,6,7 & 3 & 5 \\ 3 & 0,2 & 4,6,7 & 5 & 1 \end{array}\right]$$

encodes the permutation set:

$$(1\ 3\ 5) \circ S_2\left(\{0,2\}\right) \circ S_3\left(\{4,6,7\}\right)$$

which clearly include the permutation (1 3 5). If this permutation is found to be a symmetry of the graph, we can terminate the search in this coset and return this permutation as the coset representative. Significantly, if this permutation is found not to be a symmetry of the graph, then we can also terminate the search in this subtree since all other permutations in this subset are composed with this permutation! For large graphs, this pruning mechanism leads to a drastic reduction in the size of the search tree and a commensurate reduction in run time.

Finally, it is interesting to note that in addition to finding a set of irredundant generators for G, symmetry detection algorithms can also compute the order of G using the orbit-stabilizer and Lagrange theorems [9]: $|G| = |G_i| \cdot |Gi|$.

6 The Algorithm

The symmetry detection algorithm is basically a depth-first traversal of the permutation search tree. To enable coset and orbit pruning, the left-most tree path must correspond to a sequence of subgroup stabilizers ending in the identity (a so-called *subgroup decomposition*). In other words, "decisions" along this path must map each selected target vertex to itself. This requirement does not apply to decisions in other parts of the tree. The tree is pruned by systematic application of the four pruning rules elaborated earlier, namely:

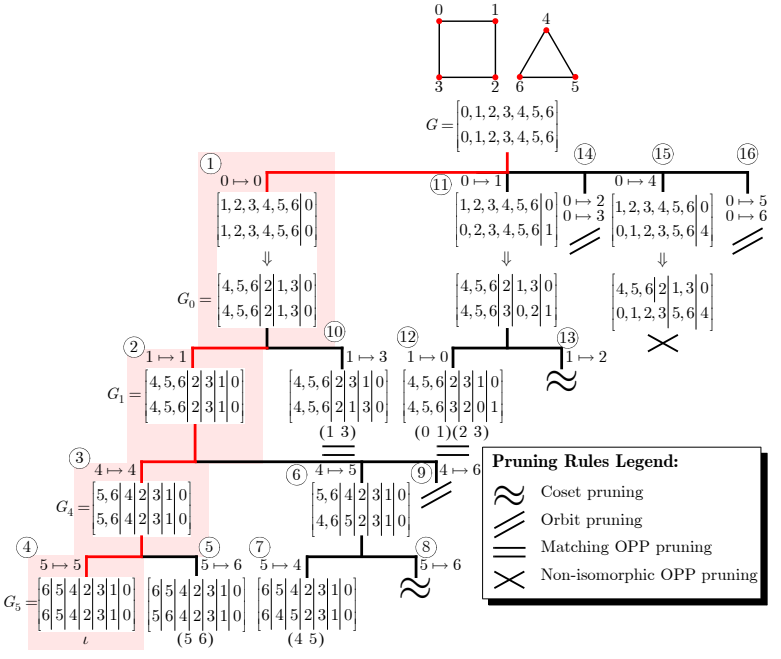
- **Coset pruning** which terminates the search in a coset subtree as soon as a coset representative is found.
- **Orbit pruning** which avoids searching the subtree of coset $H_{i \mapsto j}$ if j is already in the orbit of i.
- **Matching OPP pruning** which can identify a candidate permutation at a tree node without the need to explore the subtree rooted at that node.
- **Non-isomorphic OPP pruning** which indicates that there are no permutations in the subtree rooted at that node which are symmetries of the graph.

It is important to note that coset and orbit pruning are, in some sense, *intrinsic* and can (should?) be viewed as part of the "specification" of the automorphism problem. In other words, any graph automorphism algorithm must return a set of irredundant generators, and thus, must employ coset and orbit pruning. The two other pruning rules, based on the OPP encoding of permutation sets, represent algorithmic enhancements that assist in eliminating unnecessary search.

This algorithm has been implemented in the **saucy** 2.1 symmetry detection tool. A trace of the algorithm illustrating all four pruning mechanisms is shown in Figure 4.

7 Experimental Evaluation

We ran symmetry detection using **saucy** 2.1 on the complete set of 1183 SAT 2009 competition benchmarks and checked satisfiability with symmetry-breaking on the 47 most difficult ones (see below). Experiments were conducted on a SUN workstation equipped with a 3GHz Intel Dual-Core CPU, a 6MB cache and an 8GB RAM, running the 64-bit version of Redhat Linux. The run time results are shown in Figure 5. With a time-out of 500 seconds, **saucy** finished on all but 18 benchmarks from the crafted category belonging to three families: connum (6 instances), equilarge (3 instances), and mod2-rand3bip (9 instances). By varying the branching heuristics, **saucy** was able to quickly



Initialization: $\widehat{\pi} = \{0|\ 1|\ 2|\ 3|\ 4|\ 5|\ 6\}$, $Z = \emptyset$.

1. Fix vertex 0 and refine
2. Fix vertex 1
3. Fix vertex 4
4. Fix vertex 5; $G_5 = \{\iota\}$
5. Search for representative of coset $H_{5\mapsto 6}$;
   $Z = \{(5\ 6)\}$; $\widehat{\pi} = \{0|\ 1|\ 2|\ 3|\ 4|\ 5, 6\}$; $|G_4| = |G_5| \cdot |G5| = 1 \cdot 2 = 2$
6. Search for representative of coset $H_{4\mapsto 5}$
7. Found representative of coset $H_{4\mapsto 5}$;
   $Z = \{(5\ 6), (4\ 5)\}$; $\widehat{\pi} = \{0|\ 1|\ 2|\ 3|\ 4, 5, 6\}$
8. **Coset pruning**: no need to explore since we have already found a coset representative for $H_{4\mapsto 5}$
9. **Orbit pruning**: no need to explore since 6 is already in the orbit of 4.
   $|G_1| = |G_4| \cdot |G4| = 2 \cdot 3 = 6$
10. Search for representative of coset $H_{1\mapsto 3}$;
    Matching OPP pruning: found representative of coset $H_{1\mapsto 3}$.
    $Z = \{(5\ 6), (4\ 5), (1\ 3)\}$; $\widehat{\pi} = \{0|\ 2|\ 1,\ 3|\ 4, 5, 6\}$; $|G_0| = |G_1| \cdot |G1| = 6 \cdot 2 = 12$
11. Search for representative of coset $H_{0\mapsto 1}$
12. **Matching OPP pruning**: found representative of coset $H_{0\mapsto 1}$.
    $Z = \{(5\ 6), (4\ 5), (1\ 3), (0\ 1)(2\ 3)\}$; $\widehat{\pi} = \{0, 1, 2,\ 3|\ 4, 5, 6\}$
13. **Coset pruning**: no need to explore since we have already found a coset representative for $H_{0\mapsto 1}$
14. **Orbit pruning**: no need to explore since 2 and 3 are already in the orbit of 0.
15. **Non-isomorphic OPP pruning**: 0 cannot map to 4.
16. **Orbit pruning**: no need to explore since 5 and 6 are already in the orbit of 4.
    $|G| = |G_0| \cdot |G0| = 12 \cdot 4 = 48$

Fig. 4. Search tree for graph automorphisms of the "square and triangle" graph and relevant computations at each node. The shaded region corresponds to subgroup decomposition.

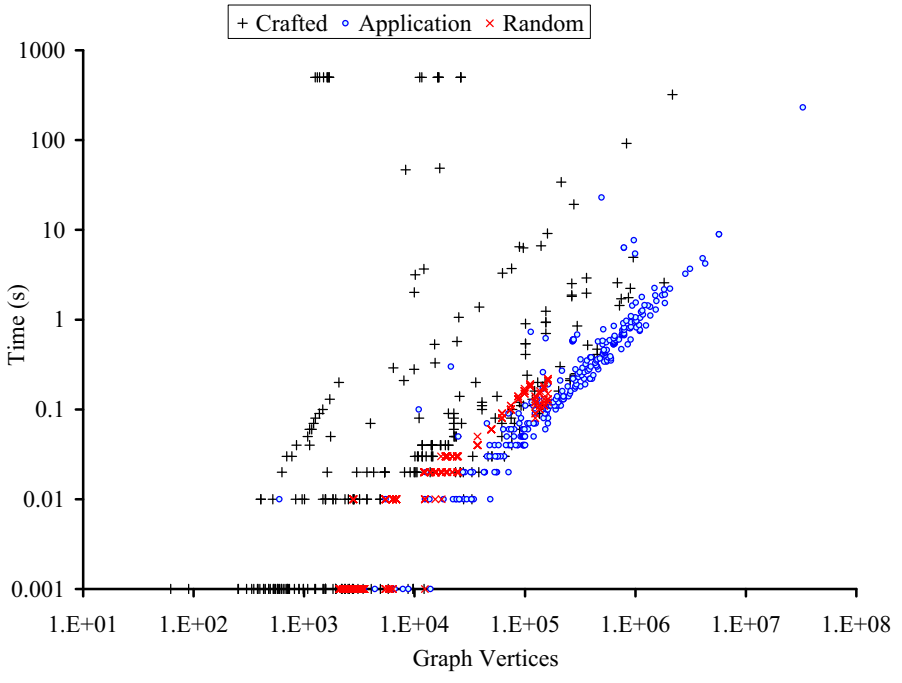


Fig. 5. saucy 2.1 run time, in seconds, as a function of graph size for the SAT 2009 competition benchmarks. A time-out of 500 seconds was applied.

solve the six connum instances (in less than 5 seconds each) but still failed to process the remaining twelve even with a much larger time-out limit. In general, instances from the crafted category were more challenging for **saucy** than similarly-sized instances from the random or application suites. The largest benchmark `post-cbmc-zfcp-2.8-u2-noholes`, an application instance with about 11 million variables and 33 million clauses, was modeled by a graph with over 32 million vertices and required about 231 seconds to process. As the figure shows, there is a weak trend towards larger run times for larger graphs. However, run time seems to also depend on other attributes of a graph besides its absolute size (number of vertices.) In any case, **saucy** is extremely fast, finishing in less than one second on 93% (1101) of all benchmarks.

The "amount" of symmetry present (order of the automorphism group) in each benchmark is shown in Figure 6. In total, only 323 benchmarks exhibited non-trivial symmetries, and the order of the largest automorphism group (for benchmark `hsat_vc11813`) was an astronomical $5.091978 \times 10^{142761}$. The figure only lists those benchmarks whose automorphism group has an order between 2 (meaning one non-trivial symmetry) and 10^{60} (a total of 293 out of 323.) Of the 610 benchmarks in the random category, 606 had no symmetry at all, and the remaining four had just one symmetry. In the application category, **saucy** reported the presence of symmetry in about 50% of the benchmarks (144 out of

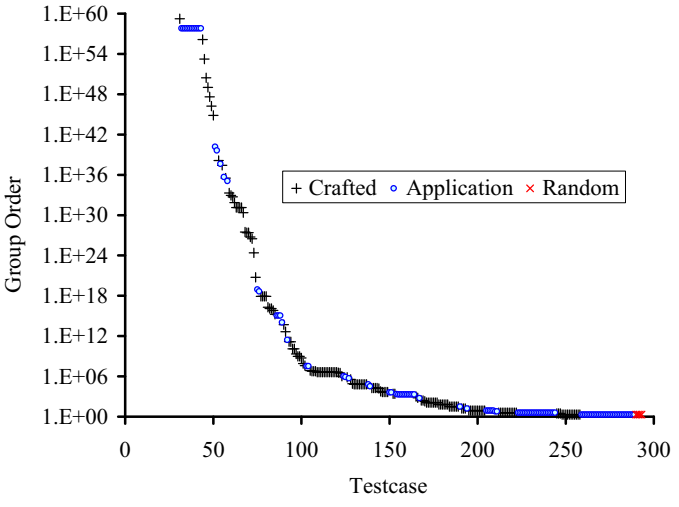


Fig. 6. saucy 2.1 group order for the SAT 2009 competition benchmarks

292), and it found symmetry in about two-thirds (175 out of 263) of the crafted benchmarks which it was able to process within the time-out limit.

Figure 7 shows the relation between the order of the automorphism group and the number of generators returned by **saucy** for the 293 benchmarks. Symmetry detection algorithms, including **saucy**, guarantee to produce no more than $n-1$ generators for an n-vertex graph. The number of reported generators in these results is significantly less than $n-1$. This, however, is not inconsistent with the well-known fact that the number of (irredundant) generators is exponentially smaller than the order of the corresponding symmetry group.

To evaluate the effectiveness of static symmetry breaking, we applied **shatter** [2] to 47 "difficult" benchmarks. These included 13 application and 34 crafted benchmarks that had significant symmetry and either could not be solved by any of the SAT 2009 competition solvers (38 benchmarks), or required at least 1000 seconds to be solved (9 benchmarks). The **shatter** flow consists of running **saucy** on a CNF instance to obtain its symmetry generators, followed by the creation of CNF symmetry-breaking predicates (SBPs) using the encoding in [4], and finally passing the original instance augmented with the SBPs to a SAT solver. Figures 8(a) and 8(b) depict the increase in instance size (variables and clauses) for each of these benchmarks due to the addition of the SBPs. For 29 of the benchmarks, the number of added SBP clauses was quite insignificant (less than 4%). The additional clauses for the remaining 18 benchmarks ranged from 25% to 133% of the original number. The number of variables increased by less than 1% for 23 benchmarks and by 9% to an order of magnitude for the remaining 24 benchmarks.

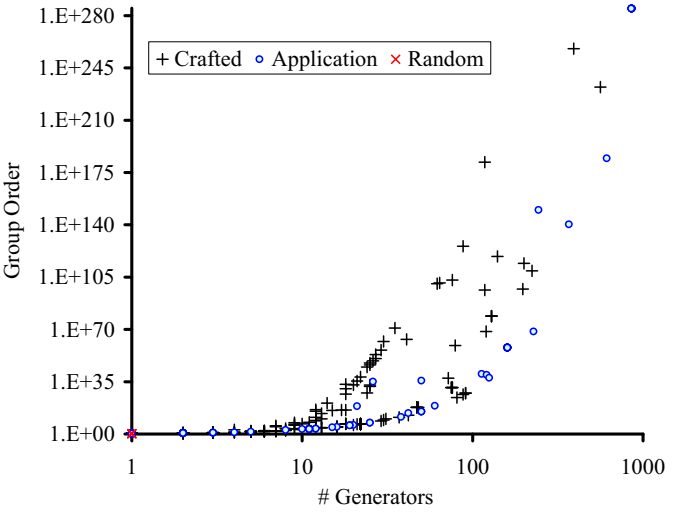
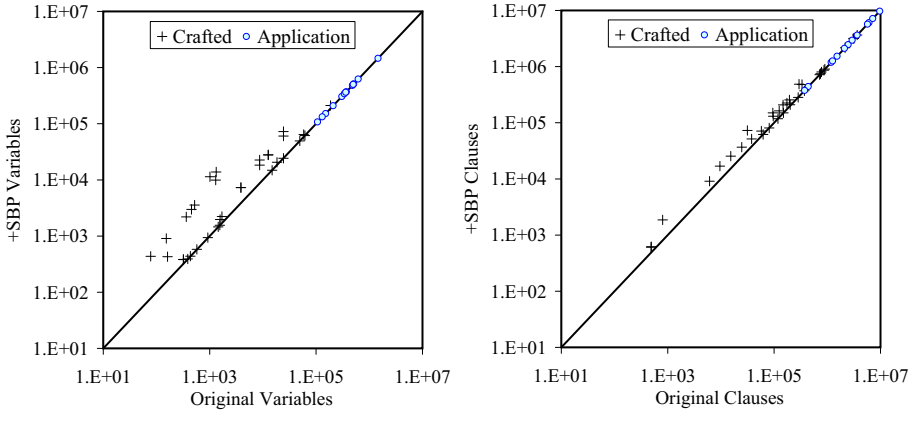


Fig. 7. saucy 2.1 group order as a function of the number of group generators for the SAT 2009 competition benchmarks



(a) Variables (b) Clauses

Fig. 8. Number of variables and clauses before and after the addition of SBPs

To obtain meaningful statistical data, we used a script that re-orders the variables and clauses in a CNF instance using a random seed[1] to create twenty different versions of each benchmark: ten for the original and ten for the SBP-augmented benchmark. We then applied the best solver for a given benchmark, based on the 2009 competition results, to these twenty versions. The run time

[1] We obtained the reorder.c script and a seed generator from Laurent Simon. The script was originally written by Edward Hirsh and later modified by Simon to handle large benchmarks.

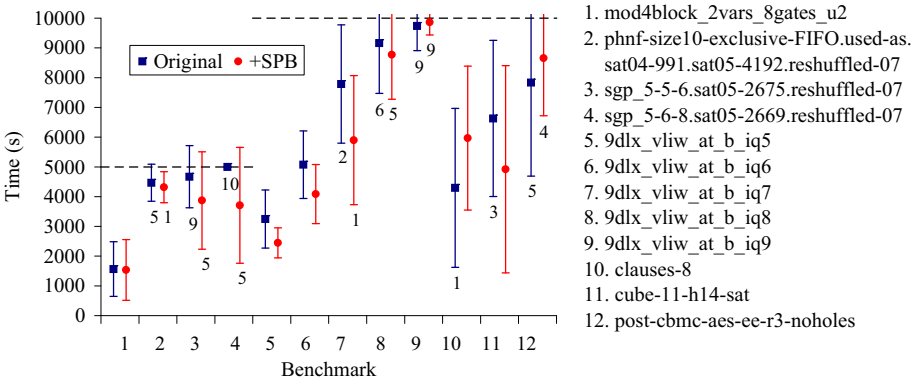


Fig. 9. SAT solver run time results before and after the addition of SBPs. The first 4 benchmarks are from the crafted category with a time-out limit of 5,000 seconds, whereas the last 8 are from the application category with a time-out limit of 10,000 seconds. minisat 2.0 was used for benchmarks 10 and 11 and glucose 1.0 was used for the others. The data for each benchmark (original or with SBPs) show the mean and standard deviation over the ten runs, including the runs that timed out. Since time-outs skew the statistics, the number of runs that timed out is indicated below the "error" bars (absence of a number indicates that all ten runs finished within the time-out limit.).

results comparing search times with and without the addition of SBPs for 12 of the 47 benchmarks are shown in Fig. 9; experiments on the remaining 35 benchmarks were still running at press time. For this limited subset, the SBP-augmented versions generally led to fewer time-outs and, in all but three cases, were solved faster than the original versions. Four of these benchmarks (2, 4, 8, and 9) which were reported to be unsolvable within the time-out limits of the competition, were solved with the addition of SBPs. Interestingly, though, benchmarks 2, 8, and 9 were solved on our experimental machine even without the addition of SBPs. These anomalies are possibly due to the use of different machines with varying configurations in the SAT 2009 competition and merely point out that we must be careful not to draw incorrect conclusions from empirical data.

8 Conclusions

It has been conjectured that symmetries in CNF formulas contribute to the intractability of SAT. The availability of extremely-efficient scalable symmetry detection algorithms, such as **saucy** 2.1, has enabled the testing of this hypothesis on very large CNF formulas. The question, however, remains open. Many intractable CNF instances (e.g., random instances) possess no or little

symmetry. Those that possess significant symmetry may or may not benefit from *static* symmetry breaking for a number of possible reasons. For example, the generators produced by a symmetry detection algorithm may not be the most suitable for symmetry breaking. Better branching heuristics while searching the permutation space might yield more useful generators for SAT solving. A more promising direction is the integration of symmetry detection within the SAT solver itself [13]. The raw speed of modern symmetry detectors like **saucy** suggests that they can be invoked during the SAT search with minimal overhead. And unlike static symmetry breaking, dynamic symmetry detection does not require the addition of large SBPs, and can uncover hidden/conditional symmetries adaptively. We plan to pursue this in our future research.

Acknowledgments

This work was funded in part by NSF award number 0705103. We gratefully acknowledge Timothy Lane's help at various stages of this project and offer our sincere thanks to Allen Van Gelder for his careful reading of, and thoughtful comments on, the manuscript.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: Efficient symmetry-breaking for boolean satisfiability. In: Proc. 40th IEEE/ACM Design Automation Conference (DAC), Anaheim, California, pp. 836–839 (2003)
3. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult instances of boolean satisfiability in the presence of symmetry. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 22(9), 1117–1137 (2003)
4. Aloul, F.A., Sakallah, K.A., MarkovEfficient, I.L.: symmetry breaking for boolean satisfiability. In: Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, pp. 271–282 (2003)
5. Crawford, J.: A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In: AAAI 1992 Workshop on Tractable Reasoning, San Jose, CA, pp. 17–22 (1992)
6. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Principles of Knowledge Representation and Reasoning (KR 1996), pp. 148–159 (1996)
7. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Proc. 41st IEEE/ACM Design Automation Conference (DAC), San Diego, California, pp. 530–534 (2004)
8. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: Proc. 45th IEEE/ACM Design Automation Conference (DAC), Anaheim, California, pp. 149–154 (2008)

9. Fraleigh, J.B.: A First Course in Abstract Algebra, 6th edn. Addison Wesley Longman, Reading (2000)
10. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007), New Orleans, LA (2007)
11. McKay, B.D.: Nauty user's guide (version 2.2), `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`
12. McKay, B.D.: Practical graph isomorphism. Congressus Numerantium 30, 45–87 (1981)
13. Schaafsma, B., Heule, M.J., Maaren, H.: Dynamic symmetry breaking by simulating Zykov contraction. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 223–236. Springer, Heidelberg (2009)