Faster Symmetry Discovery using Sparsity of Symmetries

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

ABSTRACT

Many computational tools have recently begun to benefit from the use of the symmetry inherent in the tasks they solve, and use general-purpose graph symmetry tools to uncover this symmetry. However, existing tools suffer quadratic runtime in the number of symmetries explicitly returned and are of limited use on very large, sparse, symmetric graphs. This paper introduces a new symmetry-discovery algorithm which exploits the sparsity present not only in the input but also the output, i.e., the symmetries themselves. By avoiding quadratic runtime on large graphs, it improves state-of-the-art runtimes from several days to less than a second.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph algorithms

General Terms

Algorithms, Verification.

Keywords

Symmetry, sparsity, graph automorphism, partition refinement, constraint satisfaction problems, Boolean satisfiability, model checking.

1. INTRODUCTION

Many application domains, within the electronic design automation community and beyond, have recently begun to exploit the *symmetry* inherent in their inputs. Boolean satisfiability and general constraint solvers convert the symmetries present in their inputs into additional predicates or constraints that assist solvers in avoiding symmetric, and hence redundant, portions of their search space [2, 11, 14]. Model checkers use the symmetries of a system to reduce the size of state space that must be explored [12]. The functional symmetries of a Boolean function can be utilized at numerous stages of logic synthesis and optimization, including technology mapping [4] and postplacement rewiring [5].

One popular means of discovering the symmetries of a system is to first convert the system into a graph, and employ a general-purpose graph symmetry tool to uncover the symmetries [3]. These symmetries can then be reflected back into the original system domain. The oldest and most established graph symmetry program is McKay's *nauty* [10]. However, *nauty* does not scale well with the number of vertices in a graph. Graphs from domains such as microprocessor verification CNF formulas tend to have many vertices, but are also very *sparse*: the average number of neighbors of a vertex tends to be a small constant. Darga et al. [7], with their program *saucy*, showed that *nauty*'s performance could be improved considerably on such graphs by exploiting that sparsity. Junttila and Kaski [9] made several additional improvements in sparse graph symmetry discovery with their program *bliss*.

This paper introduces a new algorithm for symmetry discovery which outperforms *nauty*, *saucy*, and *bliss* by many orders of magnitude on large, sparse graphs. The core insight that inspired our work is that symmetry discovery algorithms can exploit sparsity not only in the input graph, but also in the symmetries themselves. Many of the symmetries discovered in sparse graphs are themselves sparse, only rearranging a small number of vertices. The runtime of our algorithm scales in proportion to the total number of rearranged vertices among all the symmetries it finds, rather than roughly quadratic in the number of symmetries, as observed with several existing tools.

The rest of the paper is organized as follows. In Section 2, we describe our complete algorithm. In Section 3, we compare our algorithm to those of *nauty*, *saucy*, and *bliss*. In Section 4, we report experimental results on a representative sample of very large and sparse graphs. Finally, we conclude in Section 5.

2. THE ALGORITHM

First, a few preliminary definitions and concepts. A **graph** G is an ordered pair (V, E) with vertex set V and edge set E, where each edge is a set of two distinct elements of V. A **symmetry** γ is a permutation of V such that $G^\gamma = G$, or equivalently, that $E^\gamma = E$, since $V^\gamma = V$ by definition. A permutation γ **respects a partition** π of V if, for each $v \in V$, v and v^γ are in the same cell of π. The set of all symmetries of a graph G which respect a partition π is called the **automorphism group** of G under π and is denoted $\mathrm{Aut}(G)_\pi$.

A graph with n vertices may have as few as one symmetry (the **identity**, denoted ι), and as many as $n!$ symmetries.

However, this potentially exponential set of symmetries can be concisely represented by a subset with at most $n-1$ symmetries. This small set **generates** the large set via every possible composition of its elements. The goal of symmetry discovery, then, is to find such a set for $\mathrm{Aut}(G)_\pi$.

2.1 Refinement

The first step in symmetry discovery is to distinguish, as much as possible, vertices which are not symmetric. This is achieved using **vertex invariants**: properties of vertices which are invariant under symmetry. There are many such properties, and they vary widely in their distinguishing ability and in their computational overhead.

A simple invariant is that of vertex degree: two vertices cannot be symmetric if they have different numbers of neighbors. Therefore, we can **refine**, or further subdivide, a partition π by partitioning each cell S of π based on the degrees of the vertices contained in S. Consider the five-vertex chain graph: 1 — 2 — 3 — 4 — 5, with an initial partition $\pi = \{\{1,2,3,4,5\}\}$. Refining π using the degree invariant yields a new partition $\{\{1,5\},\{2,3,4\}\}$. Note that we are using partitions to keep track of which vertices are definitely not symmetric.

The degree invariant has pruned the space of permutations we must explore, but still more vertices can be distinguished. The vertex in the middle of the chain, vertex 3, clearly is not symmetric to any other vertex in the graph, despite having the same degree as vertices 2 and 4. This intuition is captured by a slightly more sophisticated invariant: the **connection function** c. Given a vertex v and an invariant set S of vertices, $c(v,S)$ is the number of elements of S which are neighbors of v.

The set S used in the connection function must be invariant; that is, $S^\gamma = S$ for every $\gamma \in \mathrm{Aut}(G)_\pi$. This condition is trivially satisfied by the cells of π. We can therefore select some cell S of π and compute $c(v,S)$ for every v, and then further partition the cells of π based on those values. This refinement of π into a finer partition π' does not prune away any symmetries; every symmetry which respects π also respects π'. Therefore, $\mathrm{Aut}(G)_\pi = \mathrm{Aut}(G)_{\pi'}$, and we can use the cells of π' to further refine itself. The process iterates until no further refinement is possible; we call such a partition **equitable**.

Consider again our chain graph example. The first application of the connection invariant yields the partition $\{\{1,5\},\{2,3,4\}\}$. Since a cell was split, we try to refine this partition further using the newly split cells. Using $\{1,5\}$ to refine $\{2,3,4\}$, we find that vertices 2 and 4 each have one connection to the cell while vertex 3 has no such connections. We therefore distinguish vertex 3. No other refinement is possible in this case; the final result of our vertex refinement is then $\{\{1,5\},\{2,4\},\{3\}\}$.

The formal algorithm for partition refinement is based on work by Hopcroft [1], and is discussed in detail in several sources [7, 10, 13]. The computational core of the algorithm is the use of one cell of the partition, called the **inducing cell**, to attempt to split every other cell. The innovation of *saucy* [7] is to use sparse data structures for the graph and intermediate refinement state. The algorithm described in [7] can be optimized further: rather than attempt to split all cells of the partition, only try to split those which are actually connected to the inducing cell. The bookkeeping required for this optimization is minimal and the performance gains are substantial on sparse graphs.

This work introduces a further refinement optimization, again targeted at sparse graphs. Let S be the inducing cell. Refinement in [7] works by first computing $k(v,S)$, the number of connections every vertex v in the graph has to S. Then, for each cell T of the partition with at least one vertex connected to S, T is partitioned based on the values of k. This partitioning of T is accomplished by count sorting. The present innovation is to avoid doing $O(|T|)$ work in the case of almost all elements of T having no connections to S, which is frequently the case in very sparse graphs. This is done by pre-sorting: as our algorithm discovers that elements of T are connected to S, it moves them to one end of the representation of T, so that when it count-sorts T, it only needs to sort those elements which have nonzero k. This optimization alone accounts for over one-hundredfold speedups on some benchmarks.

2.2 Problem Decomposition

Suppose we refine a vertex partition π, and the resulting partition π' is **discrete**—that is, every vertex is in its own cell. Only one permutation respects such a π', namely ι, the identity permutation. The identity is trivially a symmetry of the graph. In this case we are done: $\mathrm{Aut}(G)_\pi = \{\iota\}$, and no generators are returned.

If π' is not discrete, then there is at least one cell in π' with more than one element. Let T be such a cell; we denote it the **target cell**. Suppose $T = \{v_1, \ldots, v_j\}$. We know that for any symmetry γ of G which respects π', $v_1^\gamma = v$ for some $v \in T$. The contrapositive is perhaps more telling: for all symmetries γ, $v_1^\gamma \neq v$ for all $v \notin T$. Therefore, the image of v_1 is an equivalence relation on $\mathrm{Aut}(G)_{\pi'}$, that induces a partition of $\mathrm{Aut}(G)_{\pi'}$ into j subsets of symmetries. Note that some of these subsets may be empty: despite the best efforts of vertex partition refinement, some elements in the same cell of π' may not actually be symmetric under π'.

Consider first the subset of permutations γ such that $v_1^\gamma = v_1$; that is, the permutations which map v_1 onto itself. We can express this subset with vertex partitions: let $\widehat{\pi} = \pi'$, except with T replaced with $\{v_1\}$ and $T - \{v_1\}$. We say that $\widehat{\pi}$ is formed from π by **distinguishing** v_1. We have thus created a subproblem of our original problem: to find generators for $\mathrm{Aut}(G)_\pi$, we first find generators for $\mathrm{Aut}(G)_{\widehat{\pi}}$. The recursion terminates when the partition finally becomes discrete. After computing $\mathrm{Aut}(G)_{\widehat{\pi}}$, we then search for additional symmetries respecting π that map v_1 onto each of the other elements of T.

In fact, we only need to search for a single symmetry, for each $v \in T - \{v_1\}$, that maps v_1 onto v. This restriction of the search to only single representatives is necessary for producing a polynomially-bounded number of generators; further pruning discussed in Section 2.4 brings the bound down to linear. The justification for this restriction requires a modest foray into group theory. The set of symmetries $\mathrm{Aut}(G)_\pi$ is actually a **group**: a set closed under an associative binary operation (composition), that also contains an identity element (ι) and the inverse of every element. The group $\mathrm{Aut}(G)_{\widehat{\pi}}$ found in the subproblem discussed above is a **subgroup** of $\mathrm{Aut}(G)_\pi$. A subgroup B of a group A gives rise to a natural partition of A into equally-sized **cosets**. An elementary theorem of group theory states that the entirety of each coset of B can be generated by composing a single **representative** of each coset with the elements of B.

Consider again our chain graph example, with partition $\pi' = \{\{1,5\},\{2,4\},\{3\}\}$. Suppose we select $T = \{1,5\}$ as the target cell, and select vertex 1 as our v_1. Then we form a new partition $\widehat{\pi} = \{\{1\},\{3\},\{5\},\{2,4\}\}$ and find the generators of the graph under $\widehat{\pi}$. Refining $\widehat{\pi}$ yields a discrete partition, because, for example, vertex 2 has one connection to the cell $\{1\}$, while vertex 4 has none. Since $\widehat{\pi}$ is discrete, $\mathrm{Aut}(G)_{\widehat{\pi}}$ is simply the identity. We are left with the search for a symmetry γ which respects π' such that $1^\gamma = 5$.

2.3 Search

Suppose we have a target cell $T = \{v_1, v_2, \ldots\}$, and we are seeking a representative of the coset containing symmetries γ such that $v_1^\gamma = v_2$. This coset may not exist; that is, there may be no such γ, in which case we must prove that no such γ exists. To this end, we employ a backtracking search over partial permutations, much in the style of a SAT solver searching over partial variable assignments.

A **partial permutation** ρ is an isomorphism between two partitions of vertices. Let π_1 and π_2 be isomorphic vertex partitions, such that $\pi_1^\rho = \pi_2$. We denote ρ as partial because its induced mapping on vertices may not be well defined. Let $S_1 \in \pi_1$ and $S_2 \in \pi_2$ such that $S_1^\rho = S_2$. If $S_1 = \{v_1\}$ and $S_2 = \{v_2\}$, then the induced vertex mapping is explicit: $v_1^\rho = v_2$. If S_1 and S_2 are nonsingletons, however, the induced mapping is not necessarily defined; each $v_1 \in S_1$ could possibly map to any $v_2 \in S_2$, and thus further decisions must be made in the search. A partial permutation therefore implicitly represents a set of permutations.

Again, suppose we have an equitable partition π, with some target cell $T = \{v_1, v_2, \ldots\}$ selected, and we are seeking a γ such that $v_1^\gamma = v_2$. We construct a partial permutation ρ by creating two new partitions: π_1, with v_1 distinguished from the rest of T, and π_2, with v_2 distinguished from the rest of T. Finally, we define ρ such that $\{v_1\}$ within π_1 maps to $\{v_2\}$ within π_2, $T - \{v_1\}$ within π_1 maps to $T - \{v_2\}$ within π_2, and all the other cells of π_1 are mapped to their equivalent cell within π_2.

Just as a SAT solver propagates Boolean constraints imposed by assignments made to variables, a partial permutation becomes more completely specified through propagation of the constraints imposed by mappings assigned to vertices, such as the mapping of v_1 to v_2 above. In symmetry discovery, this propagation is performed by partition refinement, as discussed in Section 2.1. Both π_1 and π_2 above are candidates for refinement, since by splitting the target cell in each they are no longer necessarily equitable. We refine them simultaneously, so that every time cells within them are divided, we update the isomorphism ρ. Cells $S_1 \in \pi_1$ and $S_2 \in \pi_2$ **divide isomorphically** if, given corresponding inducing cells T_1 and T_2, S_1 has the same number of vertices with no connections to elements in T_1 as S_2 has vertices with no connections to T_2, and so on for each possible number of connections.

Consider again our chain graph example, which we left with $\pi = \{\{1,5\},\{2,4\},\{3\}\}$ and the need to search for some γ such that $1^\gamma = 5$. We form a pair of vertex partitions derived from π and prepare an isomorphism between them. We can represent this isomorphism as an ordering imposed on the cells of π_1 and π_2.

$$\begin{aligned}\pi_1 &= (\{1\},\{5\},\{2,4\},\{3\})\\ \pi_2 &= (\{5\},\{1\},\{2,4\},\{3\})\end{aligned}$$

These partitions are not equitable, and so we perform refinement on them simultaneously. In the case of π_1, we divide $\{2,4\}$ based on its connections to $\{1\}$. Thus, in the case of π_2, we perform an isomorphic division of $\{2,4\}$ using $\{1\}^\rho$, or $\{5\}$.

$$\begin{aligned}\pi_1 &= (\{1\},\{5\},\{4\},\{2\},\{3\})\\ \pi_2 &= (\{5\},\{1\},\{2\},\{4\},\{3\})\end{aligned}$$

Note that $\{2\}$ and $\{4\}$ are ordered differently in π_1 and π_2: in the case of π_1, vertex 4 has no connections to the inducing cell $\{1\}$, while in π_2, it is vertex 2 that has no connections to the corresponding incuding cell $\{5\}$.

After refinement completes, we examine the isomorphism ρ. There are a number of cases to consider. First, it may be the case that, during refinement, some cell in π_1 does not divide isomorphically to its corresponding cell in π_2. In this case ρ is no longer well defined, and thus it cannot induce a vertex symmetry and we must backtrack.

Suppose ρ is still well defined, and $S_1 = S_2$ for each nonsingleton $S_1 \in \pi_1$ and $S_2 \in \pi_2$ where $S_1^\rho = S_2$. That is, π_1 and π_2 only differ in their singletons. In this case, ρ induces a well defined permutation γ of vertices: $v_1^\gamma = v_2$ whenever $\{v_1\}^\rho = \{v_2\}$, and $v^\gamma = v$ otherwise. We can then check if $G^\gamma = G$; if so, we have found a symmetry and can thus terminate the search. If not, again we must backtrack. In the case of our example, both π_1 and π_2 are discrete, and thus ρ induces the vertex permutation $(1,5)(2,4)$, which turns out to be a symmetry of the chain graph.

Otherwise, π_1 and π_2 differ in at least one pair of corresponding nonsingletons, and thus we need to make another mapping decision. We select some nonsingleton cell $T_1 \in \pi_1$, and hence also select $T_1^\rho = T_2$ in π_2, which we again denote as target cells. We also select a single vertex $v_1 \in T_1$. By construction, any symmetry must map v_1 to some element of T_2. Thus, the size of the target cells is the branching factor at each point in the decision tree. We select some element $v_2 \in T_2$, construct the candidate mapping $\{v_1\}^\rho = \{v_2\}$, and propagate as before.

When we backtrack to a decision point, we simply find some candidate mapping that we had not yet considered, and continue. If we have exhausted all of the candidate mappings, we backtrack again. If we backtrack from the root of our search tree, then our search ends without finding a symmetry.

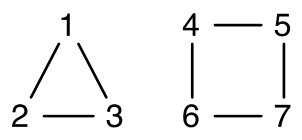
2.4 Pruning

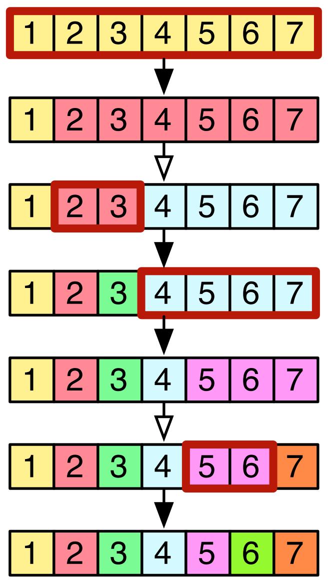
There is an additional way of eliminating redundancy in symmetry discovery: so-called orbit pruning. The following results are stated for completeness and without proof, since the mechanisms employed by our algorithm are essentially unchanged from *nauty*; see [10] for a thorough discussion.

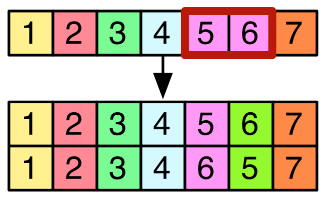
The **orbit partition** $\theta(\Gamma)$ of a permutation group Γ is a partition of the ground set of Γ such that, if x and y are in the same cell of $\theta(\Gamma)$, then there exists some $\gamma \in \Gamma$ such that $x^\gamma = y$. For instance, the orbit partition of the group of symmetries of our example chain graph is

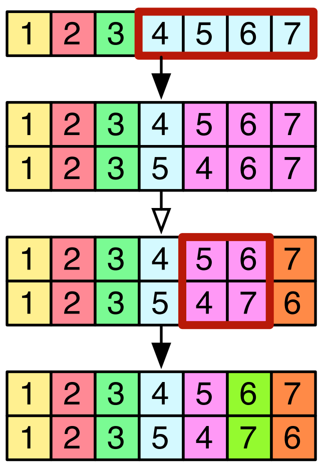
$$\theta(\{\iota, (1,5)(2,4)\}) = \{\{1,5\},\{2,4\},\{3\}\}$$

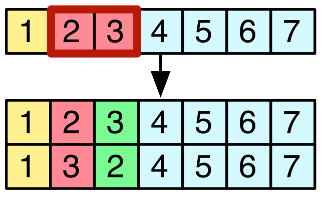
Orbit pruning can be employed within the subproblem decomposition as discussed in Section 2.2. Let π be a vertex partition with some nonsingleton cell $T = \{v_1, v_2, \ldots\}$ chosen as the target cell. Suppose we form $\widehat{\pi}$ by distinguishing v_1, and that we have already computed $\mathrm{Aut}(G)_{\widehat{\pi}}$. We

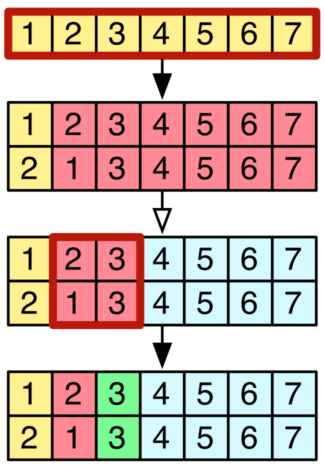

(a) The graph analyzed for symmetry


(b) Problem decomposition. Partitions are represented as colorings; elements in the same cell of the partition are given the same color, and are grouped together. Target cells are denoted by bold boxes surrounding a color. Partitions with a target cell are equitable. Shaded arrows represent partitions formed by distinguishing some element in a target cell. Open arrows represent refinements of non-equitable partitions. The initial partition is unit; the final partition is discrete, and represents the identity symmetry.


(c) First subproblem: consider symmetries γ such that $5^\gamma = 6$. Considering this partial permutation yields the isomorphic partition pair above, and corresponds to the complete permutation $(5, 6)$, which is a symmetry of the graph.


(d) Second subproblem: finding representatives of the cosets of the symmetry group found in part (c). These cosets are found by considering mappings γ such that $4^\gamma = x$ for $x \in \{5, 6, 7\}$. We first consider $4^\gamma = 5$, which yields the equitable partition pair above (with a boxed target cell pair). Since there exists a nonsingleton cell pair with unequal sets of elements, we must select a target cell, distinguish elements, and proceed with refinement. This yields the discrete partition pair corresponding to the complete permutation $(4, 5)(6, 7)$, which again is a symmetry. Orbit pruning eliminates the need to look for representatives of the other two cosets ($4^\gamma = 6$ and $4^\gamma = 7$).


(d) Third subproblem: find γ such that $2^\gamma = 3$ given the constraints of the top partition above. The partial permutation yielded by the partition pair is completely defined, because the partitions in the pair differ only in their singleton cells. Therefore, we need not continue to select target cells and refine partitions. Instead, we simply verify that $(2, 3)$ is in fact a symmetry of the graph. This is the most fundamental improvement described in this work.


(e) Fourth subproblem, part 1: we first consider finding a representative of the coset where $1^\gamma = 2$ for some γ. Refining the partition pair yields an equitable partition with a nonsingleton cell pair with different elements. Therefore, we proceed with selecting a target cell, distinguishing elements, and again refining. This yields a partition pair where, again, the only differences between the partitions are among their singleton cells. We immediately determine that $(1, 2)$ is a symmetry. The $1^\gamma = 3$ coset is pruned by orbit pruning.

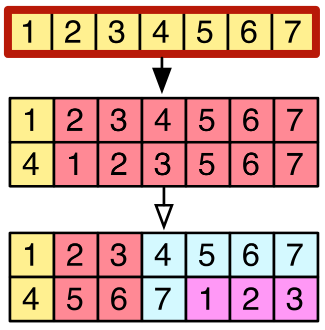

(e) Fourth subproblem, part 2: next, we consider the potential $1^\gamma = 4$ coset. Here, refinement fails: the partitions in the pair diverge. This indicates that our candidate mapping was incorrect, and we backtrack, in this case back to the root, and thus no symmetries exist which map 1 to 4. We avoid attempting to map 1 to 5, 6, or 7, again by orbit pruning.

Figure 1: A complete example of symmetry discovery.

now consider searching for a representative γ of the coset of $\mathrm{Aut}(G)_{\hat{\pi}}$ where $v_1^\gamma = v$ for all $v \in T - \{v_1\}$. As we find symmetries among these cosets, we update our orbit partition θ accordingly. When we consider a particular mapping $v_1^\gamma = v_k$, if v_1 and v_k are already in the same cell of θ, then it is not necessary to find such a γ, since we can already generate the entire coset.

The backtracking search for a symmetry can also use orbit pruning. However, it cannot use the running computation of θ, since there may be symmetries discovered earlier which do not respect the vertex partitions at a particular decision point. Therefore, a cache of found generators is kept, and when considering a candidate mapping, the orbit partition of those generators which respect the vertex partitions is constructed, and consulted as above. This work is only performed during backtracking, so that the common case of finding a symmetry without resorting to backtracking is not adversely affected.

3. COMPARISON TO PREVIOUS WORK

A key innovation in our work is the realization that symmetry discovery algorithms can be adapted to exploit sparsity not only in their input, graphs, but also in their output, the symmetries themselves. That is, if we presume that most symmetries leave the vast majority of the vertices fixed, we can squeeze more performance out of symmetry discovery.

The improvements proposed in our work are best seen in comparison to previously published algorithms for symmetry discovery. The search tree used by *saucy* was first described by McKay [10]; see the *bliss* paper [9] for an excellent exposition on the details.

A common attribute of *nauty*, *saucy*, and *bliss* is that symmetries are only discovered at leaf nodes of the search tree, where every path to a leaf node is as long as the original problem decomposition. In practice, this means that, after decomposition, the tools emit symmetries in rapid succession, but then appear to slow down, until the last few symmetries come out at a trickle for very large graphs. We modified *saucy* to explicitly check for the condition that all the differences between partitions existed only in the singletons, and short-circuited paths to leaf nodes when the condition held. Since the condition often becomes true very quickly on graphs with sparse symmetries, the number of nodes in the search tree essentially dropped from quadratic to linear with this change. This improvement alone reduced the time for symmetry discovery on some 100k-vertex CNF-derived graphs from a few days down to under a second.

The next step was to explicitly keep track of the vertices in the support of the permutations built by *saucy*. This made the short-circuit condition trivial to check, and also made the $G^\gamma = G$ predicate much faster to check for sparse permutations.

Another property of the search trees built by *nauty*, *saucy*, and *bliss* is that the target cell chosen at each node must be the same across each level of the tree. This constraint caused some graphs to take much longer than expected, because the cells with unresolved vertices would remain untouched while the target cells for many consecutive levels would be refined, uselessly, since they were equivalent anyway. This led to the elimination of the so-called distinguished leftmost decomposition, fundamental to those algorithms, and yielded the algorithm presented in Sections 2.2 and 2.3.

4. EXPERIMENTAL RESULTS

In order to evaluate our symmetry discovery approach, we compare it to *saucy*, *nauty*, and *bliss* on a variety of large, sparse graphs. Table 1 contains the results of our experiments. The *pipe* and *bug* graphs are constructed from CNF formulas representing pipelined microprocessor verification problems. The *adaptec* and *bigblue* graphs are derived from circuits used in the ISPD 2005 placement competition. The other graphs are simply very large graphs arising in other domains: *internet* represents the interconnections of major routers on the internet [6, 8], and *DE*, *ME*, *LA*, *IL*, and *CA* represent each state's road network [15].

For each graph, we report the number of its vertices and edges, and the size of its automorphism group. We also report the number of generators, and the average number of vertices in the support of (not fixed by) each generator. This number of generators is the number found by our algorithm; the other tools may have different numbers, since any given group of symmetries may have many different generating sets. However, the numbers are often the same, or otherwise only very slightly different, and so we omit similar values for the other tools.

Next, we report the number of partition refinements executed by our algorithm. We count each step in the problem decomposition as one refinement, and each decision point in each search for a symmetry as two refinements, since two partitions are refined side-by-side. We also report the time our algorithm takes to produce a complete set of generators.

Finally, we compare our algorithm to *saucy*, *nauty*, and *bliss*. We report the number of refinements performed by *saucy* only, again because the number is very similar among the three tools. We used *saucy*-0.5.2, *nauty*-2.4b7, and *bliss*-0.35 for the experiments. Note that *nauty*-2.4 includes a sparse graph representation option, which we used in our experiments.

All experiments were performed on a 2 GHz Core 2 Duo processor, with 1.5 GB memory and 4 MB cache, running a stock Fedora 7 desktop session. The tools were all compiled with *gcc*-4.1.2 with the `-O3` optimization flag.

All of the tools scale roughly linearly with the number of refinements performed during an execution. In the case of our algorithm, for most of the benchmarks this number is approximately equal to the total support of all generators found. Our algorithm thus scales well with sparse symmetries as well as sparse graphs. For the other tools, however, the number of refinements is quadratic in the number of generators. As mentioned in Section 3, this is due to the fact that symmetries are only discovered at the leaf nodes of the trees traversed by these tools. Our algorithm can find symmetries without such lengthy and costly traversals.

The *bug* benchmarks cause the symmetry discovery tools to perform a small number of refinements, but each refinement is potentially very expensive due to the number of vertices in these graphs. Here, our algorithm benefits from the refinement sparsity improvements discussed in Section 2.1, resulting in a considerable runtime improvement over the other tools.

We share the lament of the authors of [11]: we regret being unable to compare our algorithm with Puget's AUTOM [14], the latter being undocumented and not publicly available. However, extrapolating on the numbers reported in [14], we believe our algorithm outperforms AUTOM.

Benchmark						Our Algorithm		*saucy* [7]		*nauty* [10]	*bliss* [9]
Name	Vertices	Edges	$\lvert\text{Aut}(G)\rvert$	Gens	Supp/Gen	Refs	Time	Refs	Time	Time	Time
2pipe	3575	14625	10^{45}	38	97.00	149	**0.00**	2415	**0.01**	0.02	0.02
3pipe	10048	58556	10^{136}	84	131.31	333	**0.01**	13041	**0.06**	0.27	0.07
4pipe	21547	167942	10^{289}	152	161.04	605	**0.03**	43071	0.28	1.59	**0.23**
5pipe	38746	403799	10^{507}	239	189.36	953	**0.08**	108345	0.83	6.74	**0.63**
6pipe	65839	812525	10^{796}	346	223.34	1381	**0.15**	229503	2.22	23.87	**1.38**
7pipe	100668	1498971	10^{1158}	473	252.01	1889	**0.29**	431985	4.80	67.67	**2.84**
DE	51016	62087	10^{578}	1776	2.33	3789	**0.02**	1633827	**3.51**	515.04	27.63
ME	225114	246760	10^{3217}	9964	2.20	21011	**0.12**	51722042	**194.92**	time	time
LA	436535	524556	10^{4302}	12852	2.25	27211	**0.21**	85419592	**528.39**	time	time
IL	819138	1030719	10^{4843}	14999	2.24	31347	**0.43**	114958085	**958.80**	time	time
CA	1679418	2073394	10^{14376}	44439	2.28	93133	**0.84**	n/a	time	time	time
internet	284805	428624	10^{83687}	108743	2.28	374555	**0.41**	n/a	time	time	time
adaptec1	393964	919247	10^{7350}	15683	2.28	42879	**0.35**	123724315	**966.48**	time	time
adaptec2	471054	1052742	10^{10155}	21788	2.22	59459	**0.47**	n/a	time	time	time
adaptec3	800506	1846242	10^{15450}	36289	2.16	95653	**0.93**	n/a	time	time	time
adaptec4	878800	1880109	10^{24443}	53857	2.26	138797	**0.99**	n/a	time	time	time
bigblue1	508357	1127120	10^{11156}	22110	2.36	61095	**0.49**	n/a	time	time	time
bigblue2	1000790	2104613	10^{18385}	37031	2.27	104495	**1.09**	n/a	time	time	time
bigblue3	1876918	3812614	10^{52859}	98567	2.13	283351	**2.34**	n/a	time	time	time
bigblue4	3822980	8731076	10^{119182}	215132	2.22	620879	**5.36**	n/a	time	time	time
bug1	5706455	31011605	4	2	3704.00	5	**4.87**	6	**501.53**	1200.84	mem
bug2	5706326	31011322	4	2	3704.00	5	**4.78**	6	**528.21**	1197.79	mem

Table 1: Experimental results. All times are in seconds, with a time limit of 30 minutes and a memory limit of 1 GB. *Gens*: number of generators found by our algorithm. *Supp/Gen*: average support of each generator. *Refs*: number of partition refinements performed.

5. CONCLUSION

We have presented a new algorithm for symmetry discovery in sparse graphs, utilizing the sparsity of the symmetries themselves as they are discovered. This algorithm outperforms existing tools by many orders of magnitude, in some cases improving runtime from several days to a fraction of a second.

Future work will study the effects of different heuristics for target cell selection and within the search decision engine, and the feasibility of adapting our algorithm to the canonical labeling of graphs.

6. REFERENCES

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

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry breaking for boolean satisfiability. In *Design Automation Conference*, pages 836–839, 2003.

[3] 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.

[4] D. Chai and A. Kuehlmann. Building a better boolean matcher and symmetry detector. In *Design and Test in Europe*, 2006.

[5] K.-H. Chang, I. L. Markov, and V. Bertacco. Postplacement rewiring by exhaustive search for functional symmetries. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):Article 32, August 2007.

[6] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *USENIX Annual Technical Conference*, page 1, 2000.

[7] P. Darga, M. Liffiton, K. Sakallah, and I. Markov. Exploiting structure in symmetry detection for CNF. In *Design Automation Conference*, 2004.

[8] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *INFOCOM (3)*, pages 1371–1380, 2000.

[9] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *SIAM Workshop on Algorithm Engineering and Experiments*, 2007.

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

[11] C. Mears, M. Garcia de la Banda, and M. Wallace. On implementating symmetry detection. In *Sixth International Workshop on Symmetry in Constraint Satisfaction Problems*, 2006.

[12] A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3):Article 8, 2006.

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

[14] J.-F. Puget. Automatic detection of variable and value symmetries. volume 3709 of *LNCS*, pages 475–489, 2005.

[15] U.S. Census Bureau. UA Census 2000 TIGER/Line file download page, 2000. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`.