Conflict Anticipation in the Search for Graph Automorphisms

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

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

Abstract. Effective search for graph automorphisms allows identifying symmetries in many discrete structures, ranging from chemical molecules to microprocessor circuits. Using this type of structure can enhance visualization as well as speed up computational optimization and verification. Competitive algorithms for the graph automorphism problem are based on efficient partition refinement augmented with group-theoretic pruning techniques. In this paper, we improve prior algorithms for the graph automorphism problem by introducing *simultaneous refinement of multiple partitions*, which enables the anticipation of future conflicts in search and leads to significant pruning, reducing overall runtimes. Empirically, we observe an exponential speedup for the family of Miyazaki graphs, which have been shown to impede leading graph-automorphism algorithms.

1 Introduction

An *automorphism* (*symmetry*) of a graph is a *permutation* of the graph's vertices that preserves the graph's edge relation. The set of all symmetries of a graph forms a *group*[1] under functional composition. The *graph automorphism problem* seeks a generating set for the automorphism group of a graph. Closely related to graph automorphism is the problem of *canonical labeling* which assigns a unique signature to a graph that is invariant under all possible labelings of its vertices. Graph automorphisms and canonical labelings are related to the functional properties of the combinatorial objects in question. In a representative application developed in [3, 2], a CNF (conjunctive normal form) formula is modeled by a graph and passed to a symmetry detection program. During subsequent symmetry-breaking, these symmetries are used to augment the formula with a set of symmetry-breaking predicates. These predicates do not change the formula's satisfiability, but help SAT solvers prune away symmetric portions of the search space.

Graph symmetry and canonical labeling have been extensively studied over the past five decades. The **nauty** program [18, 19], developed by McKay in

[1] A group is an algebraic structure comprising a non-empty set of elements with a binary operation that is *associative*, admits an *identity* element, and is *invertible*. For example, the set of integers with addition forms a group. A *generating set* of a group is a subset of the group's elements whose combinations under the group operation generate the entire group.

1981, pioneered the first high-performance algorithms that inspired all subsequent tools. Almost two decades later, Darga et al [9] observed that the use of an adjacency matrix in **nauty** could lead to asymptotic inefficiencies in dealing with sparse graphs. This motivated the development of a new tool called **saucy** [9, 10, 16], which was limited to just finding a set of symmetry generators, but was three orders of magnitude faster than **nauty** on very large and very sparse graphs. Closely following **nauty**'s canonical labeling algorithms were two other tools, namely, **bliss** [13, 14] and **nishe** [22]. The search routines in **bliss** improved the handling of large and sparse graphs, and the branching heuristics in **nishe** facilitated a polynomial-time solution for the Miyazaki graphs [20], a family of graphs that **nauty** requires exponential time to process.

Since the emergence of the first version of **saucy** in 2004 (**saucy** 1.1) [9], different algorithmic enhancements improved **saucy**'s performance over a wide range of graphs with both theoretical and practical interest. The second version of **saucy** (**saucy** 2.0) [10] incorporated the observation that the symmetry generators of sparse graphs were mostly sparse. The major algorithmic changes that were introduced in **saucy** 2.0 separated the search for symmetries from the search for a canonical labeling. Further improvements to **saucy**'s data structures and algorithms were reported in **saucy** 2.1 [16].

In this paper, we present **saucy** 3.0 which performs *simultaneous partition refinement* to anticipate and avoid possible future conflicts. The procedure augments the method introduced in **saucy** 2.1 whereby nodes in the search tree represent sets of vertex permutations encoded by an ordered partition pair (OPP) of graph vertices. The basic idea of the new procedure is to refine the top and bottom partitions of an OPP at the same time, making sure that the two partitions conform to each other (according to the graph's edge relation) after each refinement step. We implemented this enhancement in **saucy** 3.0 and tested its performance on a wide variety of graph benchmarks. Our experimental evaluation shows that this enhancement can significantly prune the search tree for many graph families, such as the Miyazaki graphs. Furthermore, the concept of simultaneous refinement helps us better understand and explain the validity of some of the algorithms that were previously presented in **saucy** 2.1.

In the remainder, we first review some preliminaries in Section 2. Then, we discuss **saucy**'s baseline algorithms in Section 3. The new partitioning algorithm based on the concept of simultaneous refinement is presented in Section 4. Section 5 establishes the correctness of "matching OPP" pruning (this pruning mechanism was presented in **saucy** 2.1). The results of our experimental study are provided in Section 6. Finally, we discuss conclusions in Section 7.

2 Preliminaries

We assume familiarity with basic notions from group theory, including such concepts as groups, subgroups, group generators, cosets, orbit partition, etc. Information on different group theoretic concepts is available in many abstract algebra texts such as [11]. In this paper, we focus on the automorphisms of an

n-vertex *colored graph* G whose vertex is $V = \{0, 1, ..., n-1\}$. A *permutation* of V is a bijection from V to V, and a *symmetry* of G is a permutation of V that preserves G's edge relation. Permutation α, when applied to G, produces the permuted graph G^{α}. Every graph has a trivial symmetry, called the *identity*, that maps each vertex to itself. The set of symmetries of G forms a *group* under functional composition. This group is the *symmetry group* of G, and is denoted by $Aut(G)$. Given G, the objective of any symmetry detection tool is to find a set of *group generators* for $Aut(G)$.

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 called the *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 (OPP)* π is specified as

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

with π_T and π_B referred to, respectively, as the top and bottom ordered partitions of π. OPP 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).

OPPs lie at the heart of **saucy**'s symmetry detection algorithms, since each OPP compactly represents a set of permutations. This set of permutations might be empty (non-isomorphic OPP), might have only one permutation (discrete OPP), or might consist of up to $n!$ permutations (unit OPP). Several OPP examples and the permutation set encoded by them are provided below.

– Discrete OPP: $\begin{bmatrix} 2 \,|\, 0 \,|\, 1 \\ 1 \,|\, 2 \,|\, 0 \end{bmatrix} = \{(0\ 2\ 1)\}$
– 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: $\begin{bmatrix} 2 \,|\, 0,1 \\ 1 \,|\, 2,0 \end{bmatrix} = \{(1\ 2)\,, (0\ 2\ 1)\}$
– Matching OPP: $\begin{bmatrix} 1 \,|\, 0,2,4 \,|\, 3 \\ 3 \,|\, 0,2,4 \,|\, 1 \end{bmatrix} = (1\ 3) \circ S_3\,(\{0,2,4\})$
– Non-isomorphic OPPs: $\begin{bmatrix} 0,2 | \, 1 \\ 1 | \, 2,0 \end{bmatrix} = \emptyset$, $\begin{bmatrix} 2 | \, 0 | \, 1 \\ 1 | \, 2,0 \end{bmatrix} = \emptyset$

3 Baseline Algorithms

Similar to other combinatorial search algorithms, **saucy** explores the space of permutations by building a search tree and systematically traversing it. However, the representation of search nodes as OPPs in **saucy** is unique. The root of the tree is a unit OPP which is initially *refined* based on the colors and degrees of the vertices of the input graph. The depth-first traversal of the permutation space is started by choosing a *target* vertex from a non-singleton cell of the top partition and mapping it to all the vertices of the corresponding cell of the bottom partition. To propagate the *constraints of the graph* (i.e. the graph's edge relation), partition refinement is invoked after each mapping decision. The mapping procedure continues until the OPP becomes discrete, matching, or non-isomorphic (the latter is referred to as a *conflict*). In either case, **saucy** backtracks one level up, and maps the target vertex to the remaining candidate vertices. The search ends when all possible mappings are exhausted.

In addition to partition refinement, **saucy** exploits two types of pruning mechanisms: *group-theoretical* and *OPP-based.* To enable group-theoretical pruning, namely *coset* and *orbit* pruning, the left-most path of the tree should correspond to a sequence of *subgroup stabilizers* ending in the identity. In other words, the decisions along the left-most path maps each vertex to itself. This phase of the search is called *subgroup decomposition.* Note that no such requirement is needed in the remaining parts of the search tree. In contrast, OPP-based pruning mechanisms are optional techniques that assist **saucy**'s algorithms to avoid unnecessary search. Two of these techniques, embedded in **saucy** 2.1, are *non-isomorphic OPP* and *matching OPP* pruning.

In this paper, we introduce an enhanced partition refinement procedure that refines the top and bottom partitions of an OPP simultaneously. Our simultaneous refinement anticipates the conflicts that might arise in certain subtrees, and prunes the entire subtree without exploring it. The idea here is to capture conflicts that might be overlooked by the conventional refinement procedure.

4 Conflict Anticipation via Simultaneous Refinement

Partition refinement in **saucy** is adapted from **nauty**, and **nauty**'s refinement is based on the concept of *equitable* partitions. Partition $\pi = [W_1|W_2|\cdots|W_m]$ is equitable (with respect to graph G) if, for all $v_1, v_2 \in W_i$ $(1 \leq i \leq m)$, the number of neighbors of v_1 in W_j $(1 \leq j \leq m)$ is equal to the number of neighbors of v_2 in W_j. Although **saucy**'s partition refinement is adapted from **nauty**, the search tree in **saucy** is completely different from that in **nauty**. The nodes of **nauty**'s tree are **single ordered partitions**, while the nodes of **saucy**'s tree are **ordered partition pairs**. In **nauty**, an equitable partition is obtained by invoking partition refinement after each vertex *individualization.* Extending this to OPPs, the refinement procedure in **saucy** refines both partitions of an OPP *simultaneously* after each mapping decision, until 1) both partitions become equitable and the resulting OPP is isomorphic, or 2) the resulting OPP is non-isomorphic indicating an empty set of permutations, i.e., a conflict. In **saucy** 2.1 and earlier, simultaneous refinement was basically an algorithmic enhancement that detected conflicts (if any existed) earlier during refinement, without fully establishing an equitable OPP (an OPP whose top and bottom partitions are both equitable), and then examining the resulting OPP to see whether it

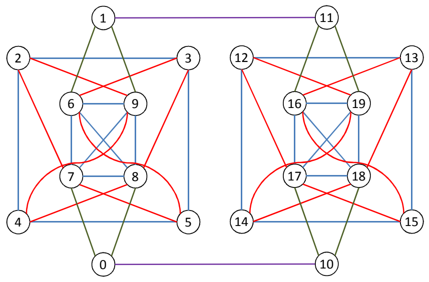

Fig. 1. A 20-vertex 46-edge graph with symmetry group of size 32.

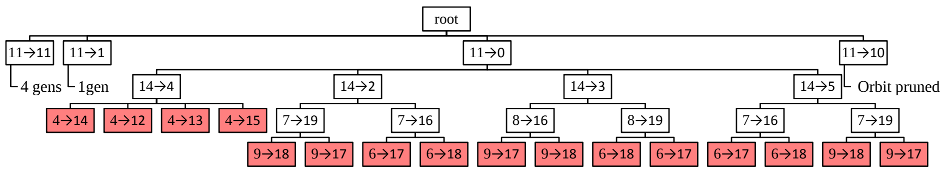

Fig. 2. The search tree constructed by **saucy** 2.1 for the graph in Figure 1.

was isomorphic/non-isomorphic. In implementation, **saucy** first refines the top partition until it becomes equitable, records where the cell splits occur, then starts refining the bottom partition, and compares the splitting locations of the bottom to the top whenever a new split occurs (i.e., checks the isomorphism of the two partitions after each split).

In this section, we argue that the significance of simultaneous refinement is not limited to the early detection of "non-isomorphic equitable OPPs". In particular, we demonstrate cases where the resulting equitable OPP is isomorphic, but the OPP still violates the edge relation of the graph. We illustrate such a case, and explain why conventional refinement fails to detect the conflict in that case. We then present an *enhanced simultaneous refinement* procedure that detects such cases and does not explore them. We discuss the impact of our proposed refinement procedure on the search tree constructed for our example.

Consider the 20-vertex 46-edge graph shown in Figure 1. The search tree generated by **saucy** 2.1 for this graph is shown in Figure 2. This search tree produces 16 conflicts (non-isomorphic OPPs), indicated by red-shaded nodes. In the remainder of this section, we focus on the path from the root that maps $11 \mapsto 0$ and then $14 \mapsto 4$. The OPPs in Figure 3a, labeled with (1), (2) and (3), represent the nodes of the search tree at the root, after mapping $11 \mapsto 0$, and after mapping $14 \mapsto 4$, respectively.

In **saucy** 2.1, the isomorphic OPP (3), obtained after mapping $14 \mapsto 4$, is not considered to be a conflict node and triggers further vertex mappings (namely, $4 \mapsto 14$, $4 \mapsto 12$, $4 \mapsto 13$, and $4 \mapsto 15$). However, this OPP violates the edge relation of the graph in Figure 1. To see this, consider the edge that connects 13 to 16. This edge, according to OPP (3), should be mapped to another edge

$$\left[\begin{array}{c|c|c} 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \\ 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \end{array}\right] \tag{1}$$

$$\left[\begin{array}{c|c|c|c|c|c|c|c|c|c} 0 & 10 & 1 & 11 & 14,12,13,15 & 2,4,5,3 & 17,18 & 8,7 & 6,9 & 16,19 \\ 11 & 1 & 10 & 0 & 4,3,5,2 & 13,14,12,15 & 9,6 & 19,16 & 18,17 & 7,8 \end{array}\right] \tag{2}$$

$$\left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} 0 & 10 & 1 & 11 & 13 & 12 & 15 & 14 & 2,4,5,3 & 17 & 18 & 8,7 & 6,9 & 16 & 19 \\ 11 & 1 & 10 & 0 & 3 & 2 & 5 & 4 & 13,14,12,15 & 6 & 9 & 19,16 & 18,17 & 7 & 8 \end{array}\right] \tag{3}$$

Fig. 3a. The search nodes of the tree in Figure 2. OPP (1) is at the root, OPP (2) is after mapping $11 \mapsto 0$, and OPP (3) is after mapping $14 \mapsto 4$.

$$[0|10|1|11|12,13,15|14|2,4,5,3|17,18|8,7|6,9|16,19] \tag{4}$$

$$[0|10|1|11|13|12,15|14|2,4,5,3|18|17|8,7|6,9|16|19] \tag{5}$$

$$[0|10|1|11|13|12|15|14|2,4,5,3|18|17|8,7|6,9|16|19] \tag{6}$$

Fig. 3b. The refinement of the top partition of OPP (2) to get OPP (3).

$$[11|1|10|0|3,5,2|4|13,14,12,15|9,6|19,16|18,17|7,8] \tag{7}$$

$$[11|1|10|0|3|5,2|4|13,14,12,15|9|6|19,16|18,17|7|8] \tag{8}$$

$$[11|1|10|0|3|2|5|4|13,14,12,15|9|6|19,16|18,17|7|8] \tag{9}$$

Fig. 3c. The refinement of the bottom partition of OPP (2) to get OPP (3).

that connects 3 to 7, since OPP (3) maps $13 \mapsto 3$, and $16 \mapsto 7$. Nevertheless, no such edge exists between 3 and 7 in Figure 1, and hence, OPP (3) is a conflict.

The question now is why the refinement procedure failed to detect the above conflict? Or, in other words, why was OPP (3) found to be isomorphic? To answer this question, we should follow the trace of the refinement procedure which is performed on OPP (2) to get OPP (3) after mapping $14 \mapsto 4$. As elaborated earlier, **saucy** first refines the top partition until it becomes equitable, then refines the bottom partition and checks the isomorphism of the bottom to the top whenever a new split occurs. The step by step refinement of the top and bottom partitions when $14 \mapsto 4$ is shown in Figure 3b and Figure 3c, respectively.

The refinement on the top starts by first making 14 a singleton cell (partition (4)). According to the graph of Figure 1, 14 is connected to 12,15,18 and 19, but not to 13, 17 and 16. Hence, refinement separates 12 and 15 from 13 (this makes 13 a singleton cell), 18 from 17, and 19 from 16 (partition (5)). The refinement continues by looking at the connections of one of the newly created cells. Here, **saucy** picks the singleton cell 16. According to the graph, 16 is connected to 11,13,15,17,18 and 19. This separates 15 from 12 (partition (6)). The top partition is now equitable, i.e., no further refinement is implied.

After refining the top partition, **saucy** starts refining the bottom partition. This is done by first making 4 a singleton cell (partition (7)). Since 4 is connected to 2,5,8 and 9, refinement separates 2 and 5 from 3 (this makes 3 a singleton

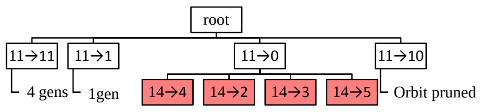

Fig. 4. The search tree constructed by **saucy** 3.0 for the graph in Figure 1.

cell), 9 from 6, and 8 from 7 (partition (8)). Note that, at this point, partition (8) is isomorphic to partition (5), i.e., no conflict is detected. This time **saucy** picks the singleton cell 7, since it had previously chosen 16 from the top, and 7 is at the same index on the bottom as 16 on the top. According to the graph, 7 is connected to 0,2,5,6,8 and 9. Since 7 is connected to both 2 and 5, no further refinement is implied. At this point, **saucy** should *detect the conflict that 16 on the top separated 15 from 12, but 7 on the bottom did not distinguish 2 from 5.* However, since no new cell is created on the bottom, **saucy** does not invoke the isomorphism check, and falsely assumes that the bottom stays isomorphic to the top. Note that the failure to detect this conflict is not a bug in refinement, since **nauty**'s (and essentially **saucy**'s) refinement procedure refines one partition at a time, and checks isomorphism once both partitions are equitable. After refining based on 7, **saucy** refines based on 6. Vertex 6 is connected to 1,3,5,7,8 and 9. Since 6 is connected to 5 but not 2, it separates 5 from 2 (partition 9). The bottom partition is now equitable and isomorphic to the top.

After the refinement procedure ends, **saucy** builds isomorphic OPP (3), and starts exploring it by mapping 4 to 14, 12, 13, and 15. However, this phase of the search is superfluous, since we know that OPP (3) violates the graph's edge relation, and its further exploration will always result in conflicts. Another case of a conflicting isomorphic OPP is when two corresponding *singleton cells* of the top and bottom partitions have different connections to the other *singleton cells* of their own partition. In this case, the conflict is again overlooked by **saucy**'s conventional refinement procedure, since singleton cells cannot be partitioned to smaller cells (i.e., no new cell splitting occurs), and hence, the top and bottom partitions remain isomorphic after this step of refinement.

To detect the conflicts that might remain undetected during partition refinement, we enhanced **saucy**'s partition refinement in two ways; 1) the isomorphism of the bottom partition to the top is checked *after each refinement step, rather than after each time a new split occurs*, and 2) in addition to the isomorphism check, we also ensure that *the connections of each newly created cell on the bottom match the connections of its corresponding cell on the top.* These two new checks verify that the top and bottom partitions remain isomorphic and conforming (according to the graph's edge relation) after each refinement step. In our implementation, the overhead of the first check is negligible, as it is performed within the main refinement loop, but the second check requires an extra iteration over the outgoing edges of the vertices of the newly created cells. We would like to emphasize that our enhancement is *enabled* by the OPP-encoding of permutations that is unique to **saucy**'s search for automorphisms.

Figure 4 shows the search tree for the graph in Figure 1 when our new simultaneous refinement is invoked. Comparing this search tree to that in Figure 2, the number of conflicts is reduced from 16 to 4.

5 The Validity of Matching OPP Pruning

When matching OPP π is encountered in the search, **saucy** "constructs" a permutation α from π by mapping the vertices in matching cells identically. It then uses α to prune the entire subtree rooted at this OPP in one of two ways; either 1) α is an automorphism of the graph, which means that the subtree is a coset of the stabilizer subgroup, and α is a coset representative, or 2) α is not an automorphism, which indicates that the subtree is not a coset, and the search for a coset representative in that subtree will always fail. In this section, we show that, if π is found to be matching by our enhanced simultaneous refinement (described in Section 4), the second case cannot occur, i.e., α must always be an automorphism of the graph. The proof of this claim is presented next.

Assume that π is an OPP that is found matching by our enhanced refinement procedure. This means that π is equitable, isomorphic, matching, and conforming according to G's edge relation. Let α be the permutation that corresponds to π, i.e., the permutation that maps the vertices in π's non-singleton cells identically. To show by contradiction that α is a symmetry of G, assume that it is not. Then, there must be an edge in G^α that does not exist in G (or vice versa). Assume that this edge connects v_1 to v_2. Trivially, both v_1 and v_2 cannot be mapped identically in α, otherwise, an edge between v_1 and v_2 in G would map to the exact same edge in G^α. Hence, permutation α either maps v_1 to v'_1 ($v_1 \neq v'_1$), or v_2 to v'_2 ($v_2 \neq v'_2$), or both. We first consider the case where v_1 is mapped to v'_1 but v_2 is mapped identically (this is similar to the case where v_2 is mapped to v'_2 but v_1 is mapped identically). This case contradicts our assumption that π is equitable, since v_1 and v'_1 were both singleton cells of π, and having an edge between v_1 and v_2 but not between v'_1 and v_2 would imply further refinement on π. Now consider the case where v_1 is mapped to v'_1 and v_2 to v'_2. This case contradicts our assumption that π is conforming according to G's edge relation, since v_1, v_2, v'_1 and v'_2 were all singleton cells of π, and having an edge between v_1 and v_2 but not between v'_1 and v'_2 would violate G's edge relation.

6 Experimental Evaluation

We implemented our simultaneous partition refinement technique in **saucy** 3.0, and tested its performance on 1445 graph benchmarks drawn from a wide variety of domains. Our 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. A time-out of 1000 seconds was applied. Table 1 lists the benchmark families used in our experiments. For these families, the name, the number of instances, the size of the smallest and largest instances, and a short description are provided. The families are divided into

Table 1. Benchmark families

Family	Instances	Smallest Instance		Largest Instance		Description
		vertices	edges	vertices	edges	
`mz` [20, 15]	25	40	60	1,000	1,500	Original Miyazaki graphs
`cmz` [15]	46	120	90	200	1,900	(`mz`), and their variants
`mz-aug` [15]	25	40	92	1,000	2,300	designed to mislead the
`mz-aug2` [15]	24	96	152	1,200	1,900	**bliss** cell selector
`circuit` [23, 1]	33	3,575	14,625	4,406,950	8,731,076	**saucy** benchmarks from
`router` [7, 12]	3	112,969	181,639	284,805	428,624	place-route, verification,
`roadnet` [6]	56	1,158	1,008	1,679,418	2,073,394	routers & road networks
`application` [8]	300	464	2,066	32,813,545	65,487,132	SAT 2011 application,
`crafted` [8]	300	105	320	776,820	3,575,337	crafted and random
`random` [8]	600	1,165	5,375	310,000	680,000	CNF instances
`binnet` [17, 4]	33	1,000	720	6,000,000	4,391,515	binary networks

four categories. These categories were chosen based on the general construction of the graphs, considering metrics such as the number of vertices and edges, connectivity and sparsity. The first category is the Miyazaki graphs [20, 15], which **nauty** takes exponential time to process. The second category contains benchmarks used to test earlier versions of **saucy**. It represents graphs from various domains, such as logic circuits and their physical layouts [23, 1], internet routers [7, 12], and road networks in the US states and its territories [6]. The third category includes CNF benchmarks from the international SAT 2011 competition [8]. The fourth category consists of graphs not previously reported in graph automorphism or satisfiability research. These graphs were proposed for testing community-detection algorithms [17, 4] [2].

Figure 5 compares the number of conflicts produced by **saucy** 3.0 and **saucy** 2.1. If a benchmark is not processed within the time-out, the number of conflicts encountered right before termination is reported. The results show that **saucy** 3.0 always produces fewer or the same number of conflicts. This is expected, as our proposed refinement procedure anticipates and avoids certain conflicts that might arise in **saucy** 2.1. Of all the benchmark families, `mz-aug` and `mz-aug2` benefit most from the new refinement procedure. For these two families, the highest number of conflicts reported by **saucy** 3.0 was 696 (for `mz-aug-50`). In contrast, the number of conflicts reported by **saucy** 2.1 was at least 10,000 for 46 out of 49 `mz-aug` and `mz-aug2` instances. Of the remaining two Miyazaki families, `mz` did not experience any change in its number of conflicts, and `cmz` showed a slight improvement for 5 out of its 46 instances (8 fewer conflicts were reported for those 5 instances). Of the graphs from circuits, internet routers, and road networks, only one instance (from `circuit`) showed significant conflict

[2] We used the implementation of the algorithm described in [17] (available at [4]) to generate 33 undirected and unweighted binary networks. We set the number of nodes to $\{1, ..., 9\} \times \{10^3, 10^4, 10^5\}$ and $\{1, ..., 6\} \times 10^6$ (generating larger networks required more than 8GB RAM), and fixed the remaining parameters in all instances. Specifically, we set the average degree to 2, the max degree to 4, the mixing parameter to 0.1, the minimum community size to 20, and the maximum community size to 50.

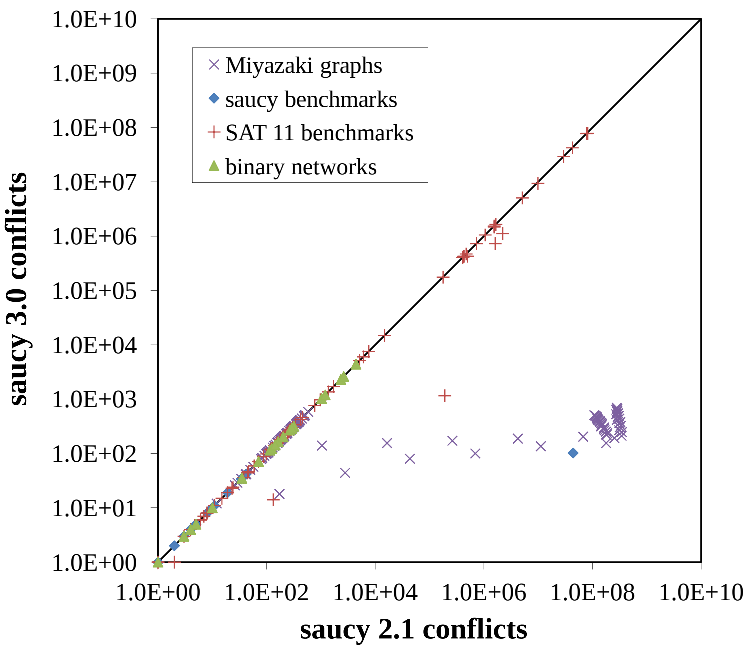

Fig. 5. Number of conflicts returned by **saucy** 3.0 versus **saucy** 2.1.

reduction (from 43 million to only 102). The remaining instances produced the same number of conflicts (not more than 42) in **saucy** 3.0 and **saucy** 2.1. Of the 1200 CNF benchmarks, only 72 (15 from application and 57 from crafted) encountered conflicts in **saucy** 2.1, and only 12 (all from crafted) experienced a reduction in the number of conflicts. The smallest reduction was 1 and the largest was 2.9 million. The `binnet` instances also produced the same results in both versions of **saucy**. The reported number of conflicts for those instances ranged from no conflicts to 4,412.

Figure 6 shows the distribution of *depth* of the conflicts that were captured and avoided by **saucy** 3.0. Recall that the new refinement procedure in **saucy** 3.0 prunes some subtrees that are explored by **saucy** 2.1. Suppose that one such subtree is found to be conflicting at level l in **saucy** 3.0, but leads to c conflicts in **saucy** 2.1, where the n-th conflict ($1 \leq n \leq c$) occurs at level l_n. Trivially, $l_n \geq l$. We define the depth of the n-th conflict as $d = l_n - l$. If $d = 0$, both **saucy** 3.0 and **saucy** 2.1 capture the conflict at the same time. If $d > 0$, **saucy** 3.0 anticipates and avoids the conflict d levels sooner than it occurs in **saucy** 2.1. We use conflict depth as a numeric criterion to evaluate the effectiveness of our new refinement procedure. The results in Figure 6 show that the deepest conflicts captured by **saucy** 3.0 occur in the instances of Miyazaki families. The greatest reported depth was 98, which occurred 2.8×10^8 times for `mz-aug-50`. The only benchmark from the `circuit` family that had significant

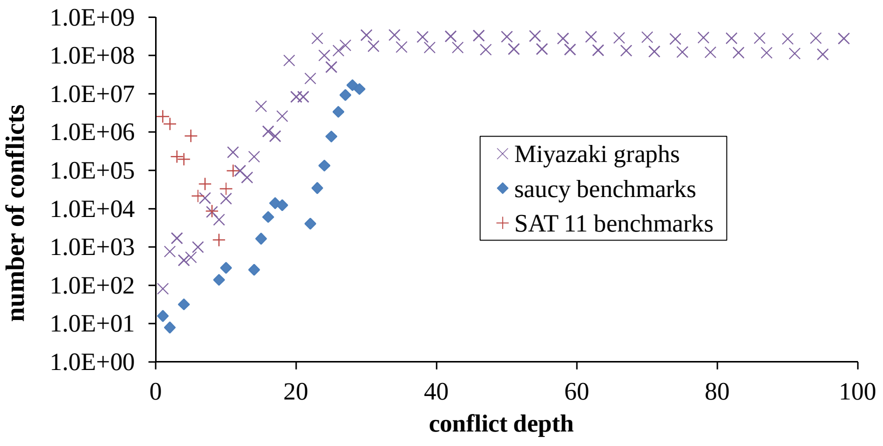

Fig. 6. Histogram of the conflict depths captured by **saucy** 3.0.

conflict reduction produced conflict depth of up to 29, where the largest conflict depth happened 1.3×10^7 times. For the CNF benchmarks, the deepest reported conflict had a depth of 11, and occurred roughly 10^5 times. The histogram in Figure 6 excludes the results for binary networks, since all those conflicts were reported at depth 0.

The runtime comparison between **saucy** 3.0 and **saucy** 2.1 is depicted in Figure 7. For the families of `mz-aug` and `mz-aug2`, we observed an exponential speedup when our proposed refinement procedure was invoked. Of the 49 instances in these two families, **saucy** 3.0 solved all in less than a second, while **saucy** 2.1 failed to process 39 within the time-out limit. For the `mz` and `cmz` families, **saucy** 2.1 and 3.0 had comparable runtimes. The instances of `router`, `roadnet`, and `binnet` did not experience much change either. For the `circuit` family, the results were comparable, except for one benchmark that was solved by **saucy** 3.0 in a second but remained unsolved in **saucy** 2.1. Interestingly enough, we did not observe any major improvement in the runtimes of the SAT 11 CNF benchmarks, although conflict reduction of up to 2.9 million was reported for some of those instances. Our further analysis revealed that high reduction in the number of conflicts was reported for instances that timed out in both **saucy** 3.0 and **saucy** 2.1, and the reduction in the remaining instances was not significant enough to reflect a major improvement in runtimes. Note that the runtimes reported in Figure 7 match with the number of conflicts reported in Figure 5. In fact, fewer conflicts generally led to better runtimes.

In order to evaluate the performance of **saucy** 3.0 versus state-of-the-art graph automorphism tools, we ran **bliss** (version 0.72, available at [5]) on all the 1445 benchmarks listed in Table 1, and compared its runtimes to those obtained from **saucy** 3.0. This comparison is shown in Figure 8. Of the four Miyazaki graph families, **bliss** showed difficulties in processing the instances of `cmz` (took up to 856 seconds to complete all those instances), but processed the

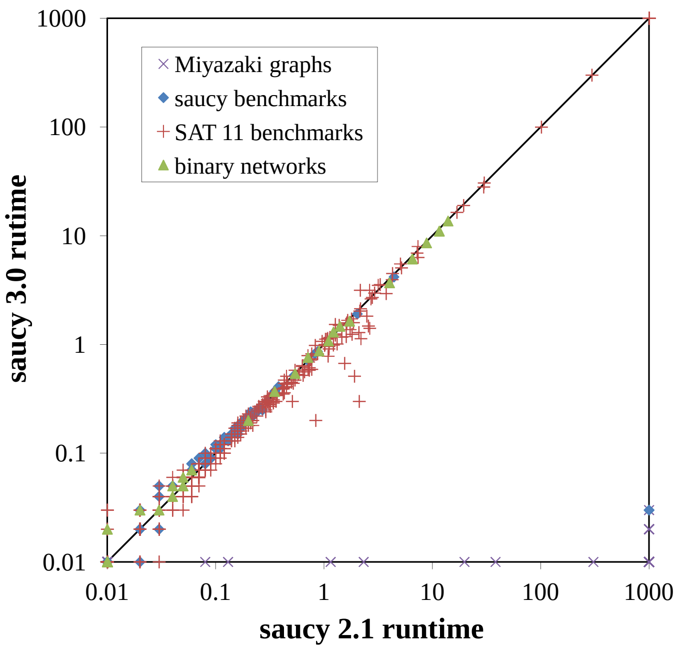

Fig. 7. Runtime of **saucy** 3.0 versus **saucy** 2.1 (timeout is 1000 seconds).

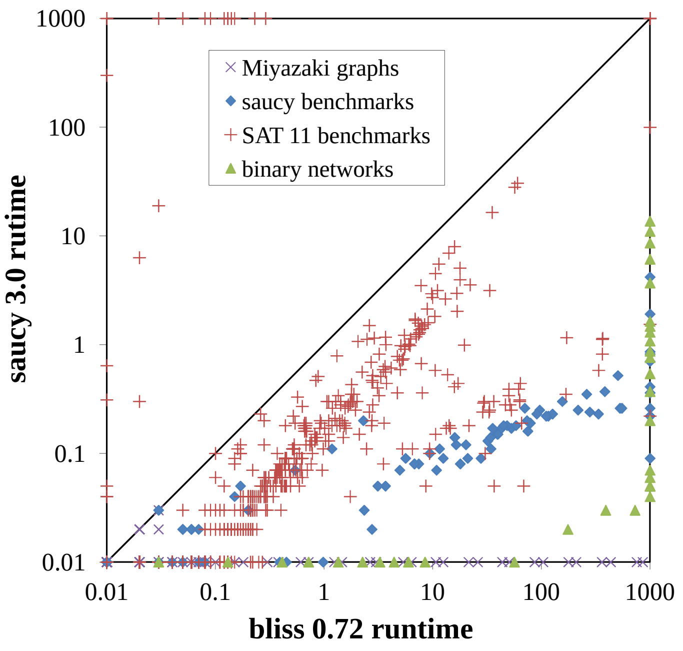

Fig. 8. Runtime of **saucy** 3.0 versus **bliss** 0.72 (timeout is 1000 seconds).

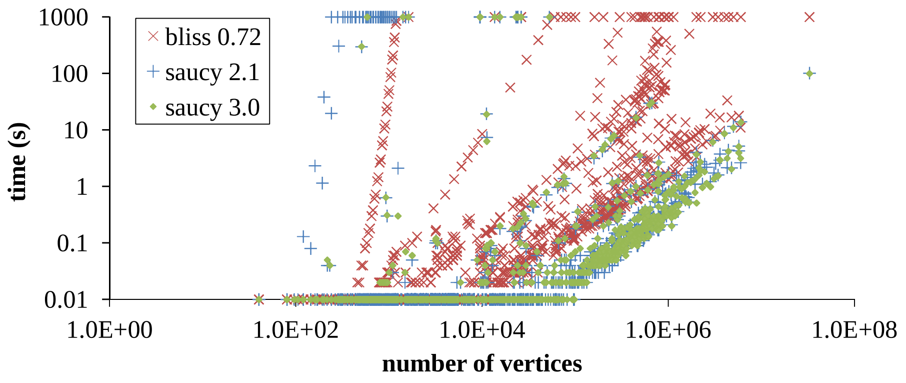

Fig. 9. Runtimes of **saucy** 3.0, **saucy** 2.1, and **bliss** 0.72 as a function of graph size.

remaining three families in less than a second. In contrast, **saucy** solved all Miyazaki graphs in less than a second. Furthermore, **bliss** timed out on 8 and 3 out of 33 and 56 instances of the `circuit` and `roadnet` families, respectively, but solved the remaining instances of those two families and all 3 instances of `router` in 550 seconds. This was while **saucy** solved all the 92 instances of these three families in 5 seconds (processed 90 in less than a second). For the CNF benchmarks, **saucy** and **bliss** showed mixed results. Of the 600 crafted and application instances, **bliss** failed to process 4 crafted and 3 application instances, whereas, **saucy** failed to process 17 crafted instances, but solved all application instances. The 4 crafted benchmarks that were unsolved by **bliss** were also unsolved by **saucy**. This means that **bliss** solved 13 crafted instances that **saucy** failed to process, and **saucy** solved 3 application instances that **bliss** did not solve. Of the remaining crafted and application benchmarks, **bliss** solved 541 in less than 10 seconds, and 52 in 366 seconds, while **saucy** solved 577 in less than 10 seconds, and 6 in 300 seconds. Both **saucy** and **bliss** solved all random benchmarks in less than a second. Overall, the results in Figure 8 indicate that **saucy** outperformed **bliss** on the majority of SAT 11 benchmarks. For binary networks, **saucy** consistently produced better results. Specifically, **saucy** solved all 33 instances of `binnet` in 14 seconds (the largest runtime was 13.67 seconds which was reported for the largest instance of this family with 6×10^6 vertices), but **bliss** timed out on 19, and solved the remaining in 727 seconds.

As part of our study, we also ran **nishe** 0.1 [21] on all the graph benchmarks in our suite, and compared its results to **saucy** 3.0. In general, we observed that the runtimes of **nishe** and **saucy** were comparable for the Miyazaki graphs. For the remaining benchmarks, however, **nishe** exhibited poor performance compared to **saucy** and **bliss**. In particular, it failed to process (either timed out or had a segmentation fault) 59 out of 92 **saucy** benchmarks, 950 out of 1200 CNF instances, and 24 out of 33 binary networks.

Figure 9 shows the runtimes of **saucy** 3.0, **saucy** 2.1, and **bliss** 0.72 as a function of graph size for all the 1445 benchmarks listed in Table 1. As this figure suggests, the smaller instances seem to be more challenging for **saucy**. This is particularly not true of **bliss**, as **bliss** tends to produce larger runtimes for larger instances. The smallest instance that **saucy** 3.0 timed out on had 583 vertices, and the largest had 52,786 vertices, while these numbers were respectively reported to be 1,620 and 33 million for **bliss** 0.72. Of the 446 benchmarks with more than 52,786 vertices, **saucy** 3.0 solved 389 in less than a second, and processed the rest in 100 seconds, while **bliss** 0.72 solved 213 in less than a second, took up to 550 seconds to process 200, and timed out on 33. On the other hand, of the 999 benchmarks that had less than 52,786 vertices, **saucy** 3.0 solved 979 in less than a second, timed out on 17, and took up to 550 seconds to process the rest, whereas, **bliss** 0.72 processed 946 in less than a second, timed out on 4, and processed the remaining in 856 seconds. To investigate the reason why **saucy** 3.0 did not perform as expected on relatively small instances, we examined the effect of different decision heuristics on the 17 benchmarks that **saucy** failed to process. Interestingly, 4 out of those 17 benchmarks were solved in less than a second with an alternative decision heuristic. Of those 4, one was reported to be unsolved by **bliss** 0.72. These results suggest that branching decisions play a crucial role in minimizing the time for automorphism search. We plan to pursue the effect of decision heuristics in our future research.

7 Conclusions

In this work, we have advanced the state of the art in algorithms for solving graph automorphism, which finds applications in many fields. Our technique takes advantage of a unique feature in the **saucy** algorithm — the representation of partial permutations (search nodes) in terms of ordered partition pairs. Previously, these partitions were refined one at a time, but we have now developed *simultaneous partition refinement*, which allows **saucy** to anticipate possible future conflicts and prune the search tree early. This optimization significantly improves runtime on several benchmark families, including the ones suggested by Miyazaki [20] for further study because **nauty** provably requires exponential time on these benchmarks. Our empirical comparisons show that our implementation **saucy** 3.0 outperforms the competition on most available benchmarks. Our ongoing work is focused on several benchmarks where **saucy** 3.0 is outperformed by **bliss** 0.72. Preliminary analysis suggests that these benchmarks tend to be small, which may be due to subtle inefficiencies in our implementation rather than asymptotic bottlenecks. We hope that our future research will shed additional light on this.

References

1. ISPD 2005. *http://archive.sigda.org/ispd2005/contest.htm.*
2. Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proc. 40th IEEE/ACM Design Automation Conference (DAC)*, pages 836–839, Anaheim, California, 2003.

3. Fadi A. Aloul, Arathi Ramani, Igor Markov, and Karem A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. 39th IEEE/ACM Design Automation Conference (DAC)*, pages 731–736, New Orleans, Louisiana, 2002.
4. binary networks. *https://sites.google.com/site/santofortunato/inthepress2.*
5. bliss 0.72. *http://www.tcs.hut.fi/Software/bliss/bliss-0.72.zip*, 2011.
6. U. S. Census Bureau. *http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html.*
7. Bill Cheswick, Hal Burch, and Steve Branigan. Mapping and visualizing the internet. In *USENIX Annual Technical Conference*, pages 1–13, 2000.
8. SAT Competition. *http://www.satcompetition.org.*
9. Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *Proc. 41st IEEE/ACM Design Automation Conference (DAC)*, pages 530–534, San Diego, California, 2004.
10. Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. 45th IEEE/ACM Design Automation Conference (DAC)*, pages 149–154, Anaheim, California, 2008.
11. John B. Fraleigh. *A First Course in Abstract Algebra.* Addison Wesley Longman, Reading, Massachusetts, 6th edition, 2000.
12. Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM*, pages 1371–1380, 2000.
13. Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 07)*, New Orleans, LA, 2007.
14. Tommi Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In *Proceedings of the First international ICST conference on Theory and practice of algorithms in (computer) systems*, TAPAS'11, pages 151–162, Berlin, Heidelberg, 2011. Springer-Verlag.
15. Petteri Kaski. *http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml.*
16. Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In *Proc. Satisfiability Symposium (SAT)*, Edinburgh, Scotland, 2010.
17. Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80:016118, Jul 2009.
18. Brendan D. McKay. nauty user's guide (version 2.2) *http://cs.anu.edu.au/˜bdm/nauty/nug.pdf.*
19. Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
20. Takunari Miyazaki. *The complexity of McKays canonical labeling algorithm*, page 239. Amer Mathematical Society, 1997.
21. nishe 0.1. *http://gregtener.com/media/upload/nishe-0.1.tar.bz2.*
22. Greg Tener and Narsingh Deo. Efficient isomorphism of miyazaki graphs. In *39th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, Boca Raton, FL, 2008.
23. Miroslav N. Velev and Randy E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proc. Design Automation Conference (DAC)*, pages 226–231, New Orleans, Louisiana, 2001.