2013 IEEE 25th International Conference on Tools with Artificial Intelligence

Conflict Analysis and Branching Heuristics in the Search for Graph Automorphisms

Paolo Codenotti*, Hadi Katebi*, Karem A. Sakallah, and Igor L. Markov
EECS Department, University of Michigan, Ann Arbor, MI
{paoloc, hadik, karem, imarkov}@umich.edu

***Abstract*—We adapt techniques from the constraint-programming and satisfiability literatures to expedite the search for graph automorphisms. Specifically, we implement conflict-driven backjumping, several branching heuristics, and restarts. To support backjumping, we extend high-performance search for graph automorphisms with a novel framework for conflict analysis. Empirically, these techniques improve performance up to several orders of magnitude.**

I. Introduction

An *automorphism* (or *symmetry*) of a graph is a permutation of the vertices that preserves the edge relation. The *graph automorphism problem* (GA for short) is the task of finding all the automorphisms of a given graph. In a number of diverse fields of science and engineering, such as computer networks [5, 12], electronic circuits [24, 14], and mathematical chemistry [4], information about the graph structure given by the automorphisms can help answer computational questions. For example, in computational chemistry, the automorphisms of molecules can be used to predict and explain chemical properties [13].

From the theoretical standpoint, it is not known whether GA is in P or is NP-hard. In fact, there are complexity theoretic reasons to believe that GA is not NP-hard [3]. Today, the best-known algorithm for GA runs in $e^{O(\sqrt{n \log n})}$ worst-case time [2].

The first practical algorithm (with a scalable software implementation) particularly optimized to solve GA on large application-derived graphs is `Saucy` [6, 7, 16], introduced in 2004. Prior algorithm such as `Nauty` [19], and subsequent efforts, such as `Bliss` [15], solve the more general problem of *canonical labeling* and report symmetries as a byproduct. A canonical labeling is an ordering of the vertices that uniquely captures the structure of the graph and serves as a permutation-invariant signature. Since graph automorphisms map each labeling to itself, the search for canonical labeling can be enhanced by finding automorphisms first and feeding them to canonical labeling routines [18].

The data structures and algorithms in `Saucy` take advantage of both the sparsity of input graphs and their symmetries to attain scalability. Furthermore, they exploit *simultaneous partition refinement* (a form of propagation through graph constraints) to anticipate and avoid conflicts during the search. The experimental results in [17] confirm that `Saucy` is currently the fastest symmetry-finding tool available for large and sparse graphs. Nevertheless, a few large sparse graphs and several other benchmarks remain challenging.

In this paper, we adapt techniques from the *constraint-programming* (CP) and *satisfiability* (SAT) literatures to expedite GA combinatorial search algorithms. Specifically, we augment `Saucy` with conflict-driven backjumping, different branching heuristics, and restarts. To perform backjumping, we analyze the conflicts encountered during the automorphism search. We reason through the chain of causes in partition refinement to identify a small set of decisions that are sufficient to expose the conflict. This is analogous to conflict analysis in CSP and SAT solvers, but the type of conflicts and the propagation procedure in GA are specific to the problem. Our experiments show that backjumping can be implemented with low overhead and significantly improves runtime for some benchmarks. In addition, we see that branching heuristics can have a huge effect on runtime, and no individual heuristic outperforms others on all instances. We further show that a combination of backjumping and a simple restart policy with different branching heuristics outperforms other symmetry-detection algorithms.

We start the remainder of the paper by discussing related work and introducing preliminaries in Section II. In Section III, we describe our conflict analysis framework and the backjumping procedure. We introduce heuristics in Section IV. We present the results of our experimental evaluation in Section V. In Section VI, we discuss ideas and obstacles to further applications of our conflict analysis framework.

II. Preliminaries and Related Work

A. Related Work

Junttila and Kaski introduced a new version of `Bliss` which used "conflict propagation based on recorded failure information" [15]. This is a similar idea to our conflict analysis at a high level; use information gained from conflicts to prune the search tree. However, the similarities end here. The approach of [15] is based on recording unique signatures for subtrees that failed to produce any automorphisms, and using those signatures to avoid searching isomorphic subtrees. Our approach, on the other hand, is based on conflict analysis that identifies the decisions that are responsible for a conflict. Computing the signatures is a by-product of

*P. Codenotti and H. Katebi contributed equally to this work.

the Bliss canonical-labeling tree and does not allow for dynamic branching (see Section IV). Nevertheless, as the two approaches prune the search tree in different ways, they may be combined.

B. Definitions and Notation

We assume familiarity with basic notions from group theory, including such concepts as groups, subgroups, group generators, cosets, orbit partition, etc. (see abstract algebra textbooks such as [11]).

We consider the automorphisms of an n-vertex *graph* G with vertex set $V = \{1, 2, \dots, n\}$ and edge set E. For a subset $X \subseteq V$ and a vertex v, we use $E(v, X) = \{(x, v) \in E \mid x \in X\}$ to denote the set of edges between v and X. 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. A permutation α of V, when applied to G, produces the permuted graph G^α. Every graph admits a trivial symmetry, called the *identity* and denoted by ι, that maps each vertex to itself. The set of symmetries of G forms a *group* under functional composition – the *symmetry group* of G 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

$$\Pi = \left[\begin{array}{c} \pi_T \\ \pi_B \end{array}\right] = \left[\begin{array}{c} T_1\,|T_2\,|\cdots|T_m \\ B_1\,|B_2\,|\cdots|B_k \end{array}\right]$$

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).

C. Baseline Algorithms for Graph Automorphism

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). 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. One can view the top and bottom partitions of an isomorphic OPP as two separate colorings where all the vertices in the same-index cells of the top and bottom partitions have the same color. This analogy can help us better understand constraint propagation within the Saucy framework.

All algorithms for GA explore the space of permutations by building a tree and systematically traversing it. However, the encoding of permutations by OPPs is unique to Saucy. The root of the Saucy tree is a unit OPP. Any node that encodes an isomorphic OPP is extended by choosing a *target* vertex v from a non-singleton cell of the top partition and *mapping* it to all the vertices $\{u_1, ..., u_k\}$ of the corresponding cell of the bottom partition. The mapping of v to u_i is accomplished by separating v from the top and u_i from the bottom such that v and u_i are in their own cells at the same index of the top and bottom partitions. 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 mappings are exhausted.

The Saucy permutation tree can be pruned significantly by performing *partition refinement* [19] before selecting and branching on a target vertex. The goal of partition refinement is to propagate the graph's vertex coloring and edge relation until the partition becomes *equitable*. Partition $\pi = [W_1|W_2|\cdots|W_t]$ is said to be equitable (with respect to a given graph) if, for all vertices $v_1, v_2 \in W_i$ $(1 \le i \le t)$, the number of neighbors of v_1 in W_j $(1 \le j \le t)$ is equal to the number of neighbors of v_2 in W_j (i.e., $|E(v_1, W_j)| = |E(v_2, W_j)|$). In Saucy, partition refinement is applied *simultaneously* to the top and bottom partitions [16]. If the top and bottom partitions refine differently, Saucy concludes that the current OPP yields no symmetry and prunes the entire subtree under the current OPP without exploring it.

To avoid enumerating an exponential space of symmetries, Saucy exploits two *group-theoretic* pruning techniques: *coset* pruning and *orbit* pruning. To explain these techniques, we need to introduce a few more group-theoretic concepts.

For any subgroup $X \le Aut(G)$ of the automorphism group of our graph. Let X_v denote the subgroup of X that "fixes" the vertex v, i.e., $X_v = \{\gamma \in X | v^\gamma = v\}$. This is referred to as the *stabilizer* subgroup of v in X. The (left) coset of X_v in X containing permutation η is the set $\{\eta\gamma | \gamma \in X_v\}$. This definition implies that *any* coset element can generate the entire coset by composing that element with the elements of X_v. The set of cosets of X_v partitions X into equal-sized subsets. Now assume that Z is a set of generators for X_v. A set of generators for the parent group X can be obtained by augmenting Z with a *single* representative from each coset of X_v.

We say that vertices u and v share the same *orbit* (denoted

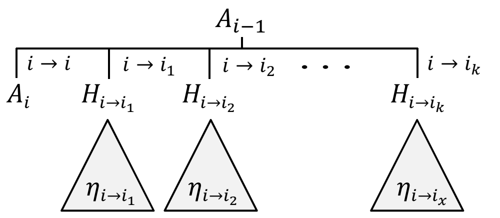


Figure 1. Structure of the permutation search tree.

$u \sim v$) if there exists symmetry $\gamma \in A = Aut(G)$ that maps u to v, i.e., $u^\gamma = v$. The partition of V induced by the *equivalence* relation $\sim$ is called the *orbit partition*. The subsets of V in the orbit partition are referred to as the *orbits*.

To enable group-theoretic pruning in Saucy, the left-most tree path should correspond to a sequence of *stabilizer subgroups* of A ending in the identity. Without loss of generality, assume that $1, 2, \ldots, n$ in that order is the chain of stabilizers. We abuse notation slightly and let $A_i = A_{1,2,\ldots,i}$ be the subgroup of A that fixes all the vertices $1, 2, \ldots, i$. The goal at the tree node at level $i-1$ is to compute the group A_{i-1}. Assume that the subtree under A_{i-1} is expanded by mapping vertex i to vertices $i, i_1, i_2, \ldots, i_k$ in that order (see Figure 1). Given A_i (the subgroup of A_{i-1} that fixes i), we can compute (generators for) A_{i-1} as follows. For every $j = 1, \ldots k$, let $H_{i\mapsto i_j}$ be the subset of permutations in A_{i-1} that map i to i_j. Every $H_{i\mapsto i_j}$ is either empty or a coset of A_i. We solve up to k independent problems where problem i_j seeks *one* automorphism $\eta_{i\mapsto i_j}$ that maps i to i_j. When the search is looking for some such $\eta_{i\mapsto i_j}$, we refer to $H_{i\mapsto i_j}$ as the *current subtree* or *current coset*. If we find such an $\eta_{i\mapsto i_j}$, we add it to the set of generators for A_{i-1}. 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 $\ell > j$, then problem i_ℓ can be skipped since its corresponding coset must contain redundant generators.

In the search for coset representatives, Saucy exploits two *OPP-based* pruning techniques: *non-isomorphic OPP* pruning and *matching OPP* pruning. Non-isomorphic OPP pruning indicates that there are no permutations in the subtree rooted at that node which are symmetries of the graph. Matching OPP pruning identify a candidate permutation at a tree node without the need to explore the subtree rooted at that node. Both of these pruning mechanisms are algorithmic enhancements that help eliminate unnecessary search.

III. Conflict Analysis and BackJumping

A combinatorial search algorithm reaches a *conflict* when it recognizes that the current state can never lead to a solution, and hence backtracks. In this case, a common technique is to analyze the conflict and use its outcome to guide and prune the search space. For example in SAT, it is likely for basic DPLL [9, 8] to encounter the same chain of *conflicting* assignments multiple times during the search. To avoid such redundancy, modern SAT solvers analyze each conflict to identify a small set of assignments that are sufficient to expose that conflict. These assignments form a new clause, which is saved (*learned*) by the solver and used by propagation to avoid the same conflict in the future [23].

In this section, we explain the type of conflicts that arise in Saucy, and present a procedure that identifies a set of decisions responsible for each conflict. Using this analysis, we describe a backjumping routine that has very low overhead and improves runtime (sometimes drastically) on a wide variety of benchmarks. Backjumping is the first use of our conflict analysis framework, which opens the way to the use of other conflict-driven techniques.

A. *Conflict Analysis in* Saucy

Current symmetry-detection algorithms use partition refinement as an effective way to propagate decisions through the graph constraints. This is analogous to constraint propagation in general purpose CSP (or SAT) solvers. However, partition refinement is specific to GA, and differs in many aspects from general propagation in CSP and SAT solvers. Moreover, Saucy's conflicts differ in nature from conflicts in CSP solvers. Therefore, performing conflict analysis in the GA setting requires appropriate adaptation.

CSP (and SAT) solvers reach a conflict when propagation assigns one variable to two non-overlapping sets of values. A conflict in Saucy arises when either an OPP is non-isomorphic or an isomorphic OPP violates the edge constraints (the latter is captured by simultaneous refinement and leads to conflicts similar to the ones in CSP solvers). The isomorphic OPP in (1) violates the edge constraint if there exists cell indices i and j such that for two vertices $v \in T_i$ and $u \in B_i$, $|E(v, T_j)| \neq |E(u, B_j)|$. In this situation, we say that cells i and j are responsible for the conflict. Note that since the partitions are equitable, the choice of u and v does not matter. In the special case where cells i and j are singletons, this corresponds exactly to violating an edge constraint in the CSP formulation of symmetry detection. The case where the OPP is non-isomorphic has no obvious analogue for CSP solvers, and requires us to further elaborate the partition refinement process in order to define the cells that caused such a conflict.

Up to the point where Saucy notices a conflict, the OPP is isomorphic, and every step of mapping and refinement proceeds identically for the top and bottom partitions. Therefore, we need to analyze the refinement procedure only for one partition, and here we pick the top partition.

Each step of mapping and refinement consists of splitting one parent cell P into two children cells: C_1 and C_2. Every split that is not a decision is induced by another cell D with the property that for every $v \in C_1, w \in C_2$, $|E(v, D)| \neq |E(w, D)|$. We call cell D that induces the split the *cause* cell. Note for future reference that cells C_1 and C_2 will be

$$\begin{bmatrix} 1234\ldots|56 \\ 1234\ldots|78 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1|234\ldots|56 \\ 2|134\ldots|78 \end{bmatrix} \rightarrow \begin{bmatrix} 1|23\ldots|4\ldots|56 \\ 2|14\ldots|3\ldots|78 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1|23\ldots|4\ldots|5|6 \\ 2|14\ldots|3\ldots|7|8 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1|3|2\ldots|4\ldots|5|6 \\ 2|4|1\ldots|3\ldots|7|8 \end{bmatrix} \rightarrow \begin{bmatrix} 1|3|2\ldots|4\ldots|\ldots|5|6 \\ 2|4|1\ldots|3\ldots \quad |7|8 \end{bmatrix}$$

Figure 2. The OPPs in a branch leading to a conflict. We use "..." to signify there are omitted vertices in a cell. The first decision is to map $1 \rightarrow 2$. Since 1 is connected to $\{4\ldots\}$ but not $\{23\ldots\}$, partition refinement splits the top cell $\{234\ldots\}$. The connections of 2 cause an equivalent split in the bottom partition. Next, we map $5 \rightarrow 7$, and suppose that this mapping does not cause further refinement. Finally, the algorithm maps $3 \rightarrow 4$. Vertex 3 is adjacent to some but not all vertices in the top cell $4\ldots$, causing a split in that cell. On the other hand, suppose vertex 4 is adjacent to none of the vertices in the bottom cell $3\ldots$. Hence, the OPP becomes non-isomorphic, and we encounter a conflict.

formed every time P and D are in the partition. We say that a cell that is split by a decision is caused by a special cell $*$, and is the cause of its sibling.

To summarize, when a cell P is split into two cells, C_1, C_2, we set:

```
parent[$C_1$] := parent[$C_2$] := $P$
if $C_1$ is a decision
    cause[$C_1$] := $*$
    cause[$C_2$] := $C_1$
else if split induced by $D$
    cause[$C_1$] := cause[$C_2$] := $D$
```

Before a conflict arises, the OPP is isomorphic. Hence, the cells responsible for a conflict caused by a non-isomorphic OPP are the cause and parent cells of the last split.

We can now create a DAG whose nodes are cells. There are two kinds of edges: parent and cause. Each node has one incoming *parent* edge (from the parent cell), and each node that does not correspond to a decision has one incoming *cause* edge (from the cause cell). We call this DAG the *implication* graph (see Figures 2 and 3 for an example). When a conflict occurs, we are interested in finding a set of cells (nodes) which is sufficient to cause the conflict in the current subtree (for the current coset). We obtain such a set by looking at a vertex cut in the implication graph. To accomplish this, we create a new source node $*$ with edges towards every cell corresponding to a decision. Any vertex cut between $*$ and the cells that caused the conflict is a NoGood set of cells for this subtree. That is, if we ever encounter this set of cells in this subtree, we know we will encounter the same conflict. Hence, if we ever see the same NoGood set of cells, we can prune the tree.

This general procedure allows us to potentially create many NoGoods from just one conflict. A NoGood will be useful if it appears again in the search tree. We also want to be able to keep track of NoGoods efficiently. For both these reasons, we focus on NoGoods that consist only of decisions. A simple modification to the code would allow us to find NoGoods consisting of singleton cells.

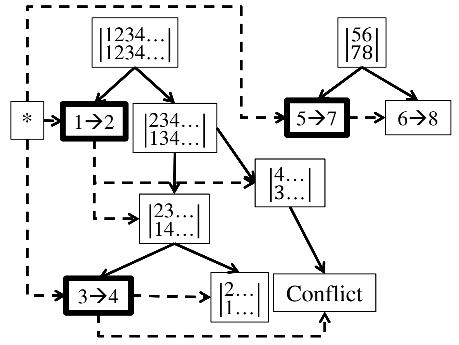


Figure 3. The implication graph corresponding to the branch of the subtree in Figure 2. Cells are shown in boxes, and bolded boxes denote decisions. Dashed edges are cause edges, and solid edges are parent edges. By finding a cut between the node * and the conflict, we can see that the decision $5 \rightarrow 7$ was not responsible for the conflict.

In implementation, we do not store the implication graph explicitly. Instead, for the ith split in the current branch, we store information about the children cells ($\mathrm{C}_1[i]$ and $\mathrm{C}_2[i]$) created by this split, and the parent and cause for these cells. If the split was a decision, then $\mathrm{C}_1[i]$ will be the cell containing the decision. Recall that the parent is the same for both children. To find the NoGood, we maintain a set R of *responsible cells*. Initially, we place the two cells responsible for the conflict in R. Then we iterate backward through the splits. Whenever we encounter one of the cells in R, we pop it out, knowing that this split was involved in causing the conflict. If the split was a decision, we add it to the NoGood and add the parent cell to R.[1] If the split was not a decision, we insert the cause and parent cells into R. Below is the pseudocode for the algorithm described above.

```
Conflict Analysis
$i$ := index of last split before conflict
$R$ := {cells responsible for the conflict }
while $i$ >= index of first split in subtree
    if $\mathrm{C}_1[i] \in R$ OR $\mathrm{C}_2[i] \in R$ then
        $R$.remove($\mathrm{C}_1[i], \mathrm{C}_2[i]$)
        $R$.add(parent[$\mathrm{C}_1[i]$])
        if cause[$\mathrm{C}_1[i]$] == $*$ then
            Add $\mathrm{C}_1[i]$ to the NoGood
        else $R$.add(cause[$\mathrm{C}_1[i]$])
    $i := i - 1$
```

[1]We must add the parent cell, otherwise the NoGood would not be complete in cases where the set of possible mappings for this decision was reduced in the current coset subtree.

B. BackJumping

In this subsection, we describe an implementation of backjumping based on our conflict analysis introduced in Section III-A. This is the first use of conflict analysis for the search for graph automorphism.

Once we completely explore the subtree rooted at a node in the search tree, we conclude that there are no automorphisms with the prefix of that node. Hence, the prefix of that node is a NoGood. In fact, we can hope to do better by taking the union of all NoGoods that appeared in the subtree. This will potentially give us a smaller NoGood, and allow us to backtrack further up the tree if the levels immediately above the current node were not involved in any of the conflicts.

In implementation, we do not need to store every (in fact any) NoGood that we encounter in order to implement backjumping. For every node N in the current branch of the search tree, we maintain the deepest ancestor that is involved in any NoGood arising from a conflict in the subtree rooted at N. Therefore, at any moment, it suffices to keep track of the *deepest conflicting ancestor* (DCA) for every level in the current search tree. When we encounter a conflict, we perform conflict analysis and update the DCA of every node above the conflict. In pseudocode, for a split i, let level$[i]$ be the level of the tree at which the split occurs. We change the conflict analysis code by adding line 01b after line 01, and replacing line 07 by lines 07a-c:

```
Conflict Analysis for Backjumping
01b  last-level := level[conflict split] $- 1$
...
07a        for ($\ell$ := last-level ; $\ell \geq$level[$i$]; $\ell := \ell - 1$)
07b              DCA[$\ell$] := max(DCA[$\ell$], level[$i$])
07c        last-level := level[$i$] $- 1$
...
```

IV. Branching Heuristics

Branching heuristics highly affect the performance of combinatorial search algorithms, including symmetry detection. In Saucy, branching is performed by choosing a target cell and a target vertex from the top partition. On the leftmost tree path, Saucy chooses the first non-singleton cell as the target cell, and the first vertex in that cell as the target vertex. In the other parts of the tree, Saucy looks for swaps of vertices, i.e., whenever it maps vertex v_1 to vertex v_2, it tries to map v_2 to v_1 right after. This is not always possible as partition refinement might preclude the mapping of v_2 to v_1. In that case, Saucy picks the first vertex of any non-singleton cell of the top partition which is not identical to its corresponding cell of the bottom partition. The vertex-swap heuristic can also be viewed as a mechanism to maximize the occurrence of matching OPPs. In practice, this heuristic is most effective when the generators of the automorphism group are sparse.

In addition to first (f), we consider four other heuristics for selecting a target cell: smallest (s), largest (l), smallest maximally connected (smc), and largest maximally connected (lmc). The smallest heuristic selects the first cell with the fewest vertices. Similarly, the largest heuristic selects the first cell with the most vertices. The other two heuristics are based on the connectivity of the cells. The goal is to try to (greedily) maximize the effect of partition refinement. In an equitable partition, we define the degree of a cell C as the number of other cells that vertices in C are connected to. A maximally connected cell is a cell of maximum degree. The smallest (resp. largest) maximally connected heuristics selects the first maximally connected cell with the smallest (resp. largest) number of vertices.

We consider a total of 10 heuristics: we apply one of the 5 heuristics for the leftmost-path (f, s, l, smc and lmc); for the rest of the tree, we either keep the same heuristic (we call this 'static'), or use Saucy's vertex-swap heuristic (we call this 'dynamic'). The vertex-swap heuristic is dynamic since it depends on the search tree so far.

V. Experimental Evaluation

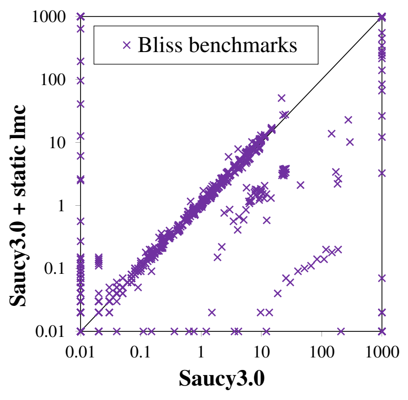
We ran Saucy 3.0 augmented with various combinations of backjumping, branching heuristics, and restarts on a suite of 2923 graph benchmarks. Our experiments were conducted on a SUN workstation equipped with a 3GHz Intel Dual-Core CPU, a 6MB cache and an 8GB RAM, running the 64-bit version of Redhat Linux. We applied a time-out of 1000 seconds to all our experiments.

The graphs in our suite are divided into three categories:

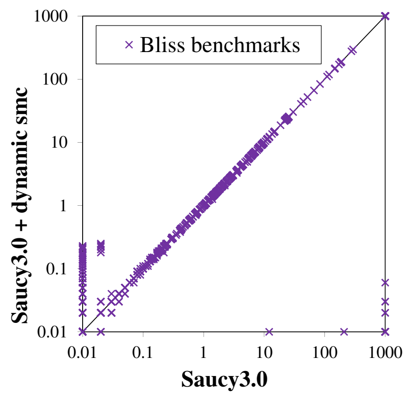
- Bliss benchmarks: 1627 relatively small and dense graphs including random graphs, highly-structured graphs, and variations of Miyazaki graphs [20].
- Saucy benchmarks: 96 large and sparse graphs from logic circuits and their physical layouts [24, 14], internet routers [5, 12], and road networks in the US states and its territories [21].
- SAT 2011 benchmarks: 1200 SAT 2011 competition CNFs [22] encoded as graphs [1].

A. Branching Heuristics

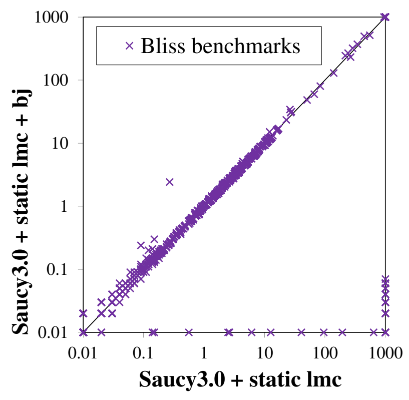
We ran 10 different Saucy branching heuristics (see Section IV) on the Bliss benchmarks. None of the branching heuristics outperformed all others on all benchmarks. For example, some Hadamard graphs (had) and projective plane graphs (pp) were only solved by lmc-static. On the other hand, many instances of cmz and mz-aug2 timed out with lmc-static, but were solved in less than a second with the default heuristic. One consistent pattern was that the largest, smallest and first heuristics (both static and dynamic) were dominated by either the lmc or the smc heuristics. Hence, we exclude their evaluation from this paper. We show a comparison of the lmc-static and smc-dynamic heuristics



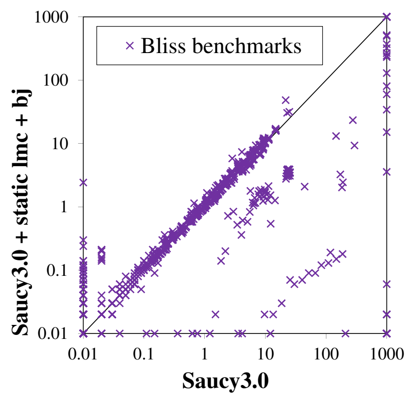
(a) static-lmc vs. Saucy3.0



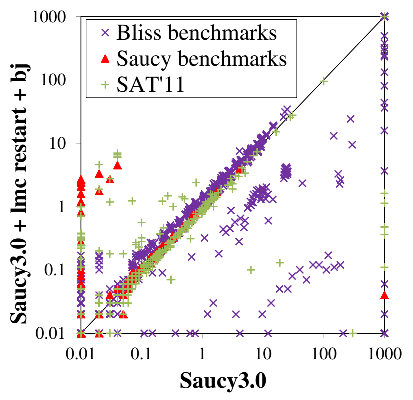
(b) dynamic-smc vs. Saucy3.0



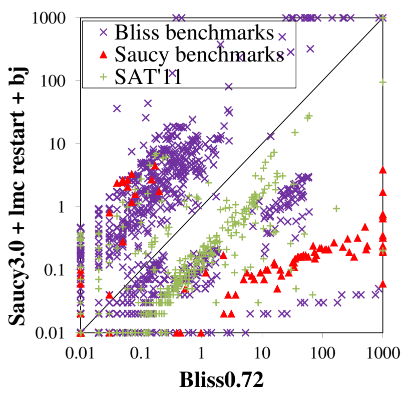
(c) static-lmc + backjumping vs. static-lmc



(d) static-lmc + backjumping vs. Saucy3.0



(e) lmc-restart + backjumping vs. Saucy3.0



(f) lmc-restart + backjumping vs. Bliss0.72

Figure 4. The runtimes of Saucy3.0 augmented with different enhancements vs. Saucy3.0 and Bliss0.72.

with the default `Saucy`3.0 heuristic (first-dynamic) for the `Bliss` benchmarks in Figures 4(a) and 4(b).

B. Backjumping

We ran our implementation of backjumping (described in Section III-B) in combination with all 10 heuristics on the `Bliss` benchmarks. In all cases, we saw negligible overhead on runtime and, in some cases, significant gains. Figure 4(c) shows the runtime for lmc-static with backjumping plotted against the same heuristic without backjumping. Adding backjumping to any other heuristic had very similar effects. The lmc-static heuristic with backjumping outperformed all other heuristics for these benchmarks. In Figure 4(d), we compare the runtime of lmc-static with backjumping to `Saucy`3.0. Backjumping with lmc reduces runtimes significantly for the `ag`, `had`, and `pp` benchmarks, and it is not noticeably different on the remaining benchmarks. The only graphs in the `Bliss` benchmarks for which the lmc-static+bj configuration takes more than 20 seconds are some of the `had` and `pp` graphs. Some of these instances are solved more efficiently by `Bliss`0.72. On the other hand, `Bliss`0.72 times out on several of the `cmz` graphs (variations of the Miyazaki graphs), which are all solved in less than one second by `Saucy` with the default heuristic (first-dynamic) or with any heuristic with backjumping.

`Saucy`3.0's dynamic heuristic is often very effective, especially for large and sparse graphs, and has the additional advantage of finding sparse generators. However, it occasionally does not perform as well as the static-lmc heuristic due to getting stuck in very deep paths in some subtrees. We noticed that in most instances where the dynamic heuristic was effective, it encountered few conflicts (less than 100) in any subtree. To harness the benefit of both the dynamic heuristic and good ordering given by lmc, we implemented a simple restart scheme. We run `Saucy` with the lmc branching heuristic for the leftmost path, and the dynamic heuristic in the subtrees. If the dynamic heuristic encounters 100 conflicts in a given subtree, we restart that subtree with the static heuristic (i.e., lmc from the leftmost). This gave some small improvements over lmc-static with backjumping for the dense `Bliss` benchmarks, as well as improving over `Saucy`3.0 on the large and sparse benchmarks (see Figure 4(e)). In fact, a few of the benchmarks were so large that computing the largest maximally connected cell at every level was prohibitive. To avoid this, we simply revert to using the first cell heuristic whenever the number of vertices is greater than 50,000 or the number of edges is greater than 200,000. We note that even for the very large benchmarks (up to millions of vertices) the backjumping procedure did not slow down the performance.

Figure 4(f) compares the results of `Saucy`3.0 with restarts and backjumping with `Bliss`0.72. On the large and sparse `Saucy` and SAT 2011 benchmarks, `Saucy`3.0 already outperforms `Bliss`0.72 (see [18] for a more detailed comparison). For the small and dense `Bliss` benchmarks, the results were mixed. In general, the `pp` and `had` graphs were more challenging for `Saucy`. This was while `Bliss` had difficulties finishing the `cmz` graphs.

VI. Further applications of conflict analysis

Our framework for conflict analysis in GA opens the way for the adaptation of more advanced techniques from the SAT solving literature. In this section, we discuss some preliminary work about the possible use of techniques analogous to conflict-driven learning in SAT solvers, and identify some roadblocks.

Conflict-driven learning in `Saucy`: We implemented ideas from conflict-driven learning directly within `Saucy`. NoGoods in our setting correspond to learned clauses. We maintain one NoGood for each conflict. For each node in the search tree, we attempt to propagate the NoGoods together with the OPP as follows. We find all NoGoods that are unit (all but one of the assignments in the NoGood has been made). Each unit NoGood precludes one assignment for one vertex. If after this process any vertex has a unique allowed assignment, then we make that assignment, and repeat partition refinement. In fact, it never happened in our experiments that any vertex was assigned because of the stored NoGoods.

There are two main limitations to adapting conflict-driven learning to GA; the branching factor is large and the type of clauses we can learn is limited. Instances that remain hard for graph symmetry detection are those where partition refinement does not yield a fine partition of the vertices. In other words, in hard instances, even after partition refinement, the branching factor remains large, making resolution of the NoGood clauses less likely. By contrast, in SAT instances, if we find that an assignment for a variable is bad (a unit clause), we can immediately assign the opposite value, which results in further propagation. Resolution is made even less likely by the fact that NoGoods correspond to clauses where all literals are negated.

Admittedly, our use of NoGoods is limited. More advanced resolution within the NoGoods and between NoGoods and the OPP may help overcome the obstacles outlined above. However, these would involve more overhead, which may not be affordable given the size of the benchmarks. It is important to keep in mind that for SAT-style resolution, the number of variables is quadratic in the number of vertices (one variable for each potential mapping).

SAT solvers on the graph automorphism problem: We wanted to further analyze whether or not the problem instances arising from GA make them amenable to SAT-style learning. We converted some of the benchmarks (small ones because the encoding is cubic) to SAT instances. As expected, `Minisat` [10] (the SAT solver we chose for our experiments) was slow compared to `Saucy`, since it is not specifically optimized for the symmetry-detection problem. Our goal was to check whether the clauses learned by the solver on these instances could help us find better invariants in symmetry detection. Since SAT solvers look for just one solution, we created SAT instances corresponding to coset-level subtrees of the `Saucy` search. We observed that most of the learned clauses (especially early in the search) were not interpretable in a graph-theoretic sense. More discouraging was the fact that the runtime for these instances did not change significantly by disabling or enabling learning. In fact, in some cases, learning slowed down the solver.

In the remainder, we describe our encoding of GA to SAT. We start with a linear program for GA, and then encode the linear constraints as CNF clauses. An $n \times n$ matrix P of 0s and 1s is a *permutation matrix* if P has exactly one 1 in each row and in each column. A permutation matrix P acts on a vector by permuting its coordinates, and corresponds to the permutation $\sigma(P) : [n] \to [n]$, where $\sigma[i] = j$ is the row index corresponding to the 1 in the the ith column (i.e., $P_{ji} = 1$). Permutation matrices are symmetric, so $P^{-1} = P^{\top}$. Let A be the adjacency matrix of graph G. Then $P^{\top}AP$ is the adjacency matrix of the graph $G^{\sigma(P)}$ obtained by permuting the vertices of G by $\sigma(P)$. Therefore, finding the automorphism group of graph G can be encoded as finding the set of permutation matrices P satisfying $P^{\top}AP = A$, or equivalently, $AP = PA$.

The variables of the SAT instance are the n^2 entries p_{ij} of the matrix P. To enforce that P is a permutation matrix, we add $\binom{n}{2} + 1$ clauses for each row and for each column. One clause ensures that at least one variable in the row (or column) is set to 1: $(p_{i1} \vee p_{i2} \vee \cdots \vee p_{in})$. The remaining $\binom{n}{2}$ clauses ensure no two variables in the row (or column) are set to 1: $(\overline{p_{ij}} \vee \overline{p_{ik}})$, for every $j < k \leq n$. A straight forward translation of the equation $AP = PA$ to CNF would give exponentially many clauses. A simple insight will help us avoid this exponential overhead. Entry i, j in the matrix equation above is: $\sum_{k=1}^{n} a_{ik}p_{kj} = \sum_{k=1}^{n} p_{ik}a_{kj}$. The variables p_{kj} that appear on the LHS all come from the same column (j is fixed), similarly for RHS the variables come from the same row. Since P is a permutation matrix, the sum on both sides of the equation above is at most 1. Therefore, for each $i, j \leq n$, we can encode the equality constraint by $2n$ clauses:

$$\bigwedge_{\ell=1}^{n}\left(\bigvee_{k=1}^{n} a_{ik}p_{kj} \vee \overline{p_{i\ell}a_{\ell j}}\right) \bigwedge_{\ell=1}^{n}\left(\bigvee_{k=1}^{n} p_{ik}a_{kj} \vee \overline{a_{i\ell}p_{\ell j}}\right).$$

Given a specific graph G, the coefficients a_{ij} will be fixed, and the clauses where the $a_{i\ell}$ or $a_{\ell j}$ coefficients are zero will be satisfied trivially. Therefore the total number of clauses in our encoding is $4en$ ($2n$ for every half-edge). Moreover, the width of these clauses is one more than the degree of a vertex. Overall, the SAT encoding has n^2 variables and $2n\left(\binom{n}{2}+1\right)+4en=O(n^3)$ clauses.

SAT solvers look for just one satisfying assignment, which could be the trivial identity permutation. If we are specifying a subtree of the `Saucy` search, we simply set the variables p_{ij} corresponding to the mappings to true, which will preclude the identity mapping as long as one of the mappings is non-identical. It is not really possible to encode the full automorphism problem to SAT, since there may be too many automorphisms to list. Instead we ask the SAT solver to look for any non-trivial automorphism by adding the clause $(\overline{p_{11}} \vee \overline{p_{22}} \vee \cdots \vee \overline{p_{nn}})$.

VII. Conclusions

We explored the effects of different heuristics and conflict-driven backjumping in symmetry detection. Experimental evaluation confirmed that our backjumping implementation improves performance significantly for many benchmarks, while maintaining a low overhead even for very large graphs. Our conflict analysis framework opens the way for adapting further conflict-driven techniques from SAT solvers. We identified several obstacles to further improvements, but expect that they can be addressed by better propagation of NoGoods, and other techniques like adaptive branching based on the frequency of variables appearing in NoGoods.

Acknowledgment

This work was funded in part by NSF award number 0810865, and was supported in part by the Institute for Mathematics and its Applications with funds provided by the National Science Foundation.

References

[1] F. A. Aloul, A. Ramani, I. Markov, and K. 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.

[2] L. Babai and E. M. Luks. Canonical labeling of graphs. In *Proc. 15th STOC*, pages 171–183. ACM Press, 1983.

[3] L. Babai and S. Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *J. Computer and Sys. Sci.*, 36:254–276, 1988.

[4] A. T. Balaban. Reflections about mathematical chemistry. *Foundations of Chemistry*, 7:289–306, Oct 2005.

[5] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the Internet. In *USENIX Annual Technical Conference*, pages 1–13, 2000.

[6] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. 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.

[7] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. 45th IEEE/ACM Design Automation Conference (DAC)*, pages 149–154, Anaheim, California, 2008.

[8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[10] N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[11] J. B. Fraleigh. *A First Course in Abstract Algebra*. Addison Wesley Longman, Reading, Massachusetts, 6th edition, 2000.

[12] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *IEEE INFOCOM*, pages 1371–1380, 2000.

[13] I. Hargittai and M. Hargittai. *Symmetry through the eyes of a chemist*. Springer, 2009.

[14] ISPD 2005, *http://archive.sigda.org/ispd2005/contest.htm* .

[15] T. Junttila and P. Kaski. Conflict propagation and component recursion for canonical labeling. In *Proc. First ICST*, TAPAS, pages 151–162, 2011.

[16] H. Katebi, K. A. Sakallah, and I. L. Markov. Symmetry and satisfiability: An update. In *Proc. Satisfiability Symposium (SAT)*, Edinburgh, Scotland, 2010.

[17] H. Katebi, K. A. Sakallah, and I. L. Markov. Conflict anticipation in the search for graph automorphisms. In *Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Merida, Venezuela, 2012.

[18] H. Katebi, K. A. Sakallah, and I. L. Markov. Graph symmetry detection and canonical labeling: Differences and synergies. In *Turing-100, EPIC*, volume 10, pages 181–195, 2012.

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

[20] T. Miyazaki. *The complexity of McKays canonical labeling algorithm*, page 239. Amer Mathematical Society, 1997.

[21] U. S. Census Bureau. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html.

[22] SAT Competition, http://www.satcompetition.org.

[23] J. P. M. Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *ICCAD '96*, pages 220–227, 1996.

[24] M. N. Velev and R. 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, 2001.