Symmetry in Finite Combinatorial Objects: Scalable Methods and Applications

by

Hadi Katebi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2013

Doctoral Committee:

Professor Karem A. Sakallah, Chair
Associate Professor Kevin J. Compton
Professor Igor L. Markov
Professor Martha E. Pollack

How many symmetries does the 3×3 Rubik's cube have?

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

AI Artificial Intelligence

AIG And-Inverter Graph

BDD Binary Decision Diagram

CAD Computer Aided Design

CNF Conjunctive Normal Form

LCA Lowest Common Ancestor

OPP Ordered Partition Pair

SAT Boolean Satisfiability

SBP Symmetry-Breaking Predicate

VLSI Very-Large-Scale Integration

ABSTRACT

Symmetry in Finite Combinatorial Objects:
Scalable Methods and Applications

by

Hadi Katebi

Chair: Karem A. Sakallah

Symmetries of combinatorial objects are known to complicate search algorithms, but such obstacles can often be removed by detecting symmetries early and discarding symmetric subproblems. Canonical labeling of combinatorial objects facilitates easy equivalence checking through quick matching. All existing canonical-labeling software also finds symmetries, but the fastest symmetry-finding software does not perform canonical labeling. In this thesis, we describe highly scalable symmetry-detection algorithms for two widely-used combinatorial objects: graphs and Boolean functions. Our algorithms are based on a decision tree that combines elements of group-theoretic computation with branching and backtracking search. Moreover, we contrast the search for graph symmetries and a canonical labeling to dissect typical algorithms and identify their similarities and differences. We develop a novel approach to graph canonical labeling where symmetries are found first and then used to speed up the canonical-labeling routines. Empirical results are given for graphs with millions of vertices and Boolean functions with hundreds of I/Os, where our algorithms can often find all symmetry group generators or a canonical labeling in seconds.

CHAPTER I

Introduction

Finite combinatorial objects in computer science are viewed as means for creating, modeling, and advancing computational problems. Examples of such objects are *graphs*, *hypergraphs*, and *Boolean functions*, which arise in various branches of computer science, including Very-Large-Scale Integration (VLSI), Computer Aided Design (CAD), networks and Artificial Intelligence (AI). In representative applications, graphs can model the backbone of the Internet [60], hypergraphs can help solve VLSI cell partitioning [15], and Boolean functions can represent inputs to CAD tools [9].

Any combinatorial object is associated with an underlying *variable set*, to which one can apply arbitrary permutations or value substitutions. For example, a directed graph is defined by a set of vertices V and a set of edges $E \subseteq V \times V$, an undirected hypergraph is similarly defined by its hyperedges $E \subseteq 2^V$, and a Boolean function is defined by its input set X and minterms $M \subseteq 2^X$.

Enumerative combinatorics is a branch of combinatorics concerned with counting objects or their certain properties. For example, counting the number of different orderings of a deck of cards, or enumerating the number of permutations of a graph are both enumerative combinatorial problems.

Solving a combinatorial problem often involves the use of algebraic methods, in

particular, *group theory*. Group theory is a branch of abstract algebra that studies the algebraic structures known as *groups*. A *group* comprises 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.

Given a combinatorial object, one enumerative combinatorial problem that requires group-theoretic treatment is *finding the symmetries* of the object. A *symmetry* of an object is defined as a permutation of its variables that leaves the object unchanged. For example, a symmetry of a graph is a permutation of the graph's vertices that preserves the graph's edge relation, and a symmetry of a Boolean function is a permutation of the function's inputs and outputs, with their possible negation, that preserves the value of the function for all input combinations.

The set of all symmetries of a combinatorial object forms a group under functional composition. This group is referred to as the *symmetry group* of the object. In general, the order (size) of the symmetry group of an object is *exponential* in the number of its variables. Nevertheless, all symmetries of an object can be generated from just a subset of its symmetries. This is accomplished by repeatedly composing the elements of that subset under functional composition. Such a subset is called a *symmetry group generating set* and each of its elements is called a (*group*) *generator*.

In the remainder of this thesis, we study symmetries of two widely-used combinatorial objects: graphs and Boolean functions. Our interest in these two objects is justified by the fact that they can compactly and conveniently model many computational problems. For instance, a graph can be used to encode a Conjunctive Normal Form (CNF) formula, which is then passed to a graph symmetry-detection program to find a set of generators for the formula's symmetry group. These symmetries are subsequently used to augment the original formula with *symmetry-breaking predicates* that preclude a Boolean Satisfiability (SAT) solver from redundant search in symmetric portions of the solution space.

To find the symmetries of a graph, we develop scalable algorithms, named **saucy**, through nested *partition refinement*. The goal of partition refinement is to prune away unpromising branches of the permutation space. We then incorporate group-theoretic techniques to avoid explicit enumeration of the possibly exponential number of symmetries. As the outcome of the search, **saucy** returns a set of generators for the symmetry group of the graph, finds the group's *orbit partition* (defined later), and reports the order of the group.

Closely related to graph symmetry detection is the problem of *canonical labeling* which assigns a unique signature to a graph that is invariant under all possible labelings of its vertices. Symmetry detection and canonical labeling are both related to the structural or functional properties of the combinatorial objects in question. An important remark is that the symmetries of a graph map each labeling to the same labeling. Therefore, if all symmetries are known, it may be sufficient to visit only one labeling from each equivalence class. As a result, all existing graph canonicalization tools, such as **nauty** [43], **bliss** [34, 35], **traces** [50] and **nishe** [55], also find symmetries along the way during the search.

Unlike the canonical-labeling packages, which also produce symmetries as a byproduct, **saucy**'s algorithms and data structures are optimized to just finding a set of symmetry group generators. This is accomplished through searching the space of *permutations*, as opposed to the space of *labelings*, which subsequently enabled **saucy** to incorporate three major enhancements, namely, *simultaneous partition refinement*, *non-isomorphic OPP pruning*, and *matching-OPP pruning* (all explained later). These enhancements delinked the search for symmetries from the search for a canonical labeling, and yielded a remarkable 1000-fold improvement in runtime for many large sparse graphs with sparse symmetry generators, i.e., generators that "move" only a tiny fraction of the graph's vertices.

In order for the existing graph canonical-labeling packages to benefit from the scal-

ability of the grah symmetry-detection algorithms, we propose a two-pass canonical-labeling approach that first finds symmetries, and then uses them to expedite the search for a canonical labeling. In other words, our approach uses the efficiency of **saucy** symmetry-finding algorithms as a pre-processing step for canonical-labeling frameworks. Extensive empirical results convincingly demonstrate the benefits of our canonical-labeling approach.

To discover symmetries of other combinatorial objects, such as a Boolean function, we adjust our proposed framework for finding symmetries of graphs to consider the specifics of the new object in question. For example, we have to be aware of the fact that the symmetries of a Boolean function are different from those of a graph, since they are related to the *functional*, and not *structural*, properties of the function. Note that althogh it is possible to encode a Boolean function as a graph and then invoke a graph symmetry-detection tool to find its symmetries, we refrain from this approach, since it is not complete, i.e., the structural symmetries of a Boolean function might be just a subset of its functional symmetries.

Symmetries of Boolean functions have numerous applications in logic synthesis and verification. One common application is in *Boolean matching*, where functional equivalence of two Boolean functions under permutation (and negation) of their inputs and outputs is investigated [37, 2]. Other applications include BDD minimization [53] and circuit power optimization [17].

Most existing symmetry-detection algorithms for Boolean functions only look for *classical symmetries*, i.e., symmetries that include just *a single swap* of variables [61, 53]. As the number of such symmetries is at most quadratic in the number of a function's inputs, they can be evaluated one by one and explicitly enumerated. The caveat of these algorithms, however, is that they overlook symmetries that involve more than two variables. For example, a 4-to-1 multiplexer exhibits 16 symmetries under the permutation and negation of its inputs and output (see Figure 1.1), but

MUX: $z = a_0 s_1' s_0' + a_1 s_1' s_0 + a_2 s_1 s_0' + a_3 s_1 s_0$

$\gamma_1 : \iota$

$\gamma_2 : (a_0, a_2)(a_0', a_2')(a_1, a_3)(a_1', a_3')(s_1, s_1')$

$\gamma_3 : (a_1, a_2)(a_1', a_2')(s_0, s_1)(s_0', s_1')$

$\gamma_4 : (a_0, a_1, a_3, a_2)(a_0', a_1', a_3', a_2')(s_0, s_1, s_0', s_1')$

$\gamma_5 : (a_0, a_1)(a_0', a_1')(a_2, a_3)(a_2', a_3')(s_0, s_0')$

$\gamma_6 : (a_0, a_3)(a_0', a_3')(a_1, a_2)(a_1', a_2')(s_0, s_0')(s_1, s_1')$

$\gamma_7 : (a_0, a_2, a_3, a_1)(a_0', a_2', a_3', a_1')(s_0, s_1', s_0', s_1)$

$\gamma_8 : (a_0, a_3)(a_0', a_3')(s_0, s_1')(s_0', s_1)$

$\gamma_9 : (a_0, a_0')(a_1, a_1')(a_2, a_2')(a_3, a_3')(z, z')$

$\gamma_{10} : (a_0, a_2')(a_0', a_2)(a_1, a_3')(a_1', a_3)(s_1, s_1')(z, z')$

$\gamma_{11} : (a_0, a_0')(a_1, a_2')(a_1', a_2)(a_3, a_3')(s_0, s_1)(s_0', s_1')(z, z')$

$\gamma_{12} : (a_0, a_1', a_3, a_2')(a_0', a_1, a_3', a_2)(s_0, s_1, s_0', s_1')(z, z')$

$\gamma_{13} : (a_0, a_1')(a_0', a_1)(a_2, a_3')(a_2', a_3)(s_0, s_0')(z, z')$

$\gamma_{14} : (a_0, a_3')(a_0', a_3)(a_1, a_2')(a_1', a_2)(s_0, s_0')(s_1, s_1')(z, z')$

$\gamma_{15} : (a_0, a_2', a_3, a_1')(a_0', a_2, a_3', a_1)(s_0, s_1', s_0', s_1)(z, z')$

$\gamma_{16} : (a_0, a_3')(a_0', a_3)(a_1, a_1')(a_2, a_2')(s_0, s_1')(s_0', s_1)(z, z')$

Figure 1.1: Symmetries of a 4-to-1 MUX under permutation and negation of its I/Os. A rotational symmetry of the form $(a_1, a_2, a_3, ..., a_n)$ maps a_1 to a_2, a_2 to a_3, ..., and a_n to a_1. Also, ι denotes the identity.

none of those symmetries would be found this way.

Higher order symmetries, formed by *simultaneous swaps* of variables, have also been addressed in the literature. For example, the algorithm in [39] captures higher order symmetries (under permutation and negation of inputs) by performing *hierarchical partitioning* on the set of variables of a netlist. This algorithm, although capable of reporting symmetries beyond classical, does not always find all symmetries of a Boolean function.

Furthermore, most symmetry-detection algorithms only allow permutation of inputs, but not permutation of outputs [61, 39]. Such algorithms report symmetries of a multi-output function by isolating each output one at a time. Nevertheless, symmetries that are formed by simultaneous permutations of inputs and outputs are beneficial in EDA. For instance, [17] uses such symmetries to enhance post-placement

algorithms. It, however, performs an exhaustive search for symmetries, and hence, can only handle small (sub)circuits.

In this thesis, we propose new algorithms for detecting symmetries of Boolean functions under permutation (but not negation) of inputs and outputs. Our algorithms take a Boolean function in the form of an *And-Inverter Graph (AIG)*, construct a complete permutation tree, and systematically prune it by integrating group-theoretic (and other) techniques. To accomplish this, they build several graphs based on functional dependency and random simulation, and use them to refine the search space. They also take advantage of satisfiability to test functional equivalence under candidate permutations, and learn from satisfiability counterexamples to avoid recurring conflicts.

To assess the performance of our proposed symmetry-detection algorithms for graphs and Boolean functions, we assembled a collection of benchmarks containing graphs and combinational circuits from a wide range of applications. We also test the efficiency of our proposed graph canonical-labeling approach on the subset of the graph benchmarks that are very large and very sparse. Furthermore, we encode the Boolean matching problem as a symmetry-detection problem, and report the results of applying our symmetry-detection algorithms for Boolean functions to several Boolean matching instances.

Key contributions of our work are summarized in three categories. These categories and their contributions are:

– Symmetries of graphs:

    1. Proposing symmetry-detection algorithms for graphs that search the space of permutations (as opposed to the space of labelings).
    2. Allowing a more powerful partitioning through simultaneous refinement.
    3. Avoiding futile branches of the search through isomorphic-OPP pruning.

4. Finding symmetries earlier in the search through matching-OPP pruning.

– Canonical labeling of graphs:

1. Developing a two-pass canonical labeling algorithm that first finds graph symmetries, and then uses them to expedite the search for a canonical labeling.

– Symmetries of Boolean functions:

1. Proposing novel symmetry-detection algorithms for Boolean functions based on group-theoretic concepts.
2. Allowing permutations of both inputs and outputs.
3. Learning from satisfiability counterexamples to avoid recurring conflicts.
4. Formulating Boolean matching instances as symmetry-detection problems, and invoking our algorithms to solve them.

The remainder of this thesis is organized as follows. Chapter II provides necessary definitions and notation, and discusses preliminary work. Chapter IV describes **saucy** graph symmetry-discovery algorithms. Chapter V explains our proposed two-pass graph canonical-labeling approach. Chapter VI describes our symmetry-discovery algorithms for Boolean functions, and discusses our formulation of Boolean matching as a symmetry-detection problem. Chapter VII validates our symmetry-detection and canonical-labeling techniques in experiments, and reports the results of solving Boolean matching instances. Finally, Chapter VIII discusses conclusions, and provides future directions for our work.

CHAPTER II

Definitions and Notation

This chapter provides necessary definitions and notation.

2.1 Partitions

We assume familiarity with basic notions from set theory, including such concepts as *sets*, *subsets*, *set membership*, *set operations*, etc. More information on different set-theoretic concepts is available in many abstract set theory texts such as [27].

Definition II.1. A *set* $A = \{a_1, a_2, ..., a_n\}$ is a collection of *members* (or *elements*) $a_1, a_2, ..., a_n$. A set B is a *subset* of set A if and only if every element of B is also an element of A.

The cardinality of set A is dented by $|A|$. Set membership is denoted by $\in$, and set non-membership is denoted by $\notin$. The set that contains all objects is the *universal set*, and is denoted by U. The set that has no elements is the *empty set*, and is denoted by $\emptyset$. If set B is a subset of set A, we write $B \subset A$. If set B is a subset of set A or is equal to A, we write $B \subseteq A$.

There are several ways to operate on given sets and produce new sets. Among the most common set operations are:

Union: $A \cup B = \{a \mid a \in A \text{ or } a \in B\}$

Intersection:	$A \cap B = \{a \mid a \in A \text{ and } a \in B\}$
Difference:	$A - B = \{a \mid a \in A \text{ and } a \notin B\}$
Complement:	$A' = U - A = \{a \mid a \notin A\}$
Cartesian Product:	$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$

Two sets A and B are said to be *disjoint* if $A \cap B = \emptyset$. A collection of sets is *pair-wise disjoint* if every pair of the collection is disjoint.

Definition II.2. A *partition* $\pi = \{W_1, W_2, \cdots, W_m\}$ of set A is a list of non-empty pair-wise disjoint subsets of A whose union is A, i.e., $\bigcup_{i=1}^{m} W_i = A$, and for all i, j, $1 \leq i, j \leq m$ and $i \neq j$, $W_i \cap W_j = \emptyset$.

The subsets W_i are called the *cells* of the partition.

Definition II.3. An *ordered partition* $\pi = [W_1|W_2|\cdots|W_m]$ of set A is a partition of A where cells W_i are ordered.

Ordered partition π is said to be *unit* if $m = 1$ (i.e., $W_1 = A$) and *discrete* if $|W_i| = 1$ for $i = 1, \cdots, m$.

Example II.4. The following are example ordered partitions on set $A = \{1, 2, 3, 5, 7\}$:

Ordered partition:	$\pi = [1, 5 \mid 2, 3, 7]$
Unit partition:	$\pi = [1, 2, 3, 5, 7]$
Discrete partition:	$\pi = [2 \mid 7 \mid 3 \mid 1 \mid 5]$

Definition II.5. An *Ordered Partition Pair (OPP)* Π of set A is specified as

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

where π_T and π_B are ordered partitions of A.

Ordered partitions π_T and π_B are referred to, respectively, as the top and bottom 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. Isomorphic OPP Π is *matching* if its corresponding non-singleton cells are *identical*, i.e., contain the same elements. Isomorphic OPP Π is discrete (resp. unit) if its top and bottom partitions are discrete (resp. unit).

Example II.6. The following are example OPPs on set $A = \{1, 2, 3, 5, 7\}$:

Isomorphic OPP: $\left[\begin{array}{c|c} 1,3,5 & 2,7 \\ 1,5,7 & 2,3 \end{array}\right]$

Matching OPP: $\left[\begin{array}{c|c|c} 1,3,5 & 2 & 7 \\ 1,3,5 & 7 & 2 \end{array}\right]$

Discrete OPP: $\left[\begin{array}{c|c|c|c|c} 1 & 2 & 3 & 5 & 7 \\ 2 & 3 & 7 & 5 & 1 \end{array}\right]$

Unit OPP: $\left[\begin{array}{c} 1,2,3,5,7 \\ 1,2,3,5,7 \end{array}\right]$

Non-isomorphic OPP: $\left[\begin{array}{c} 1,3,5 \mid 2,7 \\ 1,7 \mid 2,3,5 \end{array}\right]$

2.2 Boolean Functions

A *Boolean domain* is a set consisting of exactly two elements: 0 (representing false) and 1 (representing true). A *Boolean variable* is a variable that takes values from a Boolean domain.

There are 16 ($= 2^{2^2}$) ways to define operations on two Boolean values a and b.

The most common operations are:

Conjunction (AND):	$a \wedge b$, which is 1 if and only if $a = 1$ and $b = 1$.
Disjunction (OR):	$a \vee b$, which is 1 if and only if $a = 1$ or $b = 1$.
Negation (NOT):	$\neg a$, which is 1 if and only if $a = 0$.
Exclusive disjunction (XOR):	$a \oplus b$, which is 1 if and only if $a = 1$ and $b = 0$ or $a = 0$ and $b = 1$.

Definition II.7. A *Boolean function* F with n inputs and m outputs is a function $F : B^n \mapsto B^m$ where $B = \{0, 1\}$ is the Boolean domain.

The set of all inputs of F is the *input set* of F. Likewise, the set of all outputs of F is the *output set* of F. We denote the input set of F by $X = \{x_1, ..., x_n\}$ and its output set by $Z = \{z_1, ..., z_m\}$.

Definition II.8. The *positive (negative) cofactor* of Boolean function F with regard to input $x \in X$, denoted by F_x ($F_{x'}$), is the function that fixes the value of x to one (zero).

Example II.9. The positive cofactor of the Boolean function $z_1 = x_1 \wedge x_2$ and $z_2 = x_1 \vee x_2$ with regard to x_1 is: $z_1 = x_2$ and $z_2 = 1$, and the negative cofactor of that function with regard to x_1 is: $z_1 = 0$ and $z_2 = x_2$,

Definition II.10. The *support* of output $z \in Z$, denoted by $supp(z)$, is the set of all inputs $x \in X$ that functionally affect z, i.e, $supp(z) = \{x \in X \mid F_x \neq F_{x'} \text{ for } z\}$.

Example II.11. For the Boolean function in Example II.9: $supp(z_1) = supp(z_2) = \{x_1, x_2\}$.

A Boolean function can be expressed as a *propositional formula* in *Conjunctive Normal Form* (*CNF*). A CNF formula is a conjunction of *clauses*, where a clause is a disjunction of *literals*. A literal is an input of the Boolean function or its negation.

Example II.12. The following are CNF formulas on input set $X = \{x_1, x_2, x_3, x_4\}$:

$$x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4$$

$$\neg x_1 \wedge (x_2 \vee x_3 \vee \neg x_4)$$

$$(x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_4)$$

2.2.1 Boolean Simulation

Boolean simulation refers to the act of monitoring the output values of a Boolean function under a given vector of input values. Boolean simulation may be used as part of the verification process in designing hardware.

Definition II.13. An *input vector* $P = \langle p_1, \cdots, p_n \rangle$ of F assigns value $p_i \in \{0, 1\}$ to input $x_i \in X$. The *output vector* $R = \langle r_1, \cdots, r_m \rangle$ that corresponds to input vector P is the result of simulating P with F, where $r_i \in \{0, 1\}$ holds the simulation result for output $z_i \in Z$.

Definition II.14. An input $x_i \in X$ is *observable* to output $z_j \in Z$ (or output z_j is *controllable* by input x_i) with regard to input vector $P = \langle p_1, \cdots, p_n \rangle$ and its corresponding output vector $R = \langle r_1, \cdots, r_m \rangle$, if flipping $p_i \in P$ flips $r_j \in R$.

Intuitively, an input is observable to an output, if the value of that output can be changed just by changing the value of that particular input.

Example II.15. For the Boolean function $z = (x_1 \vee \neg x_1) \wedge x_2 \wedge x_3$, input x_1 is not observable to output z, since $x_1 \vee \neg x_1 = 1$, and hence, $z = x_2 \wedge x_3$.

Definition II.16. Input vector $P = \langle p_1, \cdots, p_n \rangle$ is said to be *proper* with regard to partition $\pi = [W_1 | \cdots | W_t]$ of input set X, if it assigns the same value to all inputs in the same cell of π, i.e., for all i and j, $p_i = p_j$ if $x_i, x_j \in W_l$, for some l.

Example II.17. For input set $X = \{x_1, x_2, x_3\}$, input vector $P = \langle 0, 0, 1 \rangle$ is proper with regard to partition $\pi = [x_1, x_2 | x_3]$, but not proper with regard to partition $\pi' = [x_1 | x_2, x_3]$.

Definition II.18. Two input vectors $P = \langle p_1, \cdots, p_n \rangle$ and $Q = \langle q_1, \cdots, q_n \rangle$ are said to be *consistent* with regard to isomorphic OPP of input set X

$$\Pi = \left[\begin{array}{c} \pi_T \\ \pi_B \end{array}\right] = \left[\begin{array}{c|c|c|c} T_1 & T_2 & \cdots & T_s \\ B_1 & B_2 & \cdots & B_s \end{array}\right]$$

if P is proper with regard to π_T, Q is proper with regard to π_B, and P and Q assign the same value to all inputs in the same-index cells of π_T and π_B, i.e., for all i and j, $p_i = q_j$ if $x_i \in T_l$ and $x_j \in B_l$, for some l.

Example II.19. For input set $X = \{x_1, x_2, x_3\}$, input vectors $P = \langle 0, 0, 1 \rangle$ and $Q = \langle 1, 0, 0 \rangle$ are consistent with regard to the following isomorphic OPP:

$$\Pi = \left[\begin{array}{c|c} x_1, x_2 & x_3 \\ x_2, x_3 & x_1 \end{array}\right]$$

2.3 Graphs

Definition II.20. A *graph* G is composed of a non-empty finite set of *vertices* V together with a set of *edges* E containing pairs of vertices, i.e., $E \subseteq V \times V$.

A *multigraph* is a graph which is permitted to have parallel edges, i.e., edges that have the same end nodes.

Graph vertices are also called *nodes* or *points*, and edges are also called *lines* or *arcs*. If $(a, b) \in E$, we say that vertices a and b are *neighbours* or are *adjacent* to each other. A *loop* (also called a *self-loop*) is an edge that connects a vertex to itself. The *degree* (or *valency*) of a vertex is the number of edges adjacent to that vertex. Graph G is *directed* if its edges have associated directions, and *undirected* otherwise. Graph G is *colored* if its vertices have associated colors.

From the data structure standpoint, a graph can be represented by either an *adjacency matrix* or an *adjacency list*. The *adjacency matrix* of an n-vertex graph

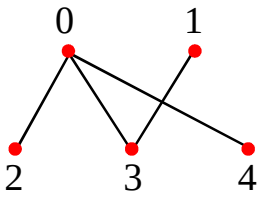


Figure 2.1: A bipartite graph with 5 vertices.

is an $n \times n$ $(0,1)$-matrix where entry $a_{i,j}$ of the matrix is 1 if and only if vertex i is adjacent to vertex j. The *adjacency list* of an n-vertex graph is a collection of n unordered lists $L_1, ..., L_n$, where list L_i contains all the vertices that are adjacent to vertex i.

Adjacency matrix allows constant-time lookup for checking the presence or absence of an edge, but takes linear time to iterate over all edges. On the other hand, adjacency list is fast in iterating over all edges, but slow in checking the presence or absence of an edge. In terms of memory, adjacency matrix takes quadratic space, but adjacency list uses memory in proportion to the number of edges. In general, adjacency matrix is more suitable to represent dense graphs, while adjacency list is more appropriate for sparse graphs.

2.3.1 Bipartite Graphs

Definition II.21. A *bipartite graph* is a graph that divides the set of vertices into two disjoint subsets, such that no two vertices in one subset are adjacent.

Example II.22. Figure 2.1 demonstrates an example of a bipartite graph with 5 vertices.

2.3.2 Miyazaki Graphs

Miyazaki graphs is a family of colored graphs that were introduced by [46] to impede the state of the art in graph symmetry detection. Subsequently, Miyazaki graphs have been used to measure the performance of graph symmetry-detection

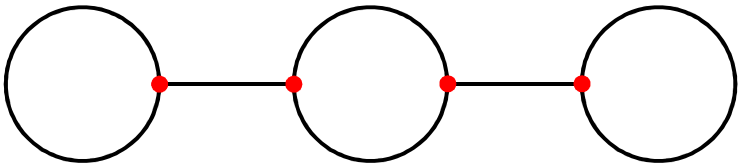


Figure 2.2: Multigraph Y_2 which will be used to construct a Miyazaki graph.

(a) F_3 (b) F_3 rotated 180 degrees

Figure 2.3: F*ü*rer gadgets F_3.

algorithms. In this section, we discuss the construction of Miyazaki graphs.

Let $Y_k(V, E)$ be a multigraph where V and E are defined as:

$V = \{v_i : 1 \leq i \leq 2k\}$,

$E_1 = \{e_l, e_r : e_l = \{v_1, v_1\}, e_r = \{v_{2k}, v_{2k}\}\}$, the self-loops,

$E_2 = \{e_i, e'_i : e_i = e'_i = \{v_{2i+1}, v_{2i+2}\}$ for $1 \leq i \leq k-1\}$, the cycles,

$E_3 = \{e_i : e_i = \{v_{2i-1}, v_{2i}\}$ for $1 \leq i \leq k\}$, the bridges, and

$E = E_1 \cup E_2 \cup E_3$.

Based on the above definition, multigraph Y_k consists of two self-loops (one at each end), and a series of $k-1$ cycles connected to each other via bridges.

Example II.23. Figure 2.2 shows multigraph Y_2 which consists of four vertices and six edges (two self-loops, one cycle, and two bridges).

Miyazaki graphs are constructed by replacing each odd vertex in Y_k by *Fürer gadgets* F_3, and each even vertex by 180-degree rotated F_3. Figure depicts F_3 and 180-degree rotated F_3. The Miyazaki graph constructed from Y_k has $20k$ vertices and $30k$ edges.

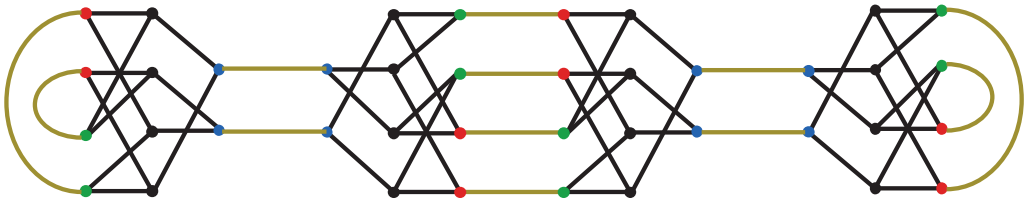


Figure 2.4: Mizayaki graph constructed from multigraph Y_2.

Example II.24. Figure 2.4 shows the Miyazaki graph that is constructed from Y_2. This graph has 40 vertices and 60 edges.

2.3.3 Trees

Definition II.25. A *path* in a graph is a sequence of edges which connect a sequence of vertices. A path with no repeated vertices is called a *simple path*.

Definition II.26. A *tree* is an undirected graph in which any two vertices are connected by exactly one simple path.

In a tree, a *leaf* is a node whose degree is one. Any node that is not a leaf is an *internal node*.

Definition II.27. A *rooted tree* is a tree that has one of its vertices designated as the root.

In a rooted tree, the *parent* of a vertex is the vertex connected to it on the path to the root. Every vertex except the root has a unique parent. A *child* of a vertex v is a vertex of which v is the parent. Every vertex except tree leaves has at least one child. A vertex v is an *ancestor* of a vertex u if it exists on the path from the root to vertex u. The vertex u is then a *descendant* or a *successor* of vertex v. The root is the ancestor of all vertices.

Example II.28. Figure 2.5 depicts a 6-vertex rooted tree with vertex 0 as the root. In this tree, vertices 1, 3, 4 and 5 are the leaves, and vertices 0 and 2 are internal

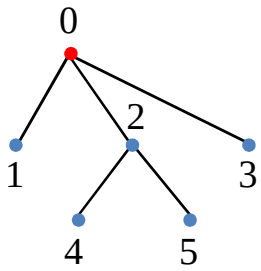


Figure 2.5: A 6-vertex rooted tree with vertex 0 designated as the root.

nodes. The children of vertex 0 are vertices 1, 2 and 3. The parent of vertices 1, 2 and 3 is vertex 0. Vertex 4 is a descendant of vertex 0, and vertex 0 is an ancestor of vertex 4.

Definition II.29. The *depth* (*level*) of a node v in a rooted tree is the length of the path from v to the root. The *height* of v is the length of the longest downward path from v to a leaf.

The height (resp. depth) of a rooted tree is the height (resp. depth) of its root.

Example II.30. For the tree of Figure 2.5, the height and depth of node 1 are zero and one, respectively, while the height and depth of node 2 are both one.

Definition II.31. The *diameter* of a tree is the length of the longest path in the tree. A *center* of a tree is a vertex v such that the longest path from v to a leaf is minimal over all vertices in the tree (i.e., half of the diameter).

A tree has either one center or two centers.

Example II.32. The diameter of the tree in Figure 2.5 is 3. This tree has two centers: vertex 0 and vertex 2.

2.4 Homomorphism

Definition II.33. A *homomorphism* is a function between two combinatorial objects that respects their structure.

For every combinatorial object, there is an underlying notion of *variables set* (for example, the vertex set of a graph and the input/output set of a Boolean function). Homomorphism of two combinatorial objects is, in fact, a mapping between their variable sets that respects their structure.

Example II.34. A graph homomorphism is a mapping between two graphs that respects the edge relation of the graphs.

Two important types of homomorphism are *isomorphisms* and *automorphism*. The next two subsections discuss these two homomorphisms.

2.4.1 Isomorphism

Definition II.35. An *isomorphism* is a bijective homomorphism.

If an isomorphism exists between two combinatorial objects A and B, the two objects are called *isomorphic*, and is denoted by $A \simeq B$.

To better understand isomorphism in combinatorial objects, we need to define the notion of *permutations*.

Definition II.36. Let $X = \{x_1, ...x_n\}$ denote the variable set of a combinatorial object. A *permutation* γ of the object (or the variable set X of the object) is defined as a bijection from X to X.

Similar to the above definition, a permutation from one object with variable set X to another object with variable set Y is a bijection from X to Y.

Example II.37. A permutation of a graph with vertex set V is a bijection from V to V.

Example II.38. A permutation of a Boolean function with input set X and output set Z is a bijection from X to X and Z to Z. A permutation of a Boolean function might also consider the negation of inputs and outputs. In that case, a permutation

of a Boolean function with input set X and output set Z is a bijection from $X \cup X'$ to $X \cup X'$ and $Z \cup Z'$ to $Z \cup Z'$, where X' denotes the set of negated inputs and Z' denotes the set of negated outputs.

Permutations can be expressed in a tabular or a cycle notation. A tabular notation is in a form of a discrete OPP, while a cycle notation comprises a number of simultaneous rotations. The permutation that maps each variable to itself is called the *identity*, and is denoted by ι.

Example II.39. The following are example permutations in tabular and cycle notation on variable set $X = \{x_1, x_2, x_3, x_4\}$:

$$\left[\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \end{array}\right] = \iota$$

$$\left[\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_4 \\ x_1 & x_3 & x_2 & x_4 \end{array}\right] = (x_2\ x_3)$$

$$\left[\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_4 \\ x_2 & x_3 & x_1 & x_4 \end{array}\right] = (x_1\ x_2\ x_3)$$

$$\left[\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_4 \\ x_2 & x_1 & x_4 & x_3 \end{array}\right] = (x_1\ x_2)(x_3\ x_4)$$

For two combinatorial objects, an isomorphism is a permutation (from the variable set of one object to the variable set of the other object) that respects the structure of the two objects.

Example II.40. An isomorphism between two graphs is a permutation of vertices that respects the edge relation of the graphs.

Example II.41. An isomorphism of a Boolean function is a permutation of inputs

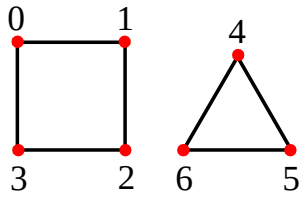


Figure 2.6: The "square and triangle" graph with 7 vertices and 7 edges.

and outputs (with their possible negation) that respects the value of the functions for all input combinations.

2.4.2 Automorphism (Symmetry)

Definition II.42. An *automorphism* (*symmetry*) is an isomorphism of a combinatorial object to itself.

In fact, an automorphism is a permutation that leaves the object unchanged.

Example II.43. A symmetry of a graph is a permutation of the graph's vertices that preserves the graph's edge relation. For example, permutation (0 2) is a symmetry of the square and triangle graph of Figure 2.6.

Example II.44. A symmetry of a Boolean function is a permutation of the function's inputs and outputs (with their possible negation) that preserves the value of the function for all input combination. For example, permutation $(x_1\ x_2)$ is a symmetry of the Boolean function $z = (x_1 \vee x_2) \wedge x_3$.

2.5 Permutation Group Theory

We assume familiarity with basic notions from group theory, including such concepts as *subgroups*, *cosets*, *group generators*, *group action*, *orbit partition*, etc. We review most of these concepts here, but additional information on them can be found in standard textbooks on abstract algebra, e.g. [28].

2.5.1 Groups

Definition II.45. A *group* is a set $\mathcal{G}$ together with a binary operation $\cdot$ that satisfies the following four group axioms:

1. Closure: for all elements $a, b \in \mathcal{G}$, $a \cdot b \in \mathcal{G}$.
2. Associativity: for all elements $a, b, c \in \mathcal{G}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
3. Identity Element: there exists an element $e \in \mathcal{G}$ such that for every element $a \in \mathcal{G}$, $e \cdot a = a \cdot e = a$.
4. Inverse Element: for every element $a \in \mathcal{G}$, there exists an element $a^{-1} \in \mathcal{G}$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$.

Example II.46. The set of all integers $\mathbb{Z}$ with addition forms a group, since:

For all integers $i, j \in \mathbb{Z}$, $i + j \in \mathbb{Z}$.

For all integers $i, j, k \in \mathbb{Z}$, $(i + j) + k = i + (j + k)$.

The identity element is 0: for all $i \in \mathbb{Z}$, $0 + i = i + 0 = i$.

The inverse element for $i \in \mathbb{Z}$ is $-i$: $i + (-i) = (-i) + i = 0$.

Definition II.47. A *generating set* S for group $\mathcal{G}$ is a subset $S \subset \mathcal{G}$ such that every element of $\mathcal{G}$ can be expressed as the combination (under the group operation) of finitely many elements of S.

Example II.48. The set $S = \{-1, 1\}$ is a generating set for the group from Example II.46, since:

$$0 = 1 + (-1)$$

For integer $i > 0$, $i = \underbrace{1 + 1 + ... + 1}_{i}$

For integer $i < 0$, $i = \underbrace{(-1) + (-1) + ... + (-1)}_{i}$

Definition II.49. A *subgroup* $\mathcal{H}$ of group $\mathcal{G}$ is a subset of $\mathcal{G}$ that forms a group under $\mathcal{G}$'s binary operation.

Example II.50. Let set $\mathbb{Z}_2 = \{2 \times i \mid i \in \mathbb{Z}\} = \{..., -2, 0, 2, ...\}$, where $\mathbb{Z}$ is the set of all integers. Set $\mathbb{Z}_2$ with addition forms a group. This group is a subgroup of the group from Example II.46.

Definition II.51. For group $\mathcal{G}$ and its subgroup $\mathcal{H}$, the *(right) coset* of $\mathcal{H}$ containing element $a \in \mathcal{G}$ is the set $\{h \cdot a \mid h \in \mathcal{H}\}$.

Based on the above definition, *any* coset element can generate the entire coset by composing that element with the elements of $\mathcal{H}$. Choosing one element from each coset yields a set of *coset representatives*. The set of all cosets of $\mathcal{H}$ in $\mathcal{G}$ partitions G into equally-sized subsets.

Example II.52. Let $\mathbb{Z}_2$ be the group from Example II.50, and $\mathbb{Z}$ be the group from Example II.46:

> The coset of $\mathbb{Z}_2$ in $\mathbb{Z}$ containing integer 0 is the set $\{2 \times i \mid i \in \mathbb{Z}\} = \{..., -2, 0, 2, ...\}$, and
>
> The coset of $\mathbb{Z}_2$ in $\mathbb{Z}$ containing integer 1 is the set $\{2 \times i + 1 \mid i \in \mathbb{Z}\} = \{..., -3, -1, 1, 3, ...\}$.

Definition II.53. For a group $\mathcal{G}$ and a set X, the *group action* of $\mathcal{G}$ on X is a function $* : \mathcal{G} \times X \mapsto X$ that satisfies the two following axioms:

1. Identity: for all $x \in X$, $e * x = x$.
2. Associativity: for all $a, b \in \mathcal{G}$ and all $x \in X$, $(a \cdot b) * x = a * (b * x)$.

Example II.54. The *trivial* action of any group $\mathcal{G}$ on any set X is defined by $a * x = x$ for all $a \in \mathcal{G}$ and all $x \in X$.

Definition II.55. For a group $\mathcal{G}$ that acts on set X, the *stabilizer subgroup* of $x \in X$ is $\mathcal{G}_x$ that fixes x, i.e., $\mathcal{G}_x = \{a \in \mathcal{G} \mid a * x = x\}$

Example II.56. For the trivial action of group $\mathcal{G}$ on set X (see Example II.54), the stabilizer subgroup of any $x \in X$ is $\mathcal{G}$.

Theorem II.57. *Let $\mathcal{G}$ be a group that acts on set X. For $x_1, x_2 \in X$, let $x_1 \sim x_2$ if and only if there exists $a \in \mathcal{G}$ such that $a * x_1 = x_2$. Then, $\sim$ is an equivalence relation on X.*

The equivalence relation $\sim$ partitions X into a so-called *orbit partition* Θ. The cell of the orbit partition that contains $x \in X$ is called the *orbit* of X, and is denoted by Θ_x.

Example II.58. For the trivial action of group $\mathcal{G}$ on set $X = \{x_1, x_2, ..., x_n\}$ (see Example II.54), the orbit partition of set X is the discrete partition on set X, i.e., $\Theta = \{x_1, x_2, \cdots, x_n\}$.

2.5.2 Permutation Groups

Definition II.59. A *permutation group* is a group $\mathcal{G}$ whose elements are permutations of a variable set X, and whose group operation is a functional composition.

Example II.60. The set of all permutations of any set X forms a permutation group. For $X = \{x_1, x_2, x_3\}$, this group consists of 6 permutations:

$$\{\iota, (x_1\ x_2), (x_1\ x_3), (x_2\ x_3), (x_1\ x_2\ x_3), (x_1\ x_3\ x_2)\}$$

Examples of functional composition on this permutation group are:

$$\iota \circ (x_1\ x_2) = (x_1\ x_2)$$
$$(x_1\ x_2) \circ (x_1\ x_2) = \iota$$
$$(x_1\ x_2) \circ (x_1\ x_3) = (x_1\ x_3\ x_2)$$
$$(x_1\ x_2\ x_3) \circ (x_1\ x_2\ x_3) = (x_1\ x_3\ x_2)$$

2.5.3 Symmetry Groups

The set of all symmetries of a combinatorial object forms a group under functional composition. This group is referred to as the *symmetry group* of the object.

Example II.61. The set of all symmetries of a square graph of Figure 2.6 contains:

$$\{\iota, (0\ 2), (1\ 3), (0\ 2)(1\ 3), (0\ 1)(2\ 3), (0\ 3)(1\ 2), (0\ 1\ 2\ 3), (0\ 3\ 2\ 1)\}$$

This set forms a group under functional composition.

A symmetry group of an object acts on the object's variables, and partitions the variable set of the object (i.e., orbit partition).

Example II.62. The orbit partition for the symmetry group of Example II.61 is $\{0, 1, 2, 3\}$.

CHAPTER III

Previous Work

In this section, we review previous work relevant to this thesis.

3.1 And-Inverter Graphs

An *AIG* is a directed acyclic graph that represents the functionality of a Boolean function. The nodes of an AIG are two-input "And" gates, and its edges are optionally marked to indicate "Not" gates.

Example III.1. Figure 3.1 shows two structurally different AIGs for the function $z = x_2 \wedge (x_1 \vee x_3)$.

Modern logic synthesis tools, such as ABC, use AIGs as alternatives to *Binary Decision Diagrams (BDDs)*, since AIGs are more memory efficient, and are faster in performing logic simulation. Unlike BDDs, AIGs are not canonical, but are *structurally hashed* to be partially canonical [45].

3.2 Boolean Satisfiability

Boolean satisfiability (*SAT*) is the problem of determining if there exists a variable assignment to a Boolean formula that makes the formula evaluate to true. A

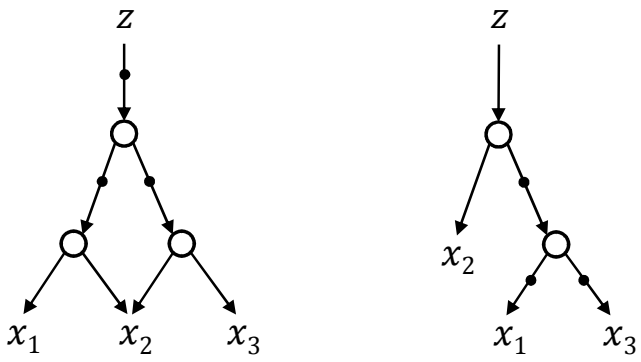


Figure 3.1: Two structurally different AIGs for the function $z = x_2 \wedge (x_1 \vee x_3)$.

Boolean formula is said to be *satisfiable* if such an assignment exists, and *unsatisfiable* otherwise.

Example III.2. The following are examples of satisfiable and unsatisfiable CNF formulas on variable set $X = \{x_1, x_2, x_3\}$:

$x_1 \wedge x_2 \wedge x_3$:	satisfiable under $x_1 = 1, x_2 = 1, x_3 = 1$
$\neg x_1 \wedge (x_2 \vee \neg x_3)$:	satisfiable under $x_1 = 0, x_2 = 1, x_3 = 1$
$x_1 \wedge \neg x_1 \wedge (x_2 \vee x_3)$:	unsatisfiable

The first algorithms to solve the satisfiability problem were introduced in the early 1960s. These algorithms are now referred to as the DPLL search framework [23, 22]. DPLL consists of three main features: *branching*, *backtracking*, and *unit propagation*. Branching is essential to move forward in the search space, and backtracking is used to retreat from futile branches of the search. Unit propagation expedites the search by detecting futile branches early (based on the decisions taken) and triggering backtracking.

Modern SAT solvers, such as **MiniSAT** [24], augment DPLL with the following concepts:

– *Clause learning*: it is likely for basic DPLL to encounter the same chain of *conflicting* assignments multiple times during the search. To avoid such redundancy, 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 the propagation process to avoid the same conflict in the future [54].

- *VSIDS adaptive branching*: VSIDS is a low-overhead branching heuristic that attempts to satisfy (the most recent) conflict clauses [47].

- *Watched literals*: two-literal watching refers to a data structure and related algorithms that speeds up unit propagation [47].

- *Random restarts*: using restarts is a typical strategy to escape from futile parts of the search space [31].

3.3 Combinational Equivalence Checking

Equivalence checking of two *combinational* circuits is the problem of checking whether two circuits are functionally equivalent, i.e., they exhibit the same output values under combinations of all input values.

One way to prove functional equivalence of two combinational circuits is to use SAT. This is accomplished by building the *miter* (see below) of the two circuits and passing it to a SAT solver.

The miter of two circuits is a single-output circuit constructed by combining inputs with the same name, feeding outputs with the same name to two-input XOR gates, and connecting the outputs of the XOR gates to one multi-fanin OR gate. Figure 3.2 visualizes miter construction for two logic circuits C_1 and C_2 with input sets $X = \{x_1, ..., x_n\}$, and output sets $Z = \{z_1, .., z_m\}$ and $W = \{w_1, ..., w_m\}$, respectively.

If the miter of two circuits is unsatisfiable, the circuits are equivalent. If not, the circuits are not equivalent, and the satisfiable assignment serves as a counterexample. This is due to the property of the two-input XOR gate, which produces 1 if and only if the two input values are the same.

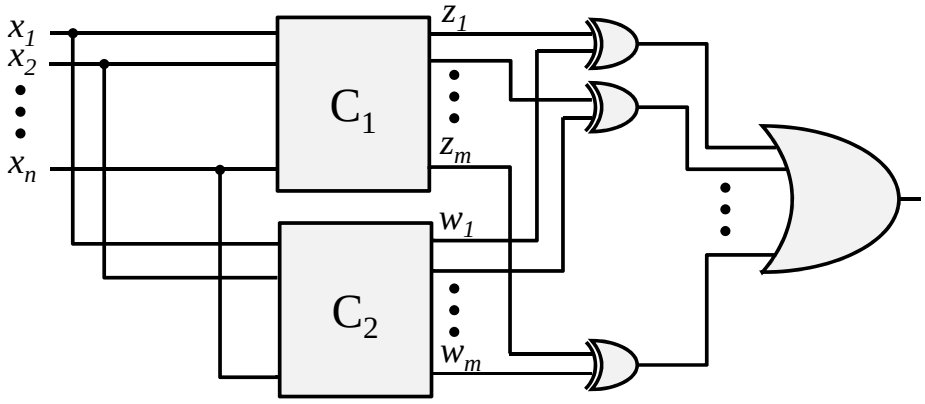


Figure 3.2: Miter of two n-input m-outputs logic circuits.

3.4 Software Tools for Graph Symmetry Detection and Canonical Labeling

Graph symmetry and canonical labeling have been extensively studied over the past five decades. The **nauty** program [43] pioneered the first high-performance algorithms that inspired all subsequent tools. The **nauty** algorithms used a depth-first branching and backtracking framework which integrated group-theoretic techniques to facilitate the search. Those algorithms were primarily designed to search for a graph canonical labeling, but were also able to report graph symmetries as a by-product.

Closely following **nauty**'s canonical-labeling algorithms were three other tools, namely, **bliss** [34, 35], **traces** [50] and **nishe** [55]. The search routines in **bliss** improved the handling of large and sparse graphs, since the algorithms in **nauty** were mostly designed to target small dense graphs. The breadth-first scan of branching choices in **traces** allowed early identification and elimination of futile branches of search. The branching heuristics in **nishe** facilitated a polynomial-time solution for a family of graphs known as Miyazaki [46], which had been shown to impede **nauty** algorithms.

In addition to the above software packages, another software package, called **saucy** (presented in this thesis), was optimized to only look for symmetries of graphs. The

first version of **saucy** was motivated by the observation that the graphs of typical CNF formulas were too large (hundreds of thousands to millions of vertices) and unwieldy for **nauty** which was more geared towards small dense graphs (hundreds of vertices). The obvious remedy, changing the data structure for storing graphs from an incidence matrix to a linked list, yielded the **saucy** system which demonstrated the viability of graph automorphism detection on very large sparse graphs [20].

The next version of the **saucy** tool [21] introduced a major algorithmic change that delinked the search for symmetries from the search for a canonical labeling. This yielded a remarkable 1000-fold improvement in run time for many large sparse graphs with sparse symmetry generators, i.e., generators that "move" only a tiny fraction of the graph's vertices.

In this thesis, we take a fresh look at **saucy** algorithms. We explain different aspects of **saucy** search tree by viewing permutation sets as ordered partition pairs. We also introduce the third version of **saucy** which uses *simultaneous partition refinement* to anticipate and avoid conflicts that might arise during the search. The **saucy** search algorithms are presented in Chapter IV.

3.5 Computational Complexity of Graph Automorphism

The graph isomorphism problem belongs to the class NP of computational complexity. This is justified by the fact that the "Yes" answer to graph isomorphism (i.e., whether two graphs are isomorphic or not) can be verified in polynomial time on a deterministic Turing machine. The graph isomorphism problem, although in NP, is one of few standard problems in computational complexity theory that is not known to be either in P or NP-complete.

The following problems are polynomial-time equivalent to graph isomorphism [12, 42], and hence, all arguments on the computational complexity of graph isomorphism hold for them as well:

1. finding a set of generators for the automorphism group of a graph,

2. computing the size of the automorphism group of a graph, and

3. finding the orbits of the automorphism group of a graph.

While no polynomial-time algorithm is known to solve the isomorphism problem for general graphs, some progress has been made towards polynomial-time algorithms for special cases. Examples of such cases include planar graphs [33], graphs of bounded genus [26, 44], bounded degree graphs [41], graphs with bounded eigenvalue multiplicities [8], and trees [3].

Furthermore, strong evidence against NP-completeness of graph isomorphism has been provided in the literature. For instance, Mathon [42] has shown that counting the number of graph isomorphisms is polynomial-time equivalent to deciding the existence of an isomorphism. This is while the counting version of a typical NP-complete problem tends to be much harder than its decision version.

Regarding the current state of knowledge on the complexity of graph isomorphism, it seems that if graph isomorphism belongs to either P or NP-complete, it is more likely to be in P than NP-complete. This conjecture is also somewhat validated by today's fast and highly scalable graph isomorphism (and automorphism) packages.

3.6 Symmetry Breaking for Boolean Satisfiability

One known source of deficiency in modern SAT solvers is their tendency to explore *symmetric* portions of the search space [51, 19, 4, 7, 29, 30]. Triggered by this observation, Crawford et al. [19] established the theoretical framework that breaks *all* the symmetries of a CNF formula. The idea is to use symmetries to augment the formula with a set of *Symmetry-Breaking Predicates (SBPs)*. These predicates do not change the formula's satisfiability, but help the solver prune away symmetric portions of the search space. Since this type of symmetry breaking refines SAT search space

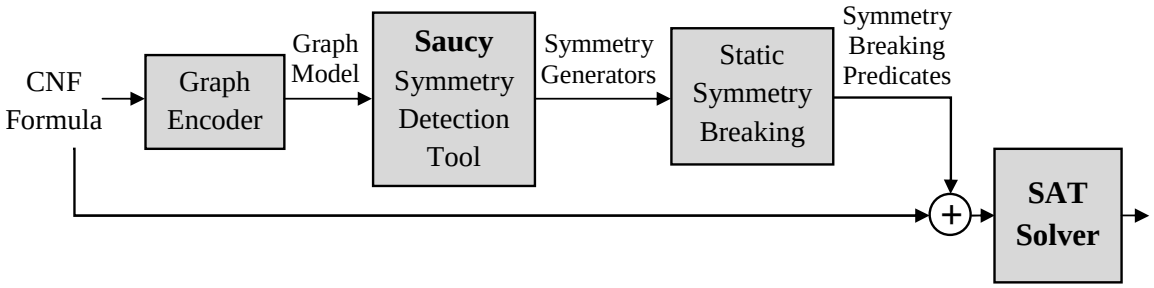
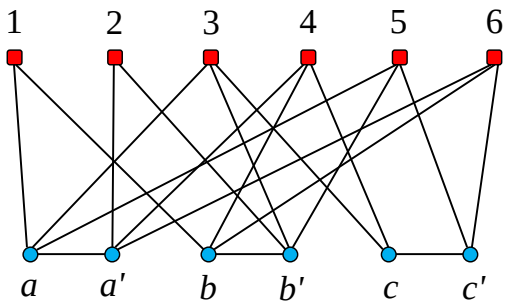


Figure 3.3: The **shatter** static symmetry-breaking flow.

by modifying the input formula and not the SAT algorithms, it is referred to as *static symmetry breaking*.

It quickly became apparent from theory that generating an SBP for every symmetry of a formula was impractical, since the number of the formula's symmetries can be *exponential* in the number of its variables. Therefore, Aloul et al. [5] proposed *partial* symmetry breaking by generating SBPs for just a subset of all symmetries. The rationale is that 1) it is not necessary to break all symmetries to rule out all symmetric solutions, and 2) a small number of symmetric solutions can be tolerated. The subset they used was the formula's symmetry group generators which was readily available from running a common symmetry-detection tool. Further advances in symmetry breaking were reported by Aloul et al. in [4] and [6], including an SBP-augmented CNF formula whose number of SBPs grew *linearly* with the number of variables of the original formula. An automated static symmetry-breaking framework that integrated the most optimized symmetry-breaking algorithms was implemented in a tool called **shatter** [4].

The **shatter** static symmetry-breaking flow is depicted in Figure 3.3. This flow consists of four consecutive steps. First, the input CNF formula is modeled as a graph. Then, the modeled graph is passed to a graph symmetry-detection tool to find the formula's symmetry generators. Next, symmetry-breaking routines generate a set of symmetry-breaking predicates based on the formula's symmetry information.



$$\phi = (a \vee b) \wedge (a' \vee b') \wedge (a \vee b' \vee c) \wedge (a' \vee b \vee c) \wedge (a \vee b' \vee c') \wedge (a' \vee b \vee c')$$

Figure 3.4: A 3-variable 6-clause CNF formula ϕ along with its modeled graph G.

Finally, the original formula, augmented with SBPs, is handed off to a SAT solver.

Given a CNF formula, its modeled graph is constructed as follows. A vertex is added for each clause and each literal of the formula. Clause vertices are colored differently than literal vertices, since clauses should only map to clauses and literals to literals. An edge is added between clause vertex i and literal vertex j, if and only if clause i contains literal j. An edge is also drawn between each literal and its negation.

Example III.3. Figure 3.4 depicts an example of a 3-variable 6-clause CNF formula ϕ, along with its modeled graph G. Vertices 1 to 6 correspond to ϕ's clauses (from left to right, respectively), and the remaining vertices are labeled with ϕ's literals.

A symmetry-breaking predicate is a *lex-leader predicate* that evaluates to true for at least one element from each orbit of the formula's symmetry group [6]. In other words, it picks one representative assignment from each orbit.

Example III.4. For the formula ϕ of Figure 3.4, permutation $\gamma = (a\ b)$ forms a symmetry. This symmetry indicates that the value of ϕ stays the same under assignment $a = 0, b = 1$, and assignment $a = 1, b = 0$. The SBP that corresponds to γ is $a' \vee b$, which only allows one of the two above assignments, i.e., $a = 0$ and $b = 1$. Formula ϕ, when augmented with this SBP, is:

$$\phi = (a \vee b) \wedge (a' \vee b') \wedge (a \vee b' \vee c) \wedge (a' \vee b \vee c) \wedge (a \vee b' \vee c') \wedge (a' \vee b \vee c') \wedge (a' \vee b)$$

CHAPTER IV

Graph Symmetry-Discovery Algorithms

In this chapter, we describe our symmetry-detection algorithms for graphs. The collection of our algorithms is implemented in the **saucy** 3.0 graph symmetry-detection software. Throughout this chapter, we assume that the input to **saucy** is an n-vertex undirected colored graph G with vertex set $V = \{0, 1, ..., n-1\}$.

4.1 The Permutation Search Space

All graph symmetry-detection software packages, except for **saucy**, are primarily designed to solve the canonical labeling problem. Examples of such packages include **nauty** [43], **bliss** [34, 35], **traces** [50] and **nishe** [55], which explore the space of graph *labelings*, and find symmetry group generators as a byproduct. These packages employ group-theoretic pruning heuristics to narrow the search for the canonical labeling of an input graph. The detection of symmetries benefit from these pruning rules, but also help prune the canonical-labeling tree, since symmetric graphs yield the same labeling.

Unlike graph canonical labelers which search the space of labelings, **saucy** finds symmetries by exploring the space of *permutation*. In fact, it builds a complete *permutation* search tree, and prunes its futile branches using algorithmic and group-theoretic techniques. The data structures and algorithms in **saucy** are optimized to

implicitly encode and manipulate sets of permutations. They take advantage of both the sparsity of an input graph and the sparsity of its symmetries to attain scalability.

4.2 Implicit Representation of Permutation Sets

To represent sets of permutations, **saucy** uses a data structure that encodes ordered partition pairs (OPPs). The goal of this data structure is to provide a *compact* and *implicit* representation of sets of permutations. Specifically, a discrete OPP represents a single permutation, whereas a unit OPP represents all $n!$ permutations of the vertices. In general, an isomorphic OPP

$$\Pi = \left[\begin{array}{c|c|c|c} T_1 & T_2 & \cdots & T_m \\ B_1 & B_2 & \cdots & B_m \end{array}\right] \tag{4.1}$$

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

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

Discrete OPP: $\left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & 2 & 0 \end{array}\right] = \{(0\ 2\ 1)\}$

Unit OPP: $\left[\begin{array}{c} 0,1,2 \\ 0,1,2 \end{array}\right] = \{\iota, (0\ 1), (0\ 2), (1\ 2), (0\ 1\ 2), (0\ 2\ 1)\}$

Isomorphic OPP: $\left[\begin{array}{c|c} 2 & 0,1 \\ 1 & 2,0 \end{array}\right] = \{(1\ 2), (0\ 2\ 1)\}$

Matching OPP: $\left[\begin{array}{c|c|c} 1 & 0,2,4 & 3 \\ 3 & 0,2,4 & 1 \end{array}\right] = (1\ 3) \circ \{\iota, (0\ 2), (0\ 4), (2\ 4), (0\ 2\ 4), (0\ 4\ 2)\}$

Non-isomorphic OPPs: $\left[\begin{array}{c|c|c} 0,2 & & 1 \\ 1 & & 2,0 \end{array}\right] = \emptyset, \left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & & 2,0 \end{array}\right] = \emptyset$

4.3 Basic Enumeration of the Permutation Search Space

The basic skeleton of **saucy**'s permutation enumeration algorithm is formed by extending isomorphic OPPs, since isomorphic OPPs encode non-empty sets of permutations. The root of the permutation tree is a unit OPP which encodes all possible permutations of an input graph.

An isomorphic OPP is extended by the routine shown in Figure 4.1. This routine first picks a non-singleton cell (*target* cell) from the top partition. It then picks a vertex (*target* vertex) from the target cell, and maps the target vertex to a vertex (*candidate* vertex) from the corresponding cell of the bottom partition.

The mapping step is accomplished by splitting the target cell so that the target vertex is in a cell of its own. The corresponding cell of the bottom partition is split similarly, placing the vertex to which the target vertex is mapped in a new singleton cell. Given the isomorphic OPP in (4.1), Figure 4.2 symbolically illustrates the mapping procedure. It assumes that the ith cell is the target cell, $j \in T_i$ is the target vertex, and j is mapped to $k \in B_i$. This mapping refines the m-cell OPP Π to the $(m+1)$-cell OPP Π'.

Using the mapping procedure illustrated in Figure 4.2, **saucy**'s enumeration algorithm constructs a complete permutation tree. The structure of this tree is depicted in Figure 4.3. For this figure, assume that the vertex set is $V = \{v_0, v_1, ..., v_{n-1}\}$. At each level of the permutation tree, a target vertex is picked and mapped to all

Inputs: Π
Outputs: $\tilde{\Pi}$

1. Set π_T to the top partition of OPP Π.
   Set π_B to the bottom partition of OPP Π.
2. Choose a non-singleton cell C_T (the *target* cell) from partition π_T.
   Set cell C_B to the corresponding cell of C_T on partition π_B (i.e., the cell that goes under C_T on the bottom partition).
3. Choose a vertex v from target cell C_T (the *target* vertex).
4. Map target vertex v to a vertex u (the *candidate* vertex) from cell C_B and save the result in $\tilde{\Pi}$ (see Figure 4.2 for the pseudocode of mapping).
5. return $\tilde{\Pi}$.

Figure 4.1: The basic skeleton for **saucy**'s permutation enumeration algorithm.

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

where

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

Figure 4.2: The results of refining the OPP Π in (4.1) assuming that the ith cell is the target cell, $j \in T_i$ is the target vertex, and j is mapped to $k \in B_i$.

possible candidate vertices. For example, the target vertex at level 1 is v_i, at level 2 is v_j, ..., and at level $n-1$ is v_k. The mapping procedure continues until a discrete OPP (i.e., a leaf) is reached at level $n-1$. Note that mapping beyond level $n-1$ is not possible, since a discrete OPP does not have any non-singleton cells. The permutation enumeration ends when all possible mappings are exhausted. In implementation, a depth-first traversal of the permutation tree is performed.

Once the basic enumeration is complete, all $n!$ permutations of the graph are visited at the leaves. These permutations can then be checked to see if they form

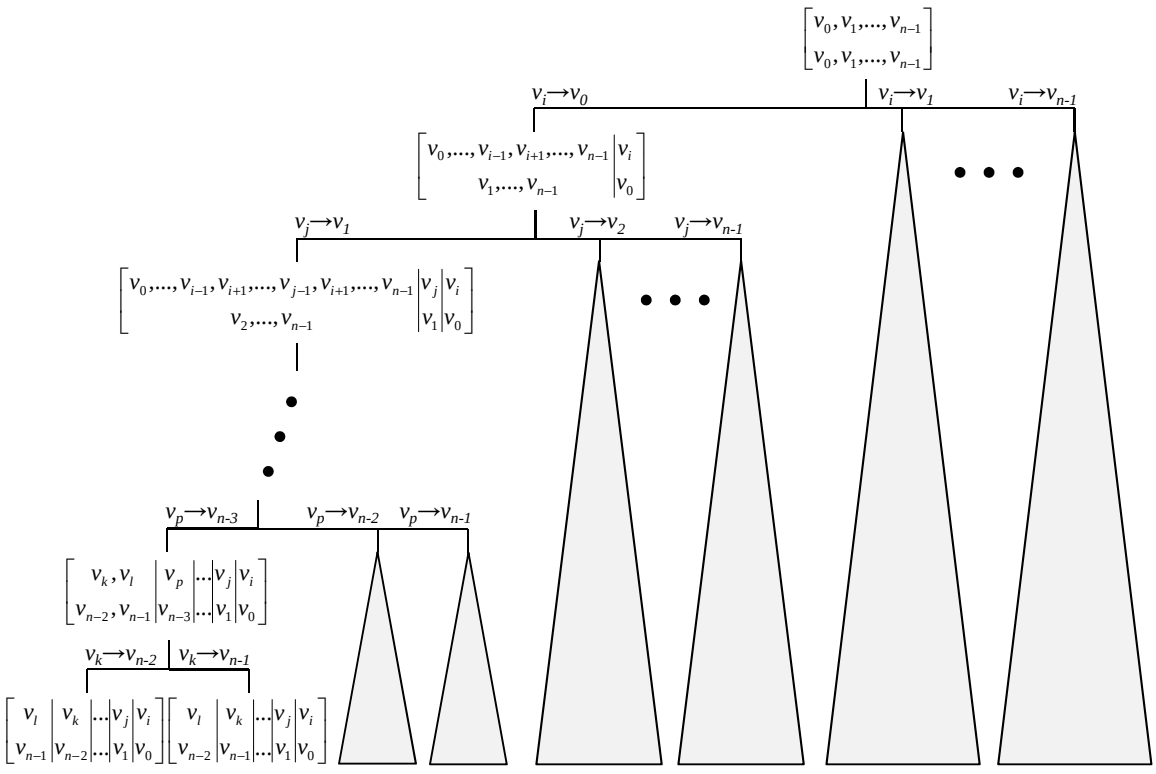


Figure 4.3: The structure of **saucy**'s permutations search tree.

symmetries of the input graph. It is important to point out that the choice of target vertex at each tree node and the order in which each of its possible mappings are processed does not affect the final set of permutations produced at the leaves of the search tree. It does, however, alter the order in which these permutations are produced.

As an example, consider the search tree in Figure 4.4(a) which enumerates all permutations of $V = \{0, 1, 2\}$, and checks which are valid symmetries of the indicated 3-vertex 2-edge graph. Each node of the search tree corresponds to an OPP whose cells contain the vertices of the graph. For example, the root of the tree is a unit OPP that represents all 6 permutations on 3 vertices. Each non-discrete OPP is the root of a subtree that is obtained by mapping a target vertex in all possible ways. For example, the unit OPP at the root of the search tree is extended into a 3-way

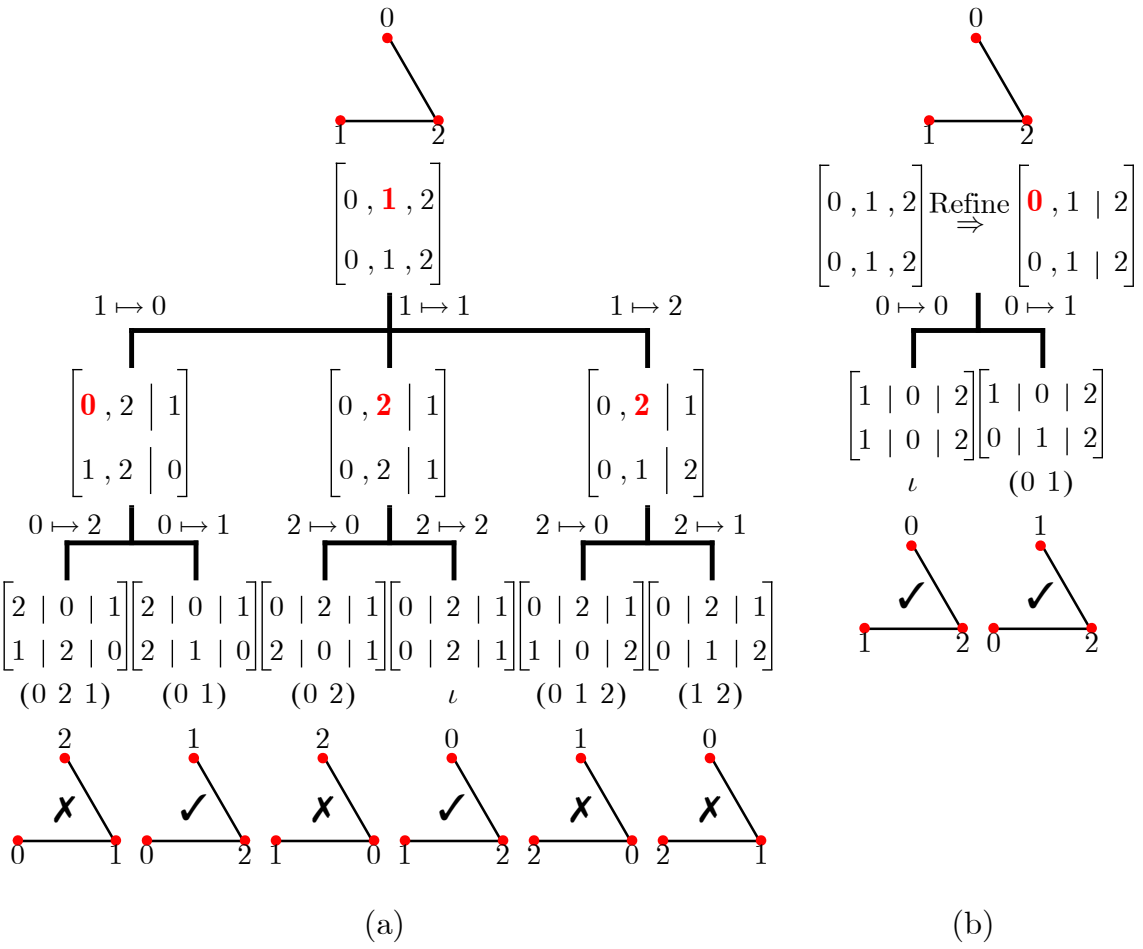


Figure 4.4: Search trees for the automorphisms of a 3-vertex "line" graph. The target vertex ("decision variable") at each tree node is highlighted. (a) without partition refinement. (b) with partition refinement.

branch by mapping target vertex 1 to 0, 1, and 2.

4.4 Partition Refinement

The **saucy** permutation search tree can be pruned significantly by performing *partition refinement* [3, 20, 43] before selecting and branching on a target vertex. The goal of partition refinement is to propagate the constraints of the graph (i.e, the graph's vertex colors, vertex degrees, and edge relation) until the partition becomes *equitable*.

Inputs: G, $\pi = [W_1|W_2|\cdots|W_t]$
Outputs: $\tilde{\pi}$

1. Set partition $\tilde{\pi}$ to π.
   Set integer m to 1.
2. If $\tilde{\pi}$ is discrete or $m > t$, then return $\tilde{\pi}$.
3. Set cell W to W_m.
   Increment m.
   Set integer k to 1.
   Suppose that $\tilde{\pi} = [V_1|V_2|\cdots|V_r]$ at this point.
4. Define $V_k = |X_1|X_2|\cdots|X_s|$ such that for any $x \in X_i$ and $y \in X_j$, the number of neighbors of x in W is smaller than the number of neighbors of y in W if and only if $i < j$.
5. If $s == 1$, go to 7.
6. Let z be the smallest integer such that $|X_z|$ is maximum for $1 \leq z \leq s$.
   If $W_j == V_k$ for some j $(m \leq j \leq t)$, set W_j to X_z.
   For $1 \leq i < z$, set W_{t+i} to X_i.
   For $z < i \leq s$, set W_{t+i-1} to X_i.
   Set t to $t + s - 1$.
   Update $\tilde{\pi}$ by replacing cell V_k with cells $X_1, X_2, ..., X_s$ in that order.
7. Increment k. If $k \leq r$, go to 4.
8. Go to 2.

Figure 4.5: Pseudocode for refining partition π = given graph G.

Definition IV.2. Partition $\pi = [W_1|W_2|\cdots|W_t]$ is said to be equitable (with respect to a given graph) if, for all vertices $v_1, v_2 \in W_i$ $(1 \leq i \leq t)$, the number of neighbors of v_1 in W_j $(1 \leq j \leq t)$ is equal to the number of neighbors of v_2 in W_j.

Figure 4.5 shows the pseudocode for partition refinement (as described in [43]). This pseudocode takes a graph G and an ordered partition $\pi = [W_1|W_2|\cdots|W_t]$, and produces equitable ordered partition $\tilde{\pi}$ from π with respect to G. An example of partition refinement is illustrated in Figure 4.4(b) where vertex 2 is split from vertices 0 and 1 because it has a different degree.

In the early versions of **saucy**, partition refinement was applied *separately* to the top and bottom partitions of an OPP. In implementation, **saucy** first refined the top

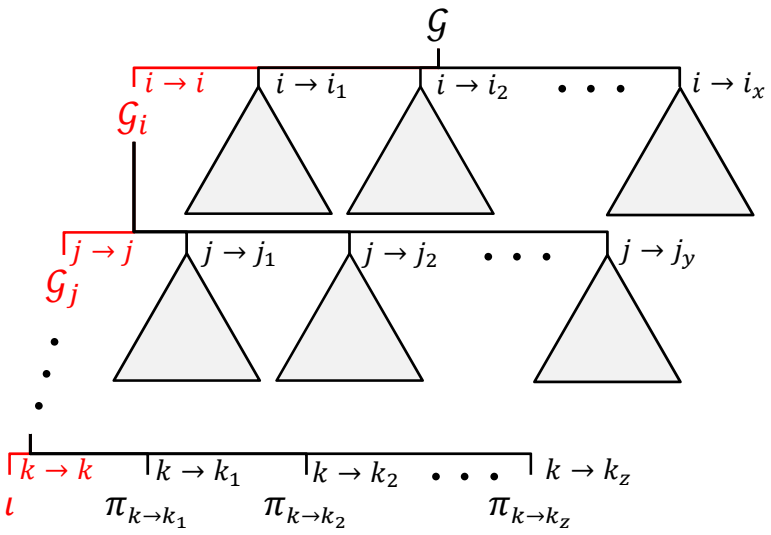


Figure 4.6: Structure of the permutation search tree.

partition until it became equitable, and recorded where the cell splits occurred. Then, it refined the bottom partition, and compared the splitting locations of the bottom to the top (i.e., checked the isomorphism of the two partitions).

In **saucy** 3.0, however, partition refinement is applied *simultaneously* to the top and bottom partitions. This new refinement scheme (explained in detail in Section 4.8) has the extra advantage of anticipating conflicts that might be overlooked by conventional refinement.

4.5 Group-Theoretic Pruning

There are two primary pruning mechanisms anchored in group theory: *coset pruning* and *orbit pruning*. These pruning techniques are routinely employed by symmetry-detection and canonical-labeling algorithms, such as **saucy**.

To enable these pruning mechanisms in **saucy**, we re-structure the permutation search tree so that each tree level represents a subgroup of the graph's symmetry group along with its potential cosets. Figure 4.6 illustrates this re-structuring. For this figure, assume that the symmetry group of the input graph is $\mathcal{G}$ which is shown at

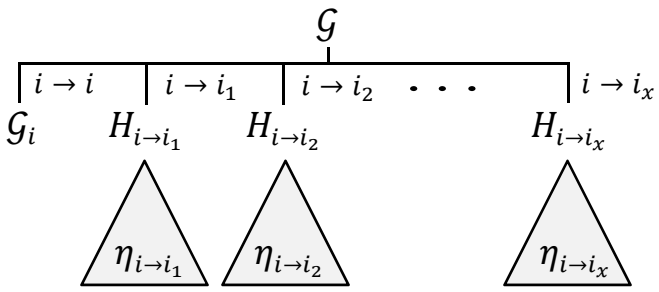


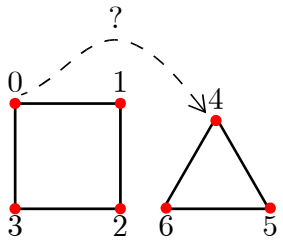
Figure 4.7: A coset level of the permutation search tree.

the root. The "decisions" along the left-most tree path map each selected target vertex to itself. In other words, this path corresponds to a sequence of *subgroup stabilizers* ending in the identity. This phase of the search is called *subgroup decomposition*, and the sequence of the mappings on the left-most path is called the *stabilizer sequence*. In the following sub-sections, we explain how the new re-structuring of the permutation tree (as shown in Figure 4.6) can help us find a set of generators for the symmetry group of the graph.

4.5.1 Coset Pruning

Consider the level of the search tree in Figure 4.6 where vertex i is mapped to vertices $i, i_1, i_2, \cdots, i_x$. This part of the search is illustrated in more details in Figure 4.7. The permutation subset corresponding to mapping i to itself is $\mathcal{G}_i$, the stabilizer subgroup of i. The other subsets, denoted by $H_{i \mapsto i_j}$, correspond to those permutations that, among other things, map i to i_j.

To find a set of generators for $\mathcal{G}$ in Figure 4.6, we must "solve" up to x independent problems where problem i_j seeks to determine whether the set of permutations $H_{i \mapsto i_j}$ is a coset of $\mathcal{G}_i$. This is accomplished by searching $H_{i \mapsto i_j}$ for a *single permutation* that satisfies the graph edge relation, i.e., a permutation that is an automorphism of the graph. If no such permutation exists, then $H_{i \mapsto i_j}$ is empty, i.e., it is not a coset of $\mathcal{G}_i$.



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

Figure 4.8: Example of non-isomorphic refinement. Attempting to map vertex 0 to vertex 4 causes the top and bottom partitions to split non-isomorphically into 4 and 3 cells, respectively.

4.5.2 Orbit Pruning

Let permutation $\eta_{i \mapsto i_j}$ denote the "solution" to problem i_j (see Figure 4.6). Clearly, $\eta_{i \mapsto i_j}$ serves as a coset representative for $H_{i \mapsto i_j}$ and can be added to the set of generators for $\mathcal{G}$. Additionally, vertices i and i_j must now be in the same orbit. Thus, if the orbit of i_j contains vertex i_l with $l > j$, then problem i_l can be skipped since its corresponding coset must necessarily contain redundant generators.

4.6 OPP-Based Pruning

The search algorithms in **saucy** incorporate two algorithmic pruning techniques that are enabled due to the OPP encoding of permutation sets: *non-isomorphic OPP pruning* and *matching OPP pruning*.

4.6.1 Non-isomorphic OPP pruning

Once partition refinement is complete (i.e., the top and bottom partitions of an OPP are equitable), **saucy** checks whether the resulting OPP is *isomorphic* or *non-isomorphic*. A non-isomorphic OPP reflects a conflict, which can be interpreted as

a violation of the graph's edge relation. Such an OPP allows early elimination of the subtree rooted at the current tree node, since that subtree does not contain valid permutations (or in fact, a symmetry of the graph).

To illustrate, consider the 7-vertex graph in Figure 2 and assume that the decision to map vertex 0 to vertex 4 has just been made. This decision triggers partition refinement which causes the top and bottom partitions of the OPP to refine non-isomorphically, proving that there are no automorphisms of this graph that map vertex 0 to vertex 4.

4.6.2 Matching OPP pruning

A key pruning mechanism enabled by the OPP encoding of permutation sets is the quick discovery of candidate coset representatives. This occurs when the OPP at a given tree node is *matching*. A matching OPP corresponds to a permutation that maps the vertices in matching cells identically.

Example IV.3. The matching OPP

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

encodes the permutation (1 3 5).

Let Π be a matching OPP, and let α be the permutation that maps the vertices in the non-singleton cells of Π identically. It can be shown that α always forms a symmetry of the graph (see Lemma IV.4 in Section 4.8.3). Hence, when **saucy** encounters OPP Π, it prunes the entire subtree rooted at Π, since symmetry α serves as a coset representative for the subtree rooted at Π.

Inputs: G
Outputs: S

1. Perform recursive subgroup decomposition.
   Assume that v_l is the target vertex at level l.
2. For each level l of the permutation tree:
3. For each candidate vertex u of target vertex v_l at level l:
4. Continue if v_l and u share the same orbit (orbit pruning).
5. Continue if we already explored the mapping of v_l to a vertex w, while w and v share the same orbit (orbit pruning).
6. Map v_l to u.
   Assume that $H_{v_l \mapsto u}$ contains the set of all permutations that map v_l to u.
7. Search $H_{v_l \mapsto u}$ for a coset representative.
   Within this search, apply the following three pruning mechanisms:
   − coset pruning,
   − matching OPP pruning, and
   − non-isomorphic OPP pruning.
8. If $H_{v_l \mapsto u}$ contains a coset representative α, add α to set of generators S.
9. return S.

Figure 4.9: The outline of the **saucy** symmetry-detection algorithm.

4.7 The Graph-Symmetry Discovery Algorithm

The **saucy** graph symmetry-detection algorithm is outlined in Figure 4.9. This algorithm performs a depth-first traversal of the permutation search tree. This is accomplished by first performing a phase of subgroup decomposition, and then searching the subspaces that map the target vertex at each level to the candidate vertices. Each subspace is pruned by systematic application of the following four pruning rules:

– **Coset pruning** which terminates the search in a coset subtree as soon as a coset representative is found.

– **Orbit pruning** which avoids searching the subtree of coset $H_{i \mapsto j}$ if j is already in the orbit of i.

– **Non-isomorphic OPP pruning** which indicates that there are no permutations in the subtree rooted at that node which are symmetries of the graph.

– **Matching OPP pruning** which can identify a candidate permutation at a tree node without the need to explore the subtree rooted at that node.

At the end, **saucy** returns the set of found coset representatives as the generating set for the symmetry group of the graph.

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

Finally, it is interesting to note that in addition to finding a set of generators for G, symmetry-detection algorithms can also compute the order of G using the orbit-stabilizer and Lagrange theorems [28]: $|\mathcal{G}| = |\mathcal{G}_i| \cdot |\Theta_i|$.

An example graph along with its search tree constructed by **saucy** are depicted in Figure 4.10. This search tree exhibits all four pruning mechanisms. Below is the trace of **saucy** algorithms for the example graph of Figure 4.10.

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

1. Fix vertex 3 and refine

2. Fix vertex 5

3. Fix vertex 6

4. Fix vertex 2; $\mathcal{G}_2 = \iota$

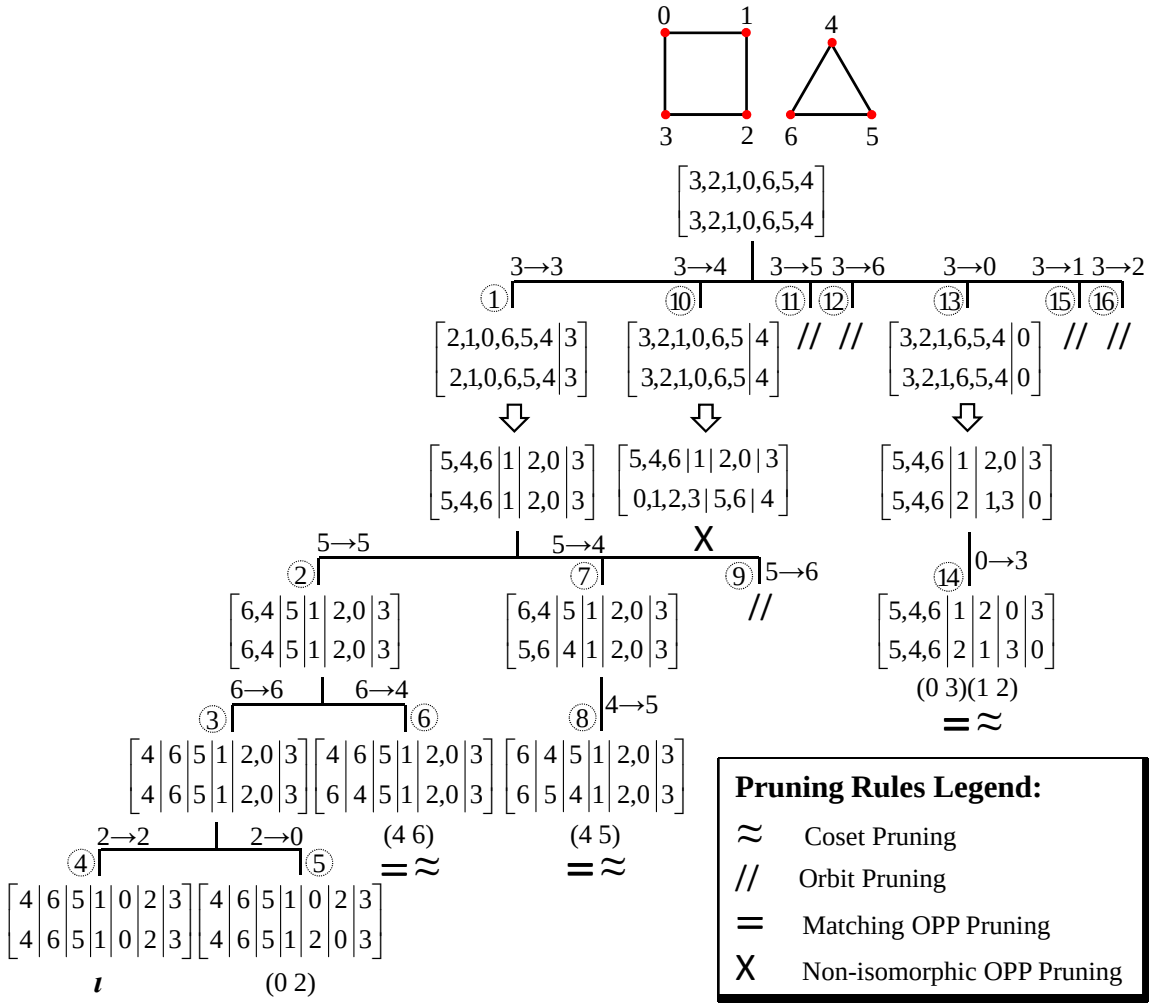


Figure 4.10: Search tree for graph automorphisms of the "square and triangle" graph and relevant computations at each node. The shaded region corresponds to subgroup decomposition.

5. Search for representative of coset $H_{2\mapsto 0}$;

   Found representative of coset $H_{2\mapsto 0}$;

   $Z = \{(0\ 2)\}$; $\Theta = \{0, 2 \mid 1 \mid 3 \mid 4 \mid 5 \mid 6\}$; $|\mathcal{G}_6| = |\mathcal{G}_2| \times |\Theta_2| = 1 \times 2 = 2$

6. Search for representative of coset $H_{6\mapsto 4}$;

   Matching OPP pruning: found representative of coset $H_{6\mapsto 4}$;

   Coset pruning: no need to explore since we have already found a coset representative for $H_{6\mapsto 4}$;

$Z = \{(0\ 2), (4\ 6)\}$; $\Theta = \{0, 2 \mid 1 \mid 3 \mid 4, 6 \mid 5\}$

7. Search for representative of coset $H_{5\mapsto 4}$

8. Map vertex 4 to vertex 5;

   Matching OPP pruning: found representative of coset $H_{5\mapsto 4}$;

   Coset pruning: no need to explore since we have already found a coset representative for $H_{5\mapsto 4}$

   $Z = \{(0\ 2), (4\ 6), (4\ 5)\}$; $\Theta = \{0, 2 \mid 1 \mid 3 \mid 4, 5, 6\}$

9. **Orbit pruning**: no need to explore since 6 is already in the orbit of 5;

   $|\mathcal{G}_5| = |\mathcal{G}_6| \times |\Theta_6| = 2 \times 3 = 6$

10. **Non-isomorphic OPP pruning**: 3 cannot map to 4

11. **Orbit pruning**: no need to explore since 5 is already in the orbit of 4

12. **Orbit pruning**: no need to explore since 6 is already in the orbit of 4

13. Search for representative of coset $H_{3\mapsto 0}$

14. Map vertex 0 to vertex 3;

   Matching OPP pruning: found representative of coset $H_{3\mapsto 0}$.

   Coset pruning: no need to explore since we have already found a coset representative for $H_{3\mapsto 0}$

   $Z = \{(0\ 2), (4\ 6), (4\ 5), (0\ 3)(1\ 2)\}$; $\Theta = \{0, 1, 2, 3 \mid 4, 5, 6\}$;

   $|\mathcal{G}_3| = |\mathcal{G}_5| \times |\Theta_5| = 6 \times 2 = 12$

15. **Orbit pruning**: no need to explore since 1 is already in the orbit of 3

16. **Orbit pruning**: no need to explore since 2 is already in the orbit of 3

4.8 Conflict Anticipation via Simultaneous Refinement

The early versions of **saucy** (i.e., **saucy** 2.1 and earlier) applied partition refinement *separately* to the top and bottom partitions of an OPP. In implementation, they first refined the top partition until it became equitable, and recorded where the cell splits occurred. Then, they refined the bottom partition, and compared the splitting locations of the bottom to the top (i.e., checked the isomorphism of the two partitions). We refer to this refinement scheme as *conventional* refinement.

In this section, we argue that the conventional partition refinement (as explained above) might overlook certain conflicts. In particular, we demonstrate cases where an OPP is found isomorphic by conventional refinement, but 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 a *simultaneous partition refinement* procedure that detects such cases and does not explore them. We discuss the impact of our refinement procedure on the search tree constructed for our example.

4.8.1 Simultaneous vs. Conventional Refinement

Consider the 20-vertex 46-edge graph shown in Figure 4.11. The search tree generated by **saucy** 2.1 for this graph is shown in Figure 4.12. This search tree produces 16 conflicts, indicated by red-shaded nodes. In the remainder, we focus on the path from the root that maps $11 \mapsto 0$ and then $14 \mapsto 4$. The OPPs in Figure 4.13, labeled with (4.2), (4.3) and (4.4), 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 (4.4), obtained after mapping $14 \mapsto 4$, is not considered to be a conflict 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 4.11. To see this, consider the edge that connects 13 to 16. This

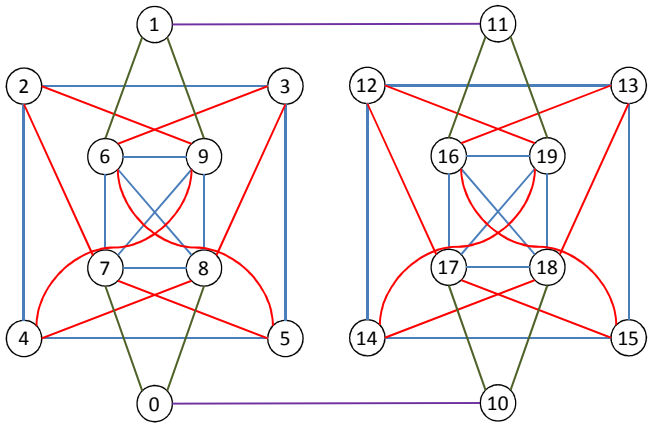


Figure 4.11: A 20-vertex 46-edge graph with symmetry group of size 32.

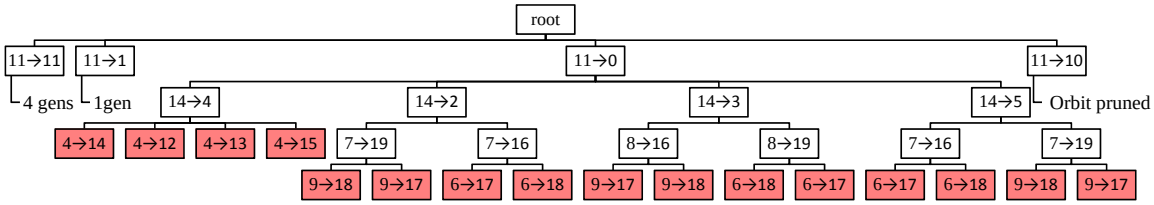


Figure 4.12: The search tree constructed by **saucy** 2.1 for the graph in Figure 4.11.

edge, according to OPP (4.4), should be mapped to another edge that connects 3 to 7, since OPP (4.4) maps $13 \mapsto 3$, and $16 \mapsto 7$. Nevertheless, no such edge exists between 3 and 7 in Figure 4.11, and hence, OPP (4.4) is a conflict.

The question now is why the refinement procedure failed to detect the above conflict? Or, in other words, why was OPP (4.4) found to be isomorphic? To answer this question, we should follow the trace of the refinement procedure which is performed on OPP (4.3) to get OPP (4.4) after mapping $14 \mapsto 4$. As elaborated earlier, **saucy** 2.1 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 4.14 and Figure 4.15, respectively.

The refinement on the top starts by first making 14 a singleton cell (partition (4.5)). According to the graph of Figure 4.11, 14 is connected to 12,15,18 and 19, but

$$\left[\begin{array}{c|c|c} 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \\ 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \end{array}\right] \tag{4.2}$$

$$\left[\begin{array}{c|c|c|c|c|c|c|c|c} 0 & 10 & 1 & 11 & 14,12,13,15 & 2,4,5,3 & 17,18 & 8,7 & 6,9 & 16,19 \\ 11 & 1 & 10 & 0 & 4,3,5,2 & 13,14,12,15 & 9,6 & 19,16 & 18,17 & 7,8 \end{array}\right] \tag{4.3}$$

$$\left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c} 0 & 10 & 1 & 11 & 13 & 12 & 15 & 14 & 2,4,5,3 & 17 & 18 & 8,7 & 6,9 & 16 & 19 \\ 11 & 1 & 10 & 0 & 3 & 2 & 5 & 4 & 13,14,12,15 & 6 & 9 & 19,16 & 18,17 & 7 & 8 \end{array}\right] \tag{4.4}$$

Figure 4.13: The search nodes of the tree in Figure 4.12. OPP (4.2) is at the root, OPP (4.3) is after mapping $11 \mapsto 0$, and OPP (4.4) is after mapping $14 \mapsto 4$.

$$\left[0 \mid 10 \mid 1 \mid 11 \mid 12,13,15 \mid 14 \mid 2,4,5,3 \mid 17,18 \mid 8,7 \mid 6,9 \mid 16,19\right] \tag{4.5}$$

$$\left[0 \mid 10 \mid 1 \mid 11 \mid 13 \mid 12,15 \mid 14 \mid 2,4,5,3 \mid 18 \mid 17 \mid 8,7 \mid 6,9 \mid 16 \mid 19\right] \tag{4.6}$$

$$\left[0 \mid 10 \mid 1 \mid 11 \mid 13 \mid 12 \mid 15 \mid 14 \mid 2,4,5,3 \mid 18 \mid 17 \mid 8,7 \mid 6,9 \mid 16 \mid 19\right] \tag{4.7}$$

Figure 4.14: The refinement of the top partition of OPP (4.3) to get OPP (4.4).

$$\left[11 \mid 1 \mid 10 \mid 0 \mid 3,5,2 \mid 4 \mid 13,14,12,15 \mid 9,6 \mid 19,16 \mid 18,17 \mid 7,8\right] \tag{4.8}$$

$$\left[11 \mid 1 \mid 10 \mid 0 \mid 3 \mid 5,2 \mid 4 \mid 13,14,12,15 \mid 9 \mid 6 \mid 19,16 \mid 18,17 \mid 7 \mid 8\right] \tag{4.9}$$

$$\left[11 \mid 1 \mid 10 \mid 0 \mid 3 \mid 2 \mid 5 \mid 4 \mid 13,14,12,15 \mid 9 \mid 6 \mid 19,16 \mid 18,17 \mid 7 \mid 8\right] \tag{4.10}$$

Figure 4.15: The refinement of the bottom partition of OPP (4.3) to get OPP (4.4).

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 (4.6)). The refinement continues by looking at the connections of one of the newly created cells. Here, **saucy** 2.1 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 (4.7)). The top partition is now equitable, i.e., no further refinement is implied.

After refining the top partition, **saucy** 2.1 starts refining the bottom partition.

This is done by first making 4 a singleton cell (partition (4.8)). Since 4 is connected to 2,5,8 and 9, refinement separates 2 and 5 from 3 (this makes 3 a singleton cell), 9 from 6, and 8 from 7 (partition (4.9)). Note that, at this point, partition (4.9) is isomorphic to partition (4.6), i.e., no conflict is detected. This time **saucy** 2.1 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** 2.1 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** 2.1 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, **saucy**'s refinement procedure refines one partition at a time, and checks isomorphism once both partitions are equitable. After refining based on 7, **saucy** 2.1 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 4.10). The bottom partition is now equitable and isomorphic to the top.

After the refinement procedure ends, **saucy** 2.1 builds isomorphic OPP (4.4), 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 (4.4) 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.

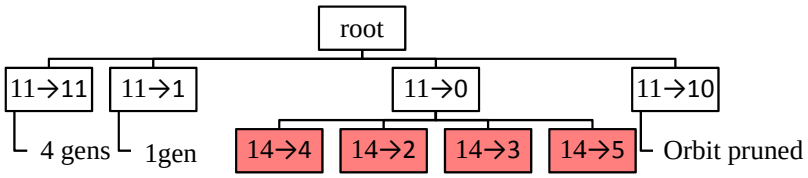


Figure 4.16: The search tree constructed by **saucy** 3.0 for the graph in Figure 4.11.

4.8.2 Simultaneous Partition 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 4.16 shows the search tree for the graph in Figure 4.11 when our new simultaneous refinement is invoked. Comparing this search tree to that in Figure 4.12, the number of conflicts is reduced from 16 to 4.

4.8.3 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 prunes the entire subtree rooted at this OPP by assuming that α is a symmetry of the graph, and hence can be returned as a coset representative. In this subsection, we prove

that such an assumption is correct when Π is found matching by our simultaneous partition refinement.

Lemma IV.4. *If* **saucy***'s simultaneous partition refinement finds OPP* Π *matching, permutation* α *that corresponds to* Π *is always an automorphism of graph* G*.*

Proof. When OPP Π is found matching by **saucy**'s simultaneous refinement, Π is equitable, isomorphic, matching, and conforming according to G's edge relation. Furthermore, permutation α that corresponds to Π 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.

4.9 The Complexity of Our Algorithm for Trees

In this section, we analyze the runtime of **saucy** for trees. To be more specific, we show that **saucy** takes linear time to find the orbits of the automorphism group of a given tree. We then show that finding the orbits of a tree is linear-time equivalent

to the tree isomorphism problem, concluding that **saucy** solves tree isomorphism in linear time.

The first linear-time algorithm to solve the tree isomorphism problem was introduced by Aho, Hopcroft and Ullman [3]. This algorithm is referred to as the *AHU (graph isomorphism) algorithm*. The AHU algorithm, in fact, solves the isomorphism problem for *rooted trees* but can also be easily adapted to unrooted trees.

To prove that **saucy** takes linear time to solve the tree isomorphism problem, we first discuss the difference in checking isomorphism of rooted trees vs. unrooted trees. We then explain the AHU tree isomorphism algorithm, and show that **saucy** essentially performs the same algorithm as AHU for checking the isomorphism of rooted trees.

4.9.1 Isomorphism of Rooted Trees vs. Unrooted Trees

Any tree isomorphism algorithm for rooted trees should map roots to roots.

Example IV.5. The two trees in Figure 4.17 are isomorphic as unrooted trees but not as rooted trees.

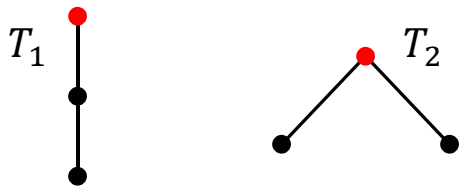


Figure 4.17: T_1 and T_2 are two trees whose roots are colored as red. These trees are isomorphic as unrooted trees but not as rooted trees.

Lemma IV.6. *If there exists a linear-time isomorphism algorithm for rooted trees, there exists a linear-time isomorphism algorithm for unrooted trees.*

Proof. Assume that $ISO\_ROOTED(T_1, r_1, T_2, r_2)$ is a linear-time procedure that verifies the isomorphism of two trees T_1 and T_2 with roots r_1 and r_2, respectively. In

fact, this procedure returns true if rooted tree T_1 is isomorphic to rooted tree T_2, and false otherwise. We describe procedure $ISO(T_1, T_2)$ which verifies the isomorphism of two unrooted trees T_1 and T_2 as follows. We find the centers of T_1 and T_2, and based on that, do one of the following:

1. if T_1 has one center c_1, and T_2 has one center c_2,
   return $ISO\_ROOTED(T_1, c_1, T_2, c_2)$

2. if T_1 has two centers c_1 and c_1', and T_2 has two centers c_2 and c_2',
   return $ISO\_ROOTED(T_1, c_1, T_2, c_2)$ or $ISO\_ROOTED(T_1, c_1', T_2, c_2)$

3. If T_1 and T_2 have different number of centers,
   return false

The procedure $ISO(T_1, T_2)$ takes linear time if finding tree centers takes linear time. The following procedure finds the centers of a tree in linear time:

1. Choose a random root r.

2. Find a vertex v_1 — the farthest from r.

3. Find a vertex v_2 — the farthest from v_1.

4. Centers are the median elements of the path from v_1 to v_2.

4.9.2 The AHU Tree Isomorphism Algorithm

Given two rooted trees T_1 and T_2, the AHU algorithm determines the isomorphism of T_1 and T_2 as follows [3] (pages 84-86):

1. Assign to all leaves of T_1 and T_2 the integer 0.

2. Inductively, assume that all vertices of T_1 and T_2 at level $i+1$ have been assigned integers. Assume L_1 is a list of the vertices of T_1 at level $i+1$ sorted by non-

decreasing value of the assigned integers. Assume L_2 is the corresponding list for T_2.

3. Assign to the non-leaves of T_1 at level i a tuple of integers by scanning the list L_1 from left to right and performing the following actions: for each vertex on list L_1, take the integer assigned to v to be the next component of the tuple associated with the parent of v. On completion of this step, each non-leaf w of T_1 at level i will have a tuple $(i_1, i_2, ..., i_k)$ associated with it, where $i_1, i_2, ..., i_k$ are integers, in non-decreasing order, associated with the sons of w. Let S_1 be the sequence of tuples created for the vertices of T_1 on level i.

4. Repeat step 3 for T_2 and let S_2 be the sequence of tuples created for the vertices of T_2 on level i.

5. Sort S_1 and S_2 lexicographically. Let S_1' and S_2' respectively be the sorted sequences of tuples. Call these sorted tuples *AHU signatures*.

6. If S_1' and S_2' are not identical then halt; the trees are not isomorphic. Otherwise, assign the integer 1 to those vertices of T_1 on level i represented by the first distinct tuple on S_1', assign the integer 2 to the vertices represented by the second distinct tuple, and so on. As these integers are assigned to the vertices of T_1 on level i, make a list L_1 of the vertices so assigned. Append to the front of L_1 all leaves of T_1 on level i. Let L_2 be the corresponding list of vertices of T_2. These two lists can now be used for the assignment of tuples to vertices of level $i-1$ by returning to step 3.

7. If the roots of T_1 and T_2 are assigned the same integer, T_1 and T_2 are isomorphic.

The AHU tree-isomorphism algorithm runs in linear time if lexicographical sorting of tuples (line 5) takes linear time. A sorting algorithm with such a runtime is described in [3] (pages 80-84).

Example IV.7. Figure 4.18 illustrates the assignment of the AHU signatures to the vertices of two isomorphic trees.

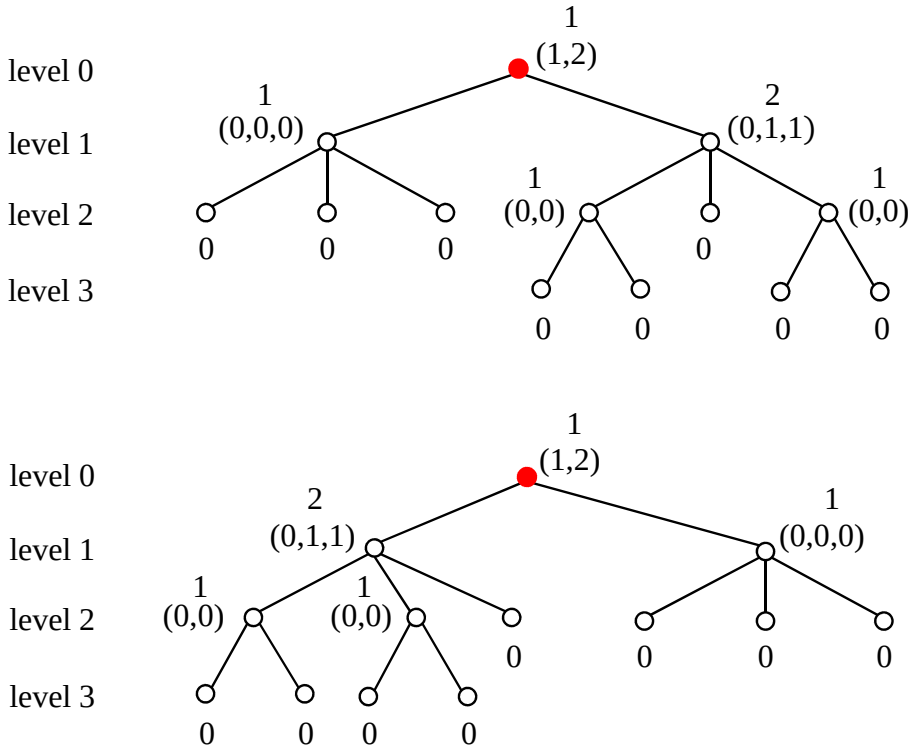


Figure 4.18: The assignment of the AHU signatures to the vertices of two isomorphic trees.

4.9.3 Tree Automorphisms

In this sub-section, we discuss automorphisms of rooted trees.

Lemma IV.8. *There exists a symmetry that maps node v to node w only if the subtree rooted at node v is isomorphic to the subtree rooted at node w.*

proof A symmetry that maps node v to node w should map the connections of v to the connections of w (in order to respect the edge relation of the tree). In other words, such a symmetry should map the children of v to the children of w, the children of the children of v to the children of the children of w, and so on until leaves are

reached. Such a level-by-level mapping is possible only if the subtree rooted at node v isomorphic to the subtree rooted at node w.

Lemma IV.9. *There exists a symmetry that maps node v to node w only if v and w have the same height.*

proof This is trivial according to Lemma IV.8; two trees are isomorphic only if they have the same height.

Definition IV.10. The *Lowest Common Ancestor (LCA)* of two nodes v and w is the deepest node that has both v and w as its descendants.

Example IV.11. In Figure 4.19, the LCA of node v and node w is node z.

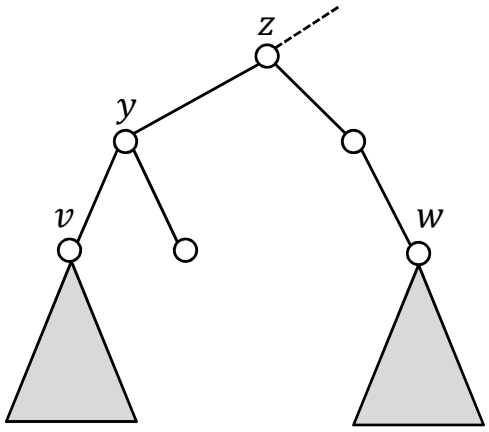


Figure 4.19: Example sub-tree rooted at node z.

Definition IV.12. Given two nodes v and w, the maximal subtree of v excluding w, denoted by MS_{v-w}, is the largest subtree that contains v but not w.

Example IV.13. In Figure 4.19, the maximal subtree of v excluding w is the subtree rooted at node y.

Lemma IV.14. *The maximal subtree of v excluding w is the subtree rooted at the child of the LCA of v and w that has v as its descendant.*

Proof. Let z denote the LCA of v and w. We know that z is the deepest node that has both v and w as its dependents. This means that all the nodes that are shallower than z which have v as a descendant, also have z, and subsequently w, as descendants. Thus, we conclude that the MS_{v-w} is rooted at a node deeper than z. Since z is the LCA of v and w, there should always exists two children of z that one has v and the other has w as a descendant. Suppose that C_v is the child of z that has v (but not w) as its descendant. Since C_v is only one level deeper than z, we conclude that C_v is the maximal subtree of v excluding w.

Lemma IV.15. *There exists a symmetry that maps node v to node w only if the maximal subtree of v excluding w is isomorphic to the maximal subtree of w excluding v.*

Proof. A symmetry that maps v to w is a permutation that maps v to w and keeps the edge relation of the tree. In the proof of Lemma IV.8, we only considered the edges that connect v and w to their children. Here, we focus on the connections of v and w to their parents. We assert that a symmetry that maps v to w should map the parents of v to the parents of w, the parents of parents of v to the parents of parents of w and so on. This mapping continues until the LCA of v and w is reached, since beyond that point, the ancestors of v and w are single nodes. Such a level-by-level mapping exists only if MS_{v-w} is isomorphic to MS_{w-v}.

Lemma IV.16. *Node v can be mapped to node w by a symmetry only if v and w have the same depth.*

Proof. According to Lemma IV.15, there exists a symmetry that maps v to w only if MS_{v-w} is isomorphic to MS_{w-v}. According to Lemma IV.9, two trees are isomorphic only if they have the same height. Hence, MS_{v-w} is isomorphic to MS_{w-v} only if they have the same height, say h. Likewise, the subtree rooted at v is isomorphic to the

subtree rooted at w only if they have the same height, say h'. Now, let $d = h - h' + 1$. Note that d corresponds to the distance of the path from v (or w) to the LCA of v and w. Also, let d' denote the depth of the LCA of v and w. It is easy to see that the depth of v (or w) is $d + d'$, i.e., v and w have the same depth.

Theorem IV.17. *There exists a symmetry that maps node v to node w if only if the subtree rooted at v is isomorphic to the subtree rooted at w, and the maximal subtree of v excluding w is isomorphic to the maximal subtree of w excluding v.*

Proof. The "only if" case: See the proof of Lemma IV.8 and Lemma IV.15.
The "if" case: The subtree rooted at v is isomorphic to the subtree rooted at w; so, the edge relation of the tree is maintained for all the nodes that are deeper than v and w who have v or w as their ancestor. Furthermore, the maximal subtree of v excluding w is isomorphic to the maximal subtree of w excluding v; so, the edge relation of the tree is maintained for all the nodes that are shallower than v and w and have the LCA of v and w as their ancestor. All the other nodes (that do not have v or w as their ancestor or successor) are not moved by a potential symmetry that maps v to w. Since the edge relation of the tree is maintained for all the nodes, we conclude that there exists a symmetry that maps v to w.

The following is a restatement of Theorem IV.17 based on the definition of signature in the AHU algorithm (see Section 4.9.2): there exists a symmetry that maps node v to node w, if v and w have the same AHU signature, and all the same-level ancestors of v and w (up to the LCA of v and w) have the same AHU signature as well.

4.9.4 Orbit Partition vs. Initial Equitable Partition

Here, we show that the initial equitable partition resulted from **saucy**'s partition refinement for a rooted tree corresponds to the orbit partition of the tree. Recall

that partition $\pi = [W_1|W_2|\cdots|W_t]$ is said to be equitable (with respect to a given graph) if, for all vertices $v_1, v_2 \in W_i$ $(1 \leq i \leq t)$, the number of neighbors of v_1 in W_j $(1 \leq j \leq t)$ is equal to the number of neighbors of v_2 in W_j. Also, recall that two vertices are in the same cell of the orbit partition, if there exists a symmetry that maps one vertex to the other.

Given a rooted tree, **saucy**'s partition refinement distinguishes the root, since it has a unique color. It also distinguishes all the leaves, since they all have degree 1. Then, it propagates the edge relation of the tree until an equitable partition is obtained.

Lemma IV.18. *Two nodes v and w are in the same cell of equitable partition π, only if they have the same height and the same depth.*

Proof. The initial refinement distinguishes the root and the leaves. Then, it propagates the edge relation of the tree from the root down to the leaves, and puts two vertices in one cell only if they have the same depth. Next, it propagates the edge relation from the leaves up to the root, and puts two vertices in one cell only if they have the same height. Therefore, two nodes are in the same cell of the equitable partition, only if they have the same height and the same depth.

Lemma IV.19. *Two nodes v and w are in the same cell of equitable partition π, only if they have the same AHU signature.*

Proof. Assume that v and w are in the same cell of π, but have different AHU signatures. We know that:

1. π is equitable; so, v and w have the same number of connections to all cells of π, and

2. the AHU algorithm computes the signatures of v and w from the signatures of the children of v and w.

According to (1) and (2), v and w can have different signatures only if (at least) one child of v and one child of w are in the same cell of π but have different signatures. If we recursively continue this argument, we finally reach the leaves (note that v and w have the same height according to Lemma IV.18), where we assert that there exists two leaves, one as a descendant of v and the other as a descendant of w, that are in the same cell but have different signatures. According to the AHU algorithm, however, all the leaves are assigned the signature 0, and this is a contradiction with our assertion. So, we conclude that our initial assumption was wrong, i.e., v and w are in the same cell of π, only if they have the same signature.

Lemma IV.20. *Two nodes v and w are in the same cell of equitable partition π, only if all the same-level ancestors of v and w (up to the LCA of v and w) have the same AHU signature.*

Proof. Let p and q denote two same-level ancestors of v and w, respectively, where $p \neq q$, and p and q have different signatures. According to Lemma IV.19, p and q cannot be in the same cell of π, since they do not have the same signature. If we now propagate the edge relation of the tree from p and q downward to the leaves, we distinguish those descendants of p from those descendants of q that are at the same level, including v and w. This suggests that if p and q do not have the same signature, then v and w cannot be in the same cell.

Theorem IV.21. *Two nodes v and w are in the same cell of equitable partition π, if and only if they have the same AHU signature, and all the same-level ancestors of v and w (up to the LCA of v and w) have the same AHU signature as well.*

Proof. The "only if" case: See Lemma IV.19 and Lemma IV.20.
The "if" case: Assume that v and w are not in the same cell of π, but have the same AHU signature, and all the same-level ancestors of v and w have the same AHU

signature as well. There are two remarks:

1. since v and w are not in the same cell, there does not exists any symmetry that maps v to w, and

2. according to Theorem IV.17, there exists a symmetry that maps v to w, since v and w have the same signature, and all the same-level ancestors of v and w (up to the LCA of v and w) have the same signature as well.

It is easy to see that (1) and (2) contradict each other, i.e., our initial assumption was wrong. In other words, v and w are in the same cell of π, if they have the same AHU signature, and all the same-level ancestors of v and w have the same AHU signature as well.

Theorem IV.22. *The initial equitable partition of a rooted tree corresponds to the orbit partition of the tree's automorphism group.*

Proof. This can be deducted trivially from Theorem IV.17 and Theorem IV.21.

4.9.5 Calculating the Orbit Partition

Here, we show that **saucy** takes linear time to compute initial equitable partition of a given rooted tree. We show this by arguing that **saucy**'s partition refinement is essentially performing the same algorithm as the AHU tree isomorphism.

Given a rooted tree T and the unit partition π of its vertices, **saucy**'s partition refinement executes the following steps:

1. Distinguish leaves from other vertices in π by putting them in their own cell (the AHU algorithm distinguishes leaves by assigning them the signature 0).

2. Inductively, assume that $C_1, C_2, ..., C_n$ are the newly created cells at level $i+1$, and all the vertices in cell C_k $(1 \leq k \leq n)$ have the same AHU signature, say k.

3. Refine the cells of π that contain the vertices at level i by scanning $C_1, C_2, ..., C_n$ from left to right and performing the following actions: For each vertex in C_i, look at the connections of that vertex to the vertices at level i, bucket sort the vertices at level i based on their number of connections to the vertices in C_i, and refine π based on the computed buckets, i.e., put two vertices in the same cell of π only if a) they are currently in the same cell of π, and b) they belong to the same bucket. On completion of this step, two vertices at level i are in the same cell if and only if they have the same number of connections to each C_k $(1 \leq k \leq n)$, i.e., they have the same tuple of the form $(i_1, i_2, ..., i_n)$ where i_k denotes the number of connections to cell C_k. Let's call these tuples *refinement tuples*.

4. Refinement tuples have a one-to-one correspondence with the AHU signatures as indicated by the following procedure: Replace each i_k in a refinement tuple with i_k copies of integer k; the resulting tuple is the corresponding AHU signature. This procedure can be done in reverse to obtain refinement tuples from AHU signature. On the completion of step 3, we conclude that two vertices at level i are in the same cell of π if only if they have the same AHU signatures.

5. Repeat step 3 until the root is reached. At this point, two vertices are in the same cell of π if only if they have the same AHU signature.

6. We now perform another round of partition refinement, but this time, we propagate the edge relation from the root to the leaves. Inductively, assume that $C_1, C_2, ..., C_n$ are the cells that contain the vertices at level $i - 1$.

7. Refine the cells of π that contain the vertices at level i by scanning $C_1, C_2, ..., C_n$ from left to right and performing the exact same actions as discussed in step 3. On the completion of this step, two vertices at level i are in the same cell of π if and only if their same-level ancestors are in the same cell of π.

8. Repeat step 7 until the leaves are reached. At this point, two vertices are in the same cell of π if and only if they have the same AHU signature, and their same-level ancestors have the same AHU signature as well. According to Theorem IV.21, π is now equitable.

The above algorithm takes linear time, since it scans the tree once from the leaves to the root and once from the root to the leaves, and bucket sort takes linear time.

4.9.6 Checking Tree Isomorphism

In Section 4.9.5, we showed that **saucy** finds the orbit partition of the automorphism group of a rooted tree in linear time. Here, we show that verifying the isomorphism is linear-time equivalent to finding the orbit partition (the other way around requires quadratic time), concluding that **saucy** solves tree isomorphism in linear time.

The following procedure checks the isomorphism of two rooted trees T_1 and T_2 with roots r_1 and r_2, respectively, using a procedure that finds the orbit partition:

1. Build rooted tree T by (a) putting T_1 and T_2 side by side, (b) adding a vertex r with unique color to T, and (c) connecting vertex r to the roots of T_1 and T_2 (i.e., r_1 and r_2).

2. Find the orbit partition π of the automorphism group of tree T.

3. Return true if r_1 and r_2 are in the same cell of partition π.

4. Return false.

The above procedure uses the orbit partition to check whether there exists a symmetry that maps tree T_1 to tree T_2 (line 3). It concludes that T_1 and T_2 are isomorphic if such a symmetry exists, and non-isomorphic otherwise. Since **saucy** takes linear time to find the orbit partition of a rooted tree, it takes linear time to solve tree isomorphism as well.

CHAPTER V

Two-Pass Graph Canonical-Labeling Algorithms

In this chapter, we contrast the graph symmetry-detection and canonical-labeling problems, and dissect typical algorithms to identify their similarities and differences. In particular, we compare the algorithms in **saucy** (for graph symmetry detection) with those in **bliss** (for graph canonical labeling). We then develop a novel approach to canonical labeling where symmetries are found first and then used to speed up the canonical-labeling algorithms. To justify the effectiveness of our approach, we analyze the runtime complexity of **saucy** and **bliss** in detecting symmetries of an example graph. Throughout this section, we assume that the input to **saucy** and **bliss** is an n-vertex undirected colored graph G with vertex set $V = \{0, 1, ..., n-1\}$.

5.1 Symmetry Finding vs. Canonical Labeling

In this section, we highlight the similarities and differences between the search for symmetries and a canonical labeling by focusing on the algorithms implemented in **saucy** 3.0 and **bliss** 0.72. While we chose **bliss** as a reference, our comparison can be extended to other **nauty**-based canonical-labeling tools. In the following subsections, we distinguish the search nodes of the trees constructed by **saucy** and **bliss**, explain what they represent, and show that the search trees used by these tools are fundamentally different. Furthermore, we discuss and compare the pruning techniques and

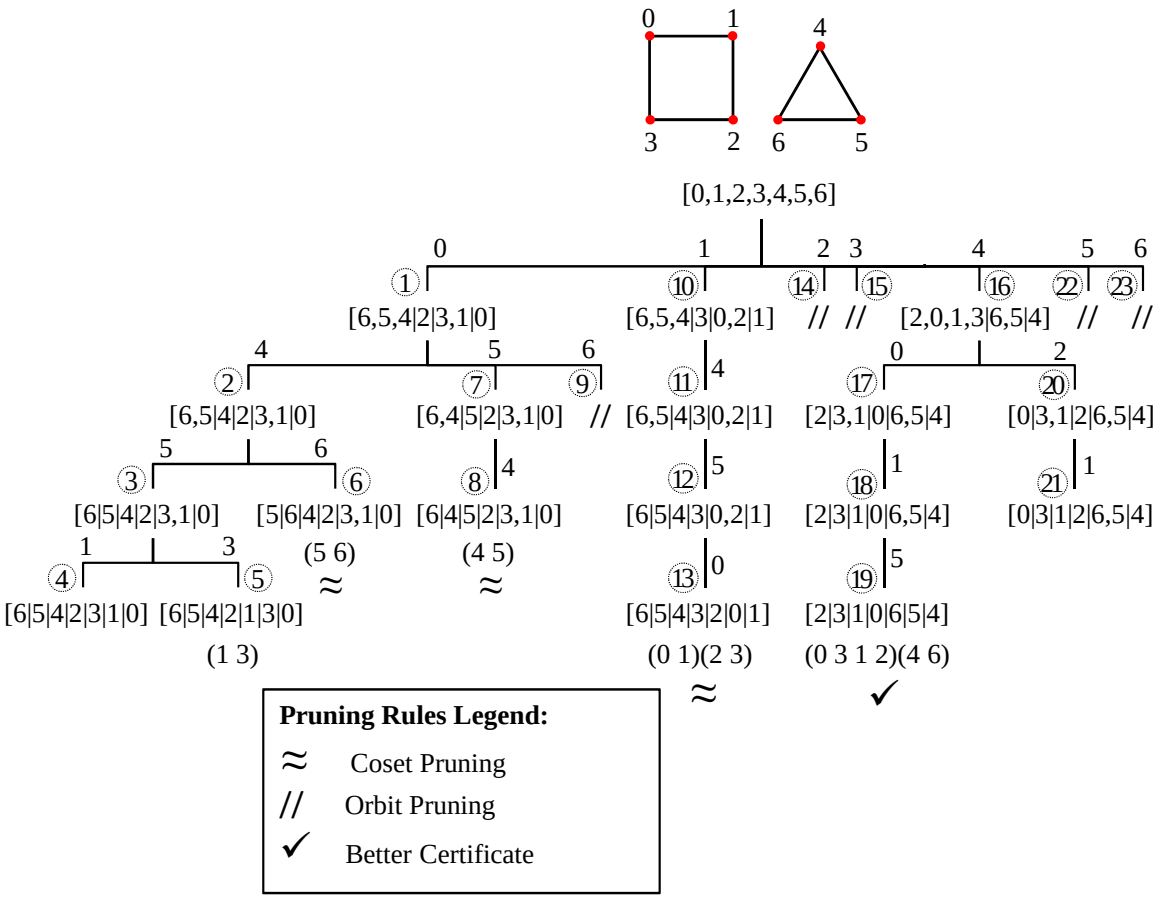


Figure 5.1: **bliss** canonical-labeling tree for the example graph of Figure 4.10.

branching mechanisms in **saucy** and **bliss**. We also point out an intrinsic limitation of the branching procedure in **bliss**, and show that this limitation does not apply to **saucy** search for automorphisms. To better understand and compare **saucy** and **bliss** baseline algorithms, Figure 5.1 depicts the **bliss** tree for the example graph of Figure 4.10. Throughout this section, we refer to the trees of Figure 4.10 and Figure 5.1 to illustrate **saucy** and **bliss** algorithms.

5.1.1 Search Trees

The nodes of the search tree in **saucy** are ordered partition pairs, each encoding 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). We discussed in Section 4.2 that, in general, an isomorphic OPP represents a non-empty sets of permutations. For example, the root of the tree in Figure 4.10 is a unit OPP encoding all $7! = 5040$ permutations on 7 vertices, and the OPP at node (7) is an isomorphic OPP representing the 4-element permutation set $\{(4\ 5), (4\ 5)(0\ 2), (4\ 6\ 5), (4\ 6\ 5)(0\ 2)\}$.

In contrast, the nodes of the search tree in **bliss** are single ordered partitions, each representing a (partial) labeling. A labeling in **bliss** is obtained by renaming each vertex with the position of that vertex in the ordered partition. The ordering of vertices in the partition suggests a permutation that, when applied to the graph, produces the labeling encoded by that partition. For example, at node (19) of Figure 5.1, vertices 0,1,2,3,4,5,6 are at indices 3,2,0,1,6,5,4, respectively, and hence, node (19) represents the labeling obtained by the permutation $(0\ 3\ 1\ 2)(4\ 6)$.

To compare labelings, each node in **bliss** is associated with a *certificate*. A certificate is a function that assigns a certain value to an ordered partition according to the graph's connection. Node certificates are computed as follows. Given an equitable partition (which is returned by partition refinement), **bliss** first makes a list of edges that connect singleton cells to other cells of the partition. For example, singleton cells $\{2\}$ and $\{0\}$ of the partition at node (1) of Figure 5.1 are connected to cell $\{1, 3\}$, and hence, the list of edges associated with node (1) is $\{\{2, 1\}, \{2, 3\}, \{0, 1\}, \{0, 3\}\}$. Then, **bliss** generates the certificate by renaming each vertex in the list of edges with the index of that vertex in the partition. In our example, vertices 0,1,2,3 are at indices 6,5,3,4 of the partition at node (1), respectively, and hence, the certificate of node (1) is $\{\{3, 5\}, \{3, 4\}, \{6, 5\}, \{6, 4\}\}$. Two ordered partitions produce the same certificate if they are isomorphic to each other. For example, nodes (1) and (10) of Figure 5.1 have the same certificate, but the certificate of node (1) is different from that of node (16).

To discard numerous impossible permutations and invalid labelings, **saucy** and **bliss** invoke partition refinement. Nevertheless, partition refinements in **bliss** is applied to one partition at a time, while **saucy**'s refinement benefits from *simultaneous* comparison of the top and bottom partitions, a concept which is unique to the OPP representation of permutations. Simultaneous refinement allows **saucy** to anticipate and avoid certain conflicts, which can lead to an exponential speed-up in run time [38].

5.1.2 Search Algorithms

In **saucy**, the search for symmetries is performed by constructing a permutation tree, and traversing it in a depth-first manner. The depth-first traversal is accomplished by mapping target vertices to candidate vertices. For example, the target vertex at level 2 (nodes (2), (7), and (9)) of Figure 4.10 is vertex 5, which is mapped to vertices 5, 4, and 6. Partition refinement is invoked after each mapping to prune away invalid permutations. The mapping procedure continues until the OPP becomes discrete, matching, or non-isomorphic (e.g., nodes (5), (6), and (10) of Figure 4.10, respectively). A discrete or matching OPP represents a symmetry, while a non-isomorphic OPP indicates a conflict. The search ends when all possible mappings are exhausted.

The root of the canonical-labeling tree in **bliss** is a unit ordered partition which is initially refined. The depth-first traversal of permutation space starts by choosing a non-singleton cell, and individualizing all the vertices in that cell one at a time. For example, at level 2 (nodes (2), (7) and (9)) of Figure 5.1, all the vertices in the first non-singleton cell of the partition at node (1), i.e., vertices 4, 5, and 6, are individualized one after the other. Each vertex individualization is followed by partition refinement to reflect the consequences of the branching decision.

Individualization in **bliss** continues until the partition becomes discrete, i.e., the

first leaf node is reached (node (4) of Figure 5.1). This leaf node is saved as a reference to compare certificates. A symmetry is found if another node during the search produces the same certificate as the first leaf node[1] (e.g., node (13) of Figure 5.1). The symmetry associated with such a node is the permutation that maps the partition at that node to the partition at the first leaf node. For example, the partition at node (13) of Figure 5.1 encodes symmetry (0 1)(2 3), since it can be obtained from the partition at node (4) by swapping vertex 0 with vertex 1 and vertex 2 with vertex 3. Furthermore, the canonical certificate is initialized to the certificate of the first leaf node, and is updated whenever a better certificate (based on any well-defined criterion, such as lexicographic ordering) is found during the search (e.g., node (19) of Figure 5.1). The canonical labeling of the graph is returned as the labeling of the node with the best certificate.

5.1.3 Pruning Techniques

Similar to **saucy**, the search algorithms in **bliss** exploit two group-theoretical pruning mechanisms: *coset pruning* and *orbit pruning*. These two pruning mechanisms follow similar routines in both **saucy** and **bliss**.

Coset pruning is based on the concept of coset representatives, i.e., one generator per coset is sufficient to generate all symmetries in the coset. For example, the symmetries found at node (8) of Figure 4.10 and (13) of Figure 5.1 are coset representatives of their corresponding subtrees rooted at node (7) and (10), and hence, those subtrees are coset pruned.

Orbit pruning relies on orbit partition to eliminate redundant generators. For instance, node (9) of Figure 4.10 is orbit pruned since vertices 5 and 6 share the same orbit. Similarly, node (9) of Figure 5.1 is orbit pruned since vertices 4 and 6 share the same orbit. The algorithms for coset and orbit pruning follow similar

[1]Obtaining symmetries at non-leaf nodes will be discussed in Section 5.1.3.

implementations in **saucy** and **bliss**.

To enable coset and orbit pruning, the left-most path of both **saucy** and **bliss** search tree corresponds to a sequence of subgroup stabilizers. In **saucy**, stabilizers are maintained by mapping each vertex to itself (fixing each vertex) until the identity is reached. For example, the tree of Figure 4.10 fixes vertices 3, 5, 6 and 2 to reach the identity at node (4). In **bliss**, subgroup decomposition individualizes vertices one at a time until the partition is discrete. In the tree of Figure 5.1, stabilizer subgroups of 0, 4, 5 and 1 result in a discrete partition at node (4).

In addition to the above group-theoretic pruning techniques, the data structures in **saucy** and **bliss** allow additional pruning mechanisms. One such pruning technique in **saucy** is non-isomorphic OPP pruning (see Section 4.6.1); For instance, the OPP at node (10) of Figure 4.10 is non-isomorphic, which indicates that the mapping of 3 to 4 is a conflict. Similarly, **bliss** identifies futile branches of the search by comparing the certificates of search nodes. Specifically, **bliss** prunes a subtree if the certificate of the root of the subtree 1) does not match the certificate of the node on the left-most path of the tree at that level (i.e., the subtree does not yield any symmetry), and 2) is not better than the current best certificate (i.e, the subtree does not include the canonical labeling). For example, node (16) of Figure 5.1 produces a different certificate than node (1), but the partial certificate associated with node (16) is better than the current best certificate (i.e, the certificate of node (4)), and hence, the subtree rooted at node (16) is explored.

Another OPP-based pruning technique in **saucy** is matching OPP pruning (see Section 4.6.2). Recall that a matching OPP is a non-discrete OPP in which corresponding non-singleton cells contain the same elements. The significance of this OPP is that it represents an early automorphism constructed by mapping the vertices of non-singleton cells identically. This automorphism can be returned as the coset representative of the current subtree, which exempts the search from exploring

the remaining permutations in that subtree. For example, the OPPs at nodes (6), (8) and (14) of Figure 4.10 are found matching, and are returned as the coset representatives of the subtrees rooted at nodes (6), (7) and (13), respectively. Until recently, no pruning mechanism in **bliss** had the same effect as the matching OPP pruning. In fact, all symmetries in **bliss** were found at leaf nodes. However, recent advances in **bliss** algorithms (version 0.72) exploit *component recursion* to enable early detection of symmetries without reaching the leaves. This is accomplished by comparing the ordered partitions at each level to the left-most ordered partition at the same level. For example, partitions (6) and (3) of Figure 5.1 both contain an identical non-singleton cell $\{3, 1\}$. This suggests that node (6) represents the symmetry (5 6), since partition (6) can be obtained from partition (3) by swapping vertex 5 with vertex 6. Although matching OPP and component recursion both aim to find symmetries early up in the tree, they are conceptually two distinct mechanisms, and impact the search trees in different ways.

The **bliss** algorithms use additional heuristics to facilitate the search for a canonical labeling. For example, **bliss** stores recently discovered symmetries to (partially) detect and prune fruitless symmetric branches of the search. It also uses a methodology to propagate conflicts beyond the most recent branching points, which helps it expedite automorphism search by pruning away subtrees that yield the same conflict. These two pruning techniques are not implemented in **saucy** 3.0, but our on-going research is investigating their possible incorporation.

5.1.4 Branching Decisions

Branching heuristics highly affect the performance of combinatorial search algorithms, including symmetry detection and canonical labeling. In **saucy**, branching is performed by choosing a target cell and a target vertex from the top partition. On the left-most tree path, **saucy** chooses the first non-singleton cell as the target cell,

and the first vertex in that cell as the target vertex (see nodes (1) to (4) of Figure 4.10). In the remaining 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. Note that 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. For example, node (13) of Figure 4.10 maps 3 to 0, and right after, node (14) maps 0 to 3. This consequently results in a matching OPP at node (14), representing the symmetry (0 3)(1 2). In practice, this heuristic is most effective when symmetry generators are sparse.

The branching procedure in **bliss** consists of a *cell-selector* function. Given graph G and partition π, cell-selector function $S(G, \pi)$ returns a non-singleton cell of π such that $S(G, \pi)^\gamma = S(G^\gamma, \pi^\gamma)$ for all $\gamma \in \mathcal{G}$ ($\mathcal{G}$ denotes the symmetry group of graph G). The cell selector's latter condition ensures that the search trees constructed for isomorphic graphs are also isomorphic. In implementation, **bliss** picks the same sequence of cells in all the paths from the root to the leaves. For example, the search tree of Figure 5.1 always individualizes the vertices in the first non-singleton cell of the partition. The default branching heuristic in **bliss** selects the *maximum nonuniformly joined* cell, i.e., the first non-singleton cell which is nonuniformly joined to the maximum number of cells (two cells are nonuniformly joined if the vertices in one cell have both neighbors and non-neighbors in the other cell). In the search tree of Figure 5.1, maximum nonuniformly joined cells happen to be the first non-singleton cells of the partition.

Considering the structures of the search trees in **saucy** and **bliss**, **saucy**'s branching procedure does not have the limitations of **bliss**'s cell-selector function, since it can choose any target cell and target vertex at each step of the search. This con-

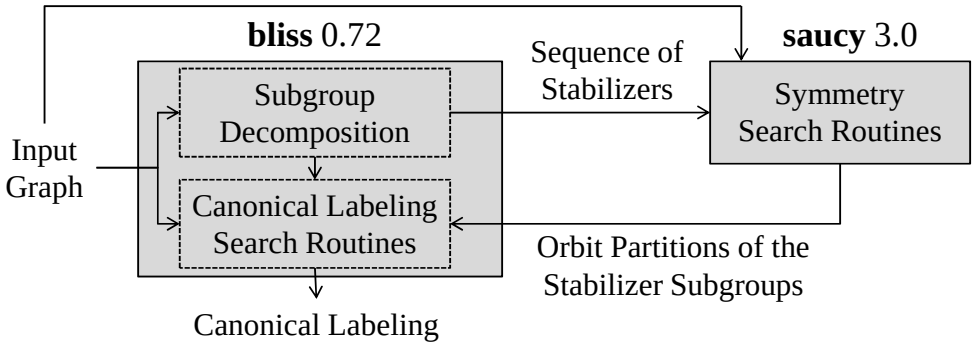


Figure 5.2: Our proposed canonical-labeling framework.

sideration raises the possibility of improving the branching heuristic in **saucy**. As mentioned, the default vertex-swap heuristic is effective when the input graph produces sparse generators. Our experimental results show that this is usually the case when the input graph is large and sparse. For other graphs, however, the vertex-swap heuristic does not necessarily produce the best results. We plan to explore other branching heuristics, and desirably, seek a methodology that adapts the branching heuristic to the characteristics of the input graph in our future research.

5.2 New Canonical-Labeling Procedure

In previous sections, we pointed out that **saucy** algorithms and data structures were optimized to solve the graph automorphism problem, whereas, **bliss** routines are mainly focused on finding a canonical representation. In this section, we propose a novel approach that takes advantage of **saucy**'s efficiency in finding graph symmetries to speed up the search for a canonical labeling. We show that once the symmetries are found, canonical labeling can be performed much faster using this information by pruning the canonical-labeling tree.

Our proposed graph canonicalization flow is depicted in Figure 5.2. It starts by launching **bliss** to perform subgroup decomposition. Once decomposition is complete, it temporarily interrupts the search, passes the sequence of stabilizers obtained from

subgroup decomposition to **saucy**, and waits for **saucy** to compute and pass back the graph's symmetry information. At the other end, **saucy**'s decomposition routines use **bliss**'s sequence of stabilizers to generate the subgroups. In other words, the left-most path of the tree in **saucy** is forced to match the one in **bliss**. As **saucy** looks for symmetries, it records the orbit partition at each level (i.e., the orbit partitions of the stabilizer subgroups). At the end of the search, it hands the computed orbit partitions over to **bliss**. The canonical-labeling algorithms in **bliss** then resume the search, but incorporate two major modifications: 1) the level-by-level orbit partitions computed by **saucy** are used to prune isomorphic subtrees, and 2) the search for symmetries is disabled in all expanded subtrees. Another way to say this is that a subtree that contains a symmetry will produce labelings that were previously examined, and hence, can be entirely pruned. On the other hand, a subtree that does not include any symmetry might lead to a better labeling (possibly, the canonical labeling), and hence, should be explored.

As elaborated above, our graph canonicalization approach divides the search into two phases: the search for symmetries and the search for a canonical labeling. In practice, this approach is effective when the input graph is highly symmetric, and the canonical-labeling algorithms spend a lot of time looking for symmetries (instead of a canonical labeling). Our experimental results show that this logic applies when the input graph is large and sparse.

5.3 Case Study: Analyzing Runtime for an Example Graph

This section analyzes and compares the run times of **saucy** and **bliss** in the search for the symmetries of an example graph shown in Figure 5.3. This graph has n vertices, $n/2$ edges, average degree of 1, and the symmetry group size of $2^{n/2}(n/2)!$ The search trees generated by **saucy** and **bliss** for this graph are demonstrated in Figures 5.4 and 5.5, respectively. The black nodes in these two trees correspond to

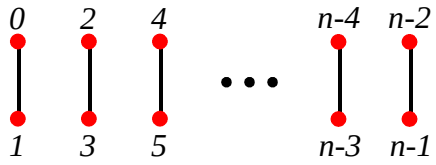


Figure 5.3: An n-vertex $(n/2)$-edge graph with symmetry group size of $2^{n/2} \times (n/2)!$

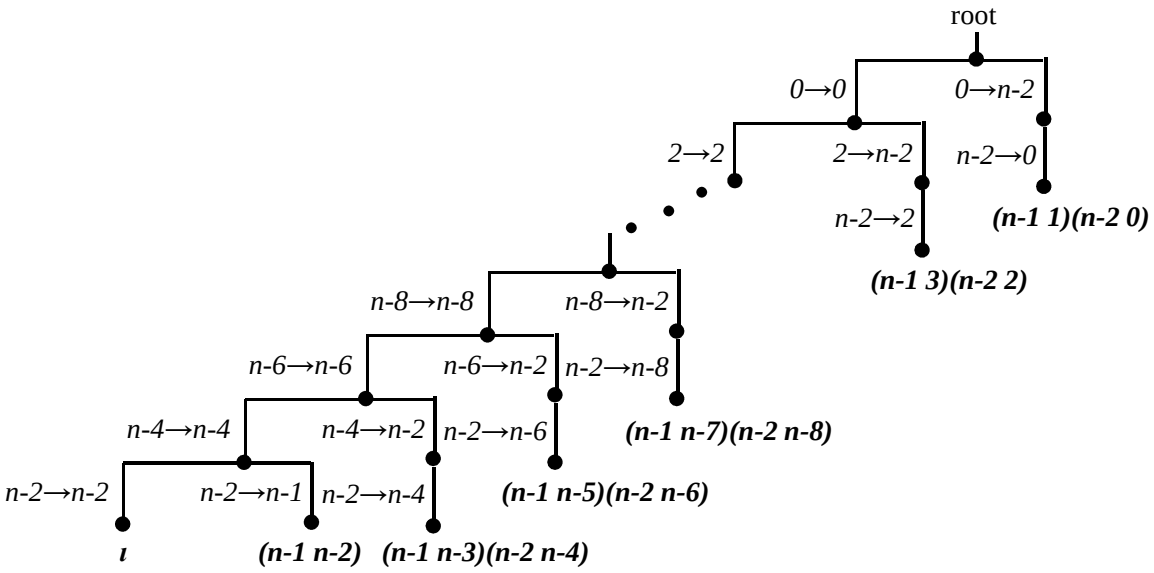


Figure 5.4: Symmetry search tree constructed by **saucy** for the graph of Figure 5.3.

OPPs/permutations. It is evident that **saucy** explores fewer nodes than **bliss**, as it finds symmetries up in the tree without reaching the leaves. A detailed analysis of run time complexity of **saucy** and **bliss** for this graph is presented next.

The **saucy** search tree shown in Figure 5.4 produces $n/2$ levels after subgroup decomposition. The number of OPPs explored by **saucy** at level l is 3 for $2 \leq l \leq (n/2)$, 2 for $l = 1$, and 1 for $l = 0$ (root of the tree). The summation of all explored nodes over $n/2$ levels is:

$$\sum_{l=2}^{n/2} 3 + 2 + 1 = \sum_{l=1}^{n/2} 3 = 3n/2$$

The **bliss** search tree shown in Figure 5.5 produces $n/2$ levels after subgroup decomposition. The number of permutations explored by **bliss** at level l is n for $l = n/2$, and $2l + 1$ for $0 \leq l < n/2$. The summation of all explored nodes over $n/2$

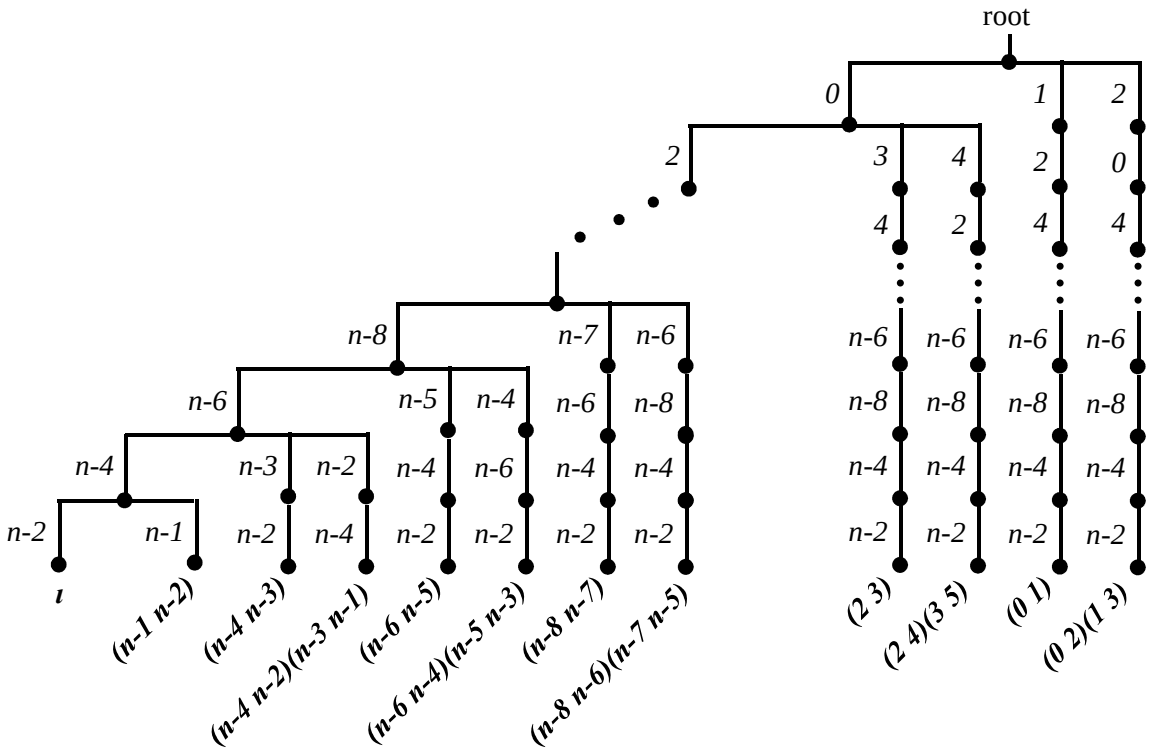


Figure 5.5: Symmetry search tree constructed by **bliss** for the graph of Figure 5.3.

levels is:

$$n + \sum_{l=0}^{n/2-1}(2l+1) = n + 2\sum_{l=0}^{n/2-1} l + n/2 = 3n/2 + (n/2-1)(n/2) = n^2/4 + n$$

The analysis above shows that **saucy** takes $\Theta(n)$ time to find the symmetries of the graph of Figure 5.3, while **bliss** takes $\Theta(n^2)$. The combination of **saucy** and **bliss** takes $\Theta(n)$ time to canonically label the graph, due to the fact that all the n vertices of the graph share the same orbit, and hence, all the subtrees encountered during **bliss** canonical-labeling search can be skipped. Our analysis discussed here matches empirical data presented next.

CHAPTER VI

Symmetry-Discovery Algorithms for Boolean Functions

In this chapter, we introduce our symmetry-discovery algorithms for Boolean formulas. Our algorithms modify the **saucy** framework (see Chapter IV) to construct a search tree that explores permutations (but not negations) of inputs and outputs of a Boolean formula. They prune away unpromising branches of search by building several *abstraction graphs* and using them to perform partition refinement. When refinement is exhausted, they resort to SAT to test candidate permutations for symmetry. They learn from SAT counterexamples to avoid similar conflicts.

Figures 6.1, 6.2, and 6.3 illustrate Boolean formulas along with corresponding search trees constructed by our symmetry-detection algorithms. In these figures, down arrows refer to refinement steps. Each refinement step uses a different set of graphs shown next to it. Throughout this chapter, we refer to these examples to illustrate our algorithms.

We assume that our algorithms take an n-input m-output Boolean function F in a form of and And-Inverter graph. Furthermore, we assume that the input set of F is $X = \{x_1, ..., x_n\}$ and its output set is $Z = \{z_1, ..., z_m\}$.

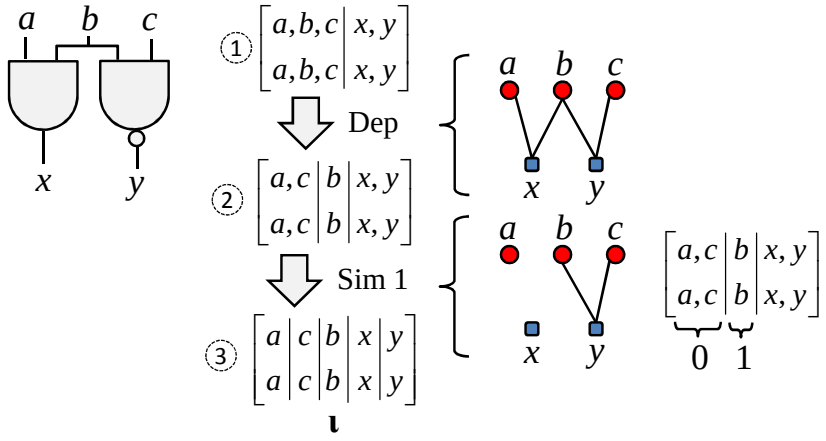


Figure 6.1: An example function with 1 symmetry.

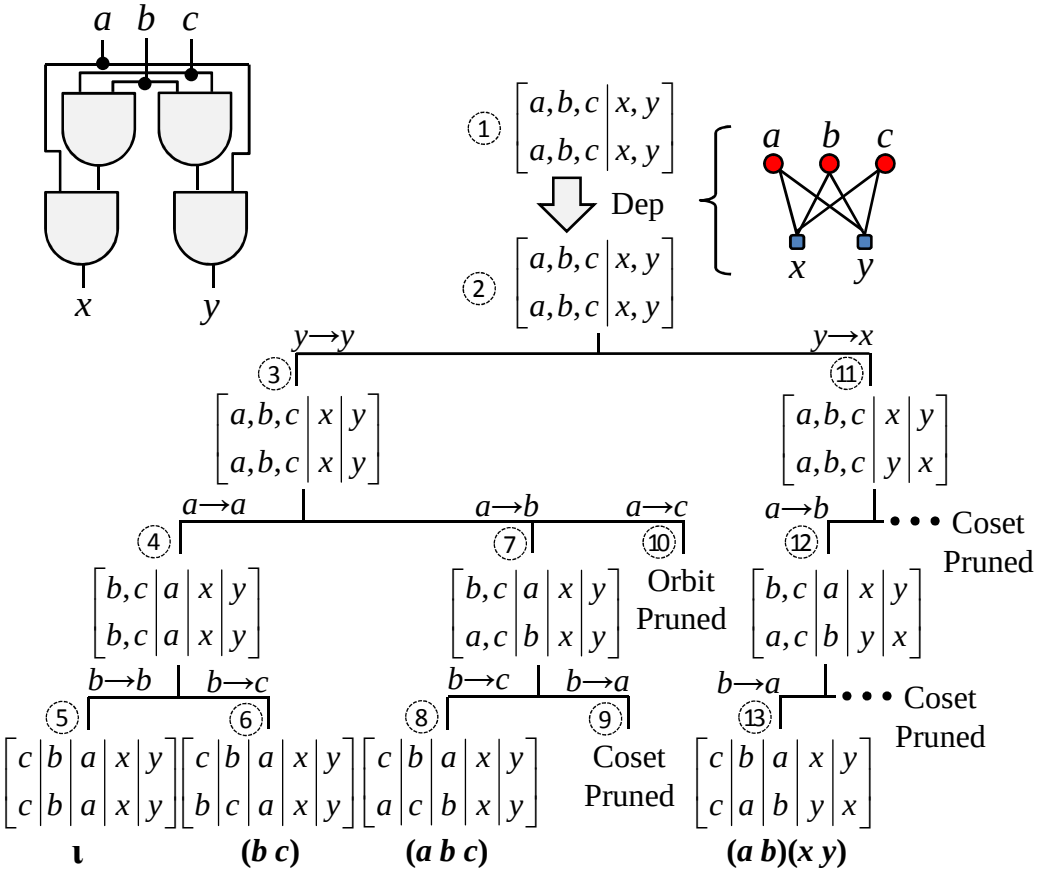


Figure 6.2: An example function with 12 symmetries.

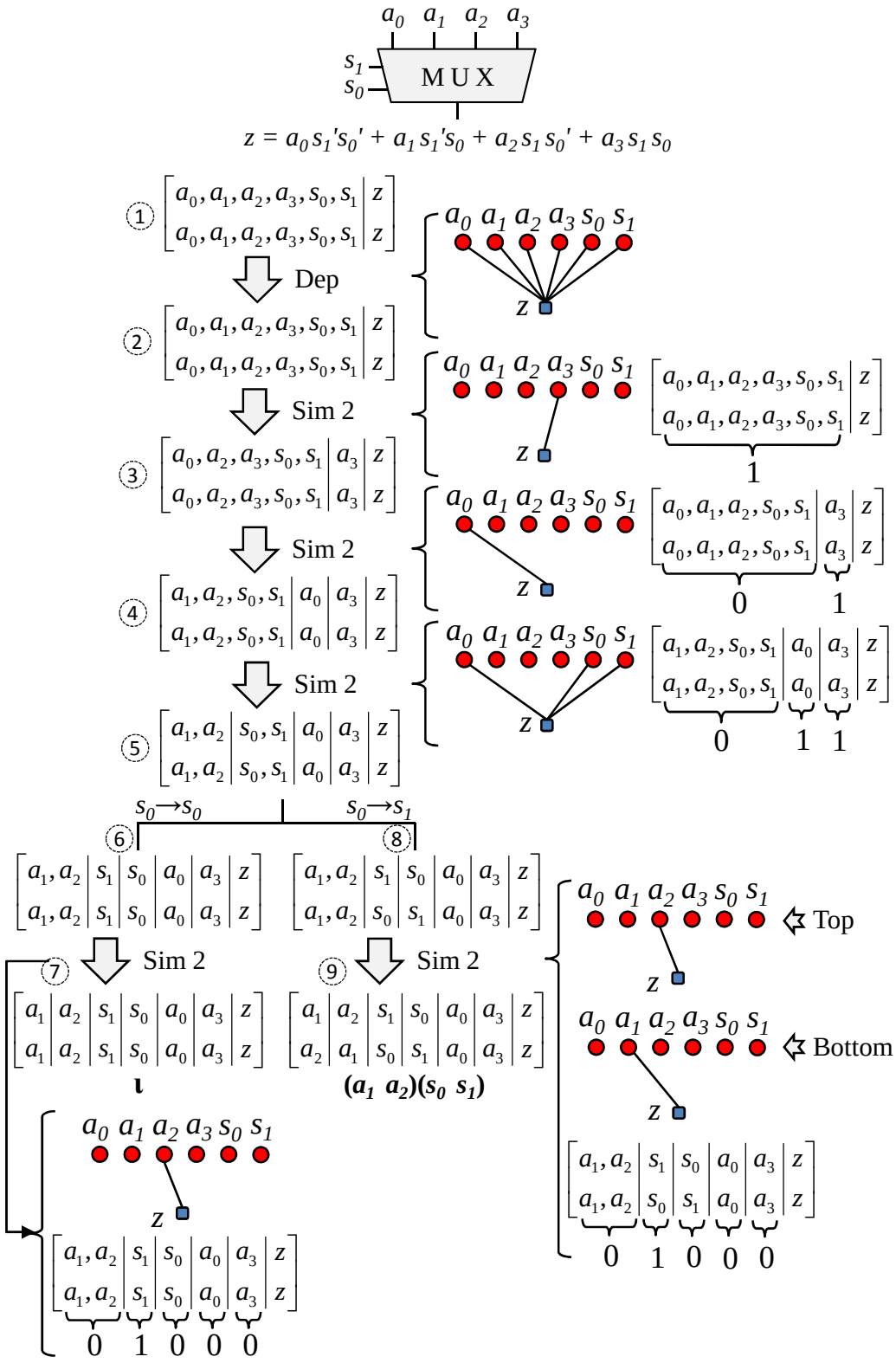


Figure 6.3: A 4-to-1 multiplexer which has 2 symmetries.

6.1 Functional Symmetries in Boolean Functions

Our symmetry-finding algorithms for Boolean functions construct a permutation search tree whose OPPs contain inputs and outputs of a Boolean function. They explore the space of permutations using the branching and backtracking framework that was introduced by **saucy**. Unlike **saucy** algorithms, they look for **functional** symmetries of a Boolean formula whereas **saucy** seeks **structural** symmetries of a graph.

To search for functional symmetry of a formula, and yet exploit the **saucy** framework, our algorithms model *partial* functionality of the formula by several *abstraction graphs*. These graphs are used by partition refinement to prune away permutations that do not yield any symmetry. Nevertheless, refinement by these graphs per se does not prove functional equivalence. This is due to the fact that these graphs capture just partial functionality of Boolean formulas, and hence might cause refinement to report *false positives*. In other words, a permutation that is not a symmetry of an abstraction graph is not a symmetry of the Boolean function, but a symmetry of abstraction graphs does not necessarily form a symmetry of the Boolean function.

To confirm functional equivalence when refinement is exhausted, our algorithms invoke SAT. If SAT disproves functional equivalence, i.e., detects a *functional conflict*, a counterexample is returned. These counterexamples are learned by our algorithms to avoid recurring conflicts.

In the remainder of this chapter, we first explain abstraction graphs and their role in partition refinement. Then, we discuss SAT-based symmetry verification and propose a learning mechanism from SAT counterexamples to avoid recurring conflicts. Next, we provide an outline of our symmetry-detection algorithms for Boolean functions. At the end, we formulate Boolean matching as a symmetry-detection problem.

6.2 Abstraction Graphs

Abstraction graphs are two-colored bipartite graphs constructed to partially capture the functionality of a Boolean function. These graphs are used by partition refinement to prune the permutation space. Here, we introduce three types of abstraction graphs; one based on functional dependency, and two based on random simulation.

6.2.1 Dependency-Based Abstraction Graph

The *dependency graph* of a Boolean function encodes the supports of the function as a graph using the following procedure. A red vertex is added for each input, and a blue vertex for each output. An edge is added between input x and output z if and only if $x \in supp(z)$.

For the functions of Figures 6.1, 6.2, and 6.3, the dependency graphs are depicted where refinement is labeled with "Dep". In Figure 6.1, the dependency graph encodes the fact that output x is dependent on inputs a and b, and output y is dependent on inputs b and c. Likewise, the dependency graph in Figure 6.2 encodes the fact that outputs x and y are dependent on inputs a, b, and c, and the dependency graph in Figure 6.3 shows that output z is dependent on all inputs of the multiplexer.

We build dependency graphs to distinguish outputs (resp. inputs) that have different functional dependency (resp. influence). For example, at node (2) of the tree in Figure 6.1, input b is separated from inputs a and c, since the degree of b in the dependency graph is different from that of a and c.

6.2.2 Simulation-based Abstraction Graphs

We construct two types of simulation graphs based on proper random input vectors. Intuitively, a proper random input vector assigns the same value to all inputs that are not yet distinguished. For the functions of Figures 6.1, 6.2, and 6.3, type-1

and type-2 simulation graphs are depicted where refinement is labeled with "Sim 1" and "Sim 2", respectively. These graphs are built based on proper input vectors that are shown next to them.

Given a proper input vector $P = \langle p_1, ..., p_n \rangle$ (with regard to partition π of input set X) and its corresponding output vector $R = \langle r_1, ..., r_m \rangle$, we build two types of simulation graphs as follows:

- *Simulation Graph Type 1*: We add a red vertex for each input, and a blue vertex for each output. We add an edge between $z_i \in Z$ and all inputs $x \in supp(z_i)$, if and only if $r_i = 1$. For example, the simulation graph at node (2) of the tree in Figure 6.1 encodes the following: if $a = c = 0$ and $b = 1$, then $x = 0$ and $y = 1$.

- *Simulation Graph Type 2*: We add a red vertex for each input, and a blue vertex for each output. We then flip each $p_i \in P$, one at a time, and save the resulting n input vectors in $P_1, ..., P_n$. We simulate $P_1, ..., P_n$ and record the resulting n output vectors in $R_1 = \langle r_1^1, ..., r_m^1 \rangle, ..., R_n = \langle r_1^n, ..., r_m^n \rangle$. We add an edge between $z_j \in Z$ and $x_i \in X$, if and only if $r_j^i \neq r_j$. For example, the simulation graph at node (7) of the tree in Figure 6.3 encodes the following: if $a_1 = a_2 = s_0 = a_0 = a_3 = 0$ and $s_1 = 1$, flipping a_2 flips z.

We build type-1 simulation graphs primarily to distinguish outputs. Once outputs are distinguished, inputs might be distinguished as well. For example, at node (3) of the tree in Figure 6.1, output y is separated from output x, and subsequently, input c is separated from input a. Furthermore, we build type-2 simulation graphs to distinguish inputs (outputs) that have different observability (controllability). For example, at node (7) of the tree in Figure 6.3, input a_2 is separated from input a_1, since the observability of a_2 is different from that of a_1 (with regard to the given random input vector).

6.3 Refinement by Abstraction Graphs

In the **saucy** framework, partition refinement is applied *simultaneously* to the top and bottom partition of an OPP, 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 implementation, it first refines the top partition until it becomes equitable, and records where the cell splits occur. Then, it starts refining the bottom partition, and compares the splitting locations of the bottom to the top (i.e., checks the isomorphism of the two partitions) after each refinement step. It also ensures that the connections of each newly created cell on the bottom match the connections of its corresponding cell on the top.

Similar to **saucy**, our algorithms invoke two separate refinement routines for the top and bottom partitions. Nonetheless, our algorithms refine the permutation space based on several abstraction graphs, while refinement in **saucy** uses only one global graph throughout the search. We explain the refinement procedure based on multiple graphs in this section.

Figure 6.4 demonstrates the refinement routine for the top partition. This routine primarily refines the top partition by the dependency graph (lines 1-2). It then generates a number of proper input vectors with regard to the subset of the top partition that includes just the inputs of the function (lines 3-6 and 10). Next, it builds type-1 simulation graphs based on the generated input vectors (line 7), and uses them to refine the top partition (line 8). It refines once more by dependency graph if new cells were induced at the previous step (line 9). It also saves the bit vectors whose corresponding simulation graphs caused further refinement (line 9). These vectors will later be used by the refinement of the bottom partition to generate consistent input vectors. This routine ends by following similar refinement steps (as explained above) for type-2 simulation graphs (line 11).

Figure 6.5 demonstrates the refinement routine for the bottom partition. This

Inputs: Π, G_D
Outputs: L_1, L_2

1. Set π_T to the top partition of OPP Π.
2. Refine partition π_T by dependency graph G_D.
3. Build ordered partition $\pi = [W_1|\cdots|W_t] \subset \pi_T$ by removing the cells of π_T that contain the outputs.
4. Set counter $i = 0$.
5. Generate random bit vector $B = \langle b_1, ..., b_t \rangle$, where $b_i \in \{0, 1\}$.
6. Generate proper random input vector $P = \langle p_1, ..., p_n \rangle$ with regard to π, where $p_i = b_j$ if input $x_i \in W_j$.
7. Generate type-1 simulation graph G that corresponds to input vector P.
8. Refine π_T by simulation graph G.
9. If new cells are induced at line 8, refine π_T by dependency graph G_D, save bit vector B in list L_1, and set counter $i = 0$.
10. Increment i. Go to line 5 if $i < 200$.
11. Repeat lines 4-10, but this time generate type-2 simulation graph G at line 7, and save bit vector B in list L_2 at line 9.

Figure 6.4: Pseudocode for refining the top partition of OPP Π.

routine resembles that of Figure 6.4, but with two main differences. First, it does not generate new random bit vectors. Instead, it uses the ones generated by Figure 6.4 to make pairs of consistent input vectors (lines 5-6). In other words, it assigns the same Boolean value to all potentially mappable inputs of the top and bottom partitions. Second, it checks OPP isomorphism during refinement, and returns 0 if a conflict is detected (lines 8-9).

In the tree of Figure 6.3, refinement at node (8) assigns the same Boolean value to all inputs in the same-index cells of the top and bottom partitions. It then refines the OPP using type-2 simulation graph. The result of refinement is the isomorphic OPP at node (9).

Inputs: Π, G_D, L_1, L_2
Outputs: 0 or 1

1. Set π_B to the bottom partition of OPP Π.
2. Refine partition π_B by dependency graph G_D.
3. Build ordered partition $\pi = [W_1|\cdots|W_t] \subset \pi_B$ by removing the cells of π_B that contain the outputs.
4. Set counter $i = 0$.
5. Set bit vector $B = \langle b_1, ..., b_t \rangle$ to the i-th element of list L_1.
6. Generate proper random input vector $P = \langle p_1, ..., p_n \rangle$ with regard to π, where $p_i = b_j$ if input $x_i \in W_j$.
7. Generate type-1 simulation graph G that corresponds to input vector P.
8. Refine π_B by simulation graph G. Return 0 if a conflict is detected.
9. Refine π_B by dependency graph G_D. Return 0 if a conflict is detected.
10. Increment i. Go to line 5 if $i < size(L_1)$.
11. Repeat lines 4-10, but this time set bit vector B to the i-th element of list L_2 at line 5, generate type-2 simulation graph G at line 7, and check $i < size(L_2)$ at line 10.
12. Return 1.

Figure 6.5: Pseudocode for refining the bottom partition of OPP Π.

6.4 Checking Functional Equivalence by SAT

A candidate symmetry (returned by refinement) needs to be verified by SAT, since refinement by abstraction graphs per se does not prove functional equivalence. Figure 6.6 shows the routine that performs this verification. This routine first duplicates the function (line 1), and permutes the I/Os of one function according to the candidate symmetry (line 2). It then builds the miter of the original and permuted functions (line 3), and hands it off to SAT (line 4). If SAT finds the miter unsatisfiable, a symmetry is found; otherwise, a *functional conflict* is detected, and a counterexample is saved (line 5). In Figures 6.1, 6.2, and 6.3, no functional conflict

Inputs: γ, f
Outputs: 0 or 1, C

1. Duplicate function f and save it in function g.
2. Permute g's I/Os according to permutation γ.
3. Build miter ϕ of f and g.
4. Pass ϕ to SAT.
5. If ϕ is unsatisfiable, return 1. Otherwise, save SAT counterexample in input vector C and return 0.

Figure 6.6: Pseudocode for checking functional equivalence by SAT.

is detected, i.e., all discrete OPPs form symmetries of the corresponding functions.

6.5 Learning From SAT Counterexamples

Partition refinement typically reduces the number of possible matches from $n!m!$ to hundreds or less, often making exhaustive search (with SAT-based equivalence checking) practical. However, this phase of search can be significantly improved by learning from SAT counterexamples.

A SAT counterexample is in the form of an input vector that forces the miter of the original function and the permuted function to be satisfiable. Our algorithm learns a collection of such input vectors, along with their corresponding output vectors, and uses them to backjump to the tree level where functional conflicts are resolved.

Figure 6.7 shows the routine that learns from a SAT counterexample. This routine makes two copies of the counterexample (lines 1-2), and permutes one copy based on the candidate symmetry (line 3). It then simulates the function with the two copies, and saves the results in simulation pairs of the form $\langle$input vector, output vector$\rangle$ (lines 4-6). It attaches the two simulation pairs to the *database of simulation pairs*, and set their *activities* to zero (lines 7-8). The activity of a simulation pair quantifies its participation in conflict detection. This routine ends by potentially *reducing* the

Inputs: γ, f, C, DB
Outputs: None

1. Set input vectors P to SAT counterexample C.
2. Set input vectors Q to SAT counterexample C.
3. Permute input vector Q based on permutation γ.
4. Simulate function f with input vector P and save the simulation result in output vector R.
5. Simulate function f with input vector Q and save the simulation result in output vector U.
6. Build simulation pairs $\langle P, R\rangle$ and $\langle Q, U\rangle$.
7. Check if simulation pair $\langle P, R\rangle$ already exists in database DB. If no, add $\langle P, R\rangle$ to DB, and set the activity of $\langle P, R\rangle$ to 0.
8. Repeat line 7, but this time for simulation pair $\langle Q, U\rangle$.
9. If $size(DB) > 50$, reduce DB.

Figure 6.7: Pseudocode for learning from a SAT counterexample.

database (line 9) by finding the median of the activities of all simulation pairs and deleting pairs whose activities fall below the median.

Once a functional conflict is detected, our backjumping routine backtracks one level up at a time, checks the current OPP for functional conflicts, and returns once the OPP is found free of conflicts.

Figure 6.8 shows the routine that checks for a functional conflict in an OPP. This routine searches the database of simulation pairs for two consistent input vectors (lines 1-13). Suppose that it finds input vectors P and Q consistent with regard to the OPP, and suppose that $\langle P, R\rangle$ and $\langle Q, U\rangle$ are the simulation pairs that correspond to P and Q. This routine counts the number of outputs in each cell of the top (resp. bottom) partition whose values in R (resp. U) is one (lines 14-15). It declares a conflict, if two same-index cells of the top and bottom partitions have different counters (line 16). In fact, it anticipates that such a pair of cells will eventually map two outputs

whose simulation values (i.e., functional behaviors) are different under P and Q. At the end, it increases the activity of $\langle P, R\rangle$ and $\langle Q, U\rangle$ to credit their participation in conflict detection (line 16).

Figure 6.9 shows an example of learning when refinement is disabled.[1] For this example, our algorithm encounters functional conflicts at nodes (4) and (6). While backtracking from node (6), it finds that node (5) has a functional conflict under simulation pairs Γ_3 and Γ_4. Hence, it backtracks from node (6) to node (8), and skips node (7).

6.6 Symmetry Discovery for Boolean Functions

Similar to **saucy**, our proposed symmetry-detection algorithms for Boolean functions traverse the space of permutations in a depth-first manner. They enable coset and orbit pruning by performing a phase of subgroup decomposition on the left-most tree path. In other words, "decisions" along that path map each selected target I/O to itself.

Partition refinement by abstraction graphs is invoked before selecting and branching on a target I/O. The tree is pruned by systematic application of four pruning rules:

- **Coset pruning** which terminates the search in a coset subtree as soon as a coset representative is found.
- **Orbit pruning** which avoids searching the subtree of the coset that maps I/O i to I/O j, if j is already in the orbit of i.
- **Non-isomorphic OPP pruning** which indicates that there are no permutations in the subtree rooted at that node which are symmetries of the Boolean function.

[1] A search tree that could illustrate learning and perform partition refinement was too large to fit in this thesis.

Inputs: Π, *DB*
Outputs: 0 or 1

1. Set ordered partition π_T and ordered partition π_B to the top and bottom partitions of OPP Π, respectively.
2. Build two ordered partitions π_T^i and π_T^o from partition π_T, where π_T^i contains cells of π_T that have inputs, and π_T^o contains cells of π_T that have outputs.
3. Repeat line 2, but this time for partition π_B, and save the resulting sub-partitions in π_B^i and π_B^o.
4. Set counter $i = 0$.
5. Return 1 if $i \geq size(DB)$.
6. Set simulation pair $\langle P, R \rangle$ to the i-the element of database *DB*.
7. Check if input vector P is proper with regard to partition π_T^i. If no, increment i, and go to line 5.
8. Set counter $j = 0$.
9. Check $j \geq size(DB)$. If yes, increment i and go to line 5.
10. Set simulation pair $\langle Q, U \rangle$ to the j-the element of database *DB*.
11. Check if input vector Q is proper with regard to partition π_B^i. If no, increment j, and go to line 9.
12. Build OPP Π^i by putting partitions π_T^i and π_B^i as the top and bottom partitions of Π^i, respectively.
13. Check if input vectors P and Q are consistent with regard to Π^i. If no, increment j, and go to line 9.
14. Set cell C_k to the k-th cell of π_T^o. Set N_k to the number of outputs in C_k whose value in R is 1. Do this for all $1 \leq k \leq size(\pi_T^o)$.
15. Set cell C_k to the k-th cell of π_B^o. Set M_k to the number of outputs in C_k whose value in U is 1. Do this for all $1 \leq k \leq size(\pi_B^o)$.
16. Check if $N_k \neq M_k$ for some k. If yes, increase activity of pairs $\langle P, R \rangle$ and $\langle Q, U \rangle$, and return 0.
17. Return 1.

Figure 6.8: Pseudocode for checking functional conflicts in isomorphic OPP Π.

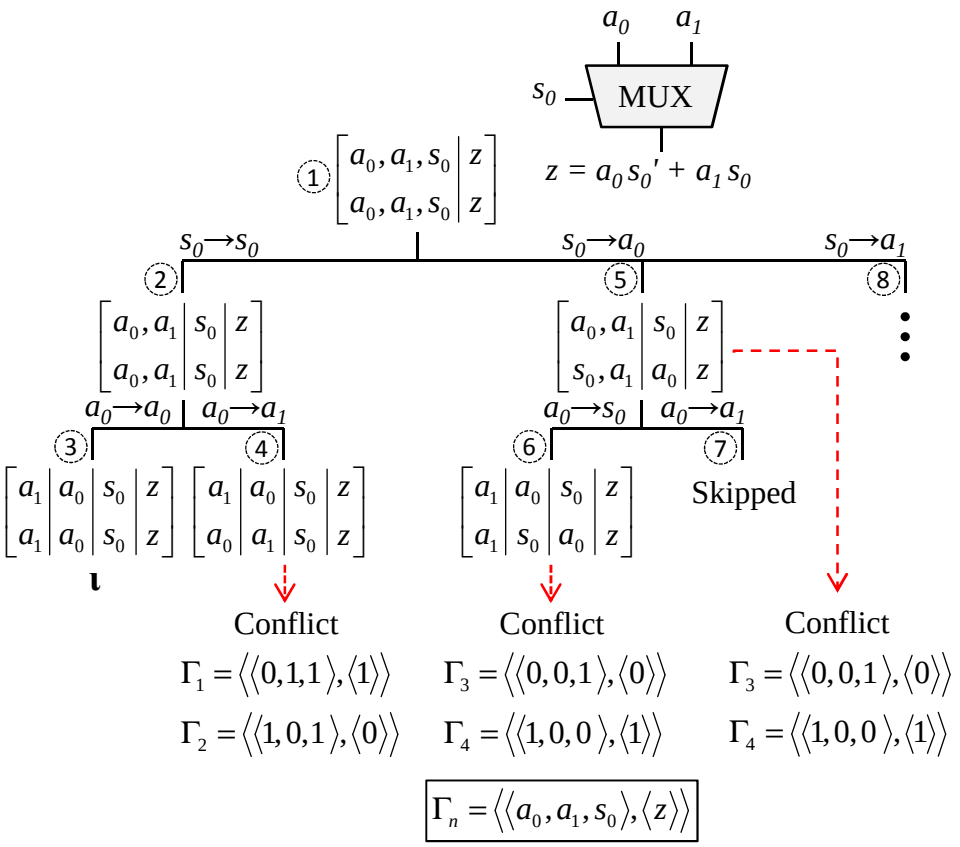


Figure 6.9: An example function with its partial search tree built by our symmetry-detection algorithm when refinement is disabled.

– **SAT-based pruning** which uses SAT to backtrack from functional conflicts, and uses SAT counterexamples to avoid recurring conflicts later in the search.

Our symmetry-detection algorithms here disable matching OPP pruning; a pruning technique that was introduced in the **saucy** framework. This is due to the fact that partition refinement by abstraction graphs might produce false positives, which can be in the form of matching OPPs. Note that disabling matching OPP pruning forces our algorithms to find potential symmetries at the leaves.

To reduce the branching factor of our permutation tree, our branching procedure first maps outputs and then inputs. Our experiments show that once outputs are distinguished, inputs can be distinguished quickly. Furthermore, it picks the first smallest non-singleton cell whose outputs (inputs) have the highest functional de-

pendency (influence). Its rationale is that, once such I/Os are separated, partition refinement can use them to effectively distinguish more I/Os.

Below is the trace of our symmetry-detection algorithms for the 4-to-1 multiplexer of Figure 6.3:

– Initialization: $\Theta = \{a_0|\ a_1|\ a_2|\ a_3|\ s_0|\ s_1|\ z\}$, $Z = \emptyset$.

1. Separate inputs of the multiplexer from its outputs

2. Refine the OPP by the dependency graph

3. Set all inputs to 1 and simulate the multiplexer;
   Build the type-2 simulation graph;
   Refine the OPP by the type-2 simulation graph

4. Set all inputs except a_3 to 0 (set a_3 to 1) and simulate the multiplexer;
   Build the type-2 simulation graph;
   Refine the OPP by the type-2 simulation graph

5. Set all inputs except a_0 and a_3 to 0 (set a_0 and a_3 to 1) and simulate the multiplexer;
   Build the type-2 simulation graph;
   Refine the OPP by the type-2 simulation graph

6. Fix input s_0

7. Set all inputs except s_1 to 0 (set s_1 to 1) and simulate the multiplexer;
   Build the type-2 simulation graph;
   Refine the OPP by the type-2 simulation graph;
   $\mathcal{G}_{s_0} = \iota$

8. Search for representative of coset $H_{s_0 \mapsto s_1}$

9. On the top partition, set all inputs except s_1 to 0 (set s_1 to 1) and simulate the multiplexer;

   On the bottom partition, set all inputs except s_0 to 0 (set s_0 to 1) and simulate the multiplexer;

   Build the type-2 simulation graphs;

   Refine the OPP by the type-2 simulation graphs;

   Verify the candidate symmetry by SAT;

   Found representative of coset $H_{s_0 \mapsto s_1}$;

   $Z = \{(a_1\ a_2)(s_0\ s_1)\}$; $\Theta = \{a_0 \mid a_1, a_2 \mid a_3 \mid s_0, s_1 \mid z\}$; $|\mathcal{G}_{s_0}| = |\mathcal{G}_{s_0}| \times |\Theta_{s_0}| = 1 \times 2 = 2$

6.7 Case Study: Checking for PP-Equivalence

The *PP-equivalence checking* problem seeks functional equivalence of two functions under permutation of their I/Os (the first P stands for permutation of inputs and the second P stands for permutation of outputs) [37, 2]. In other words, PP-equivalence checking verifies the isomorphism of two functions under permutation of I/Os. Here, we explain how our symmetry-detection algorithm can be modified to solve the PP-equivalence checking problem.

An automorphism of a Boolean function is an isomorphism to itself. Hence, one can check isomorphism of two functions by putting them side by side, and passing them to an automorphism-detection algorithm that incorporates the two following modifications.

– The isomorphism-checking algorithm only needs to look for permutations that map one function to the other. In other words, it immediately prunes subtrees that (partially) map one function to itself.

– The isomorphism-checking algorithm only needs to find one symmetry to con-

firm isomorphism.

The isomorphism-checking algorithm described above does not perform subgroup decomposition, since subgroup decomposition maps a graph to itself. Furthermore, it does not use coset or orbit pruning, since it terminates the search as soon as it finds one symmetry.

CHAPTER VII

Empirical Evaluation

In this chapter, we test the performance of **saucy** symmetry-detection algorithms, namely, **saucy**, on a collection of graph benchmarks, including those from the SAT 2011 competition, verification, place and route, and networks. We examine the performance of our proposed canonical-labeling approach on the subset of the graphs that are very large and very sparse. Furthermore, we test the performance of our algorithms on a collection of benchmarks from ISCAS'85, ISCAS'89, MCNC, and ITC'99.

7.1 Empirical Results for Graph Symmetry Detection

Table 7.1 lists the families of the graph benchmarks in our collection. It includes 1439 benchmarks drawn from a wide variety of application domains. These benchmarks are divided into four families:

– **saucy** benchmarks: this set contains 92 very large and very sparse graphs first assembled to test **saucy**'s scalability. This suite represents graphs from logic circuits and their physical layouts [57, 1], Internet routers [18, 32], and road networks in the US states and its territories [16].

– SAT 2011 benchmarks: this set consists of 1200 SAT 2011 competition CNF

Table 7.1: Families of graph benchmarks.

Family	Inst.	Smallest Instance		Largest Instance		Description
		vertices	edges	vertices	edges	
`circuit`	33	3,575	14,625	4,406,950	8,731,076	**saucy**
`router`	3	112,969	181,639	284,805	428,624	graphs
`roadnet`	56	1,158	1,008	1,679,418	2,073,394	
`Application`	300	453	859	32,813,545	65,487,132	SAT 2011
`Crafted`	300	105	320	776,820	3,575,337	CNFs
`Random`	600	1,165	5,375	310,000	680,000	
`binnet`	27	1,000	720	9,000,000	658,675	networks
`mz`	25	40	60	1,000	1,500	Miyazaki
`cmz`	46	120	90	200	1,900	graphs (`mz`),
`mz-aug`	25	40	92	1,000	2,300	and their
`mz-aug2`	24	96	152	1,200	1,900	variants

benchmarks [52]. These benchmarks are divided into three categories: Application (300 benchmarks), Crafted (300 benchmarks), and Random (600 benchmarks). The CNF benchmarks are modeled as graphs using the procedure explained in Section 3.6.

– binary networks: this set includes graph benchmarks proposed to test community-detection algorithms [40]. We generated 27 undirected and unweighted binary networks using the implementation of the procedure described in [40] (available at [10]). We set the number of nodes to $\{1, ..., 9\} \times \{10^3, 10^4, 10^5\}$, but fixed 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 in all instances.

– Miyazaki graphs: this set comprises Miyazaki graphs [46, 36] and their variants designed to mislead the **bliss** cell selector. It consists of 120 graphs, of which 25 are the original Miyazaki graphs and the remaining are their variants.

We ran symmetry detection using **saucy** 3.0 on the complete set of benchmarks in Table 7.1. All 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 to all experiments.

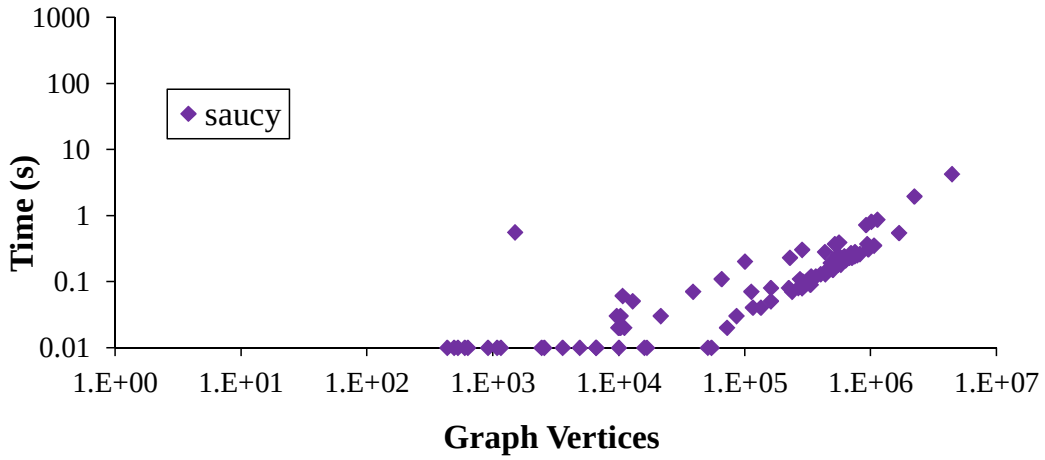
The runtime results are shown in Figure 7.1. In total, 1432 out of 1439 benchmarks were solved within a 1000 seconds time-out limit. All the 7 unsolved benchmarks belonged to the Crafted category of the SAT 2011 competition benchmarks. In general, SAT 2011 benchmarks were more challenging for **saucy** than the remaining benchmark families.

Of the 92 **saucy** benchmarks, all were solved in less than 5 seconds. The largest runtime for those graphs was 4.26 seconds, which was reported for the largest graph with more than 4.4 million vertices.

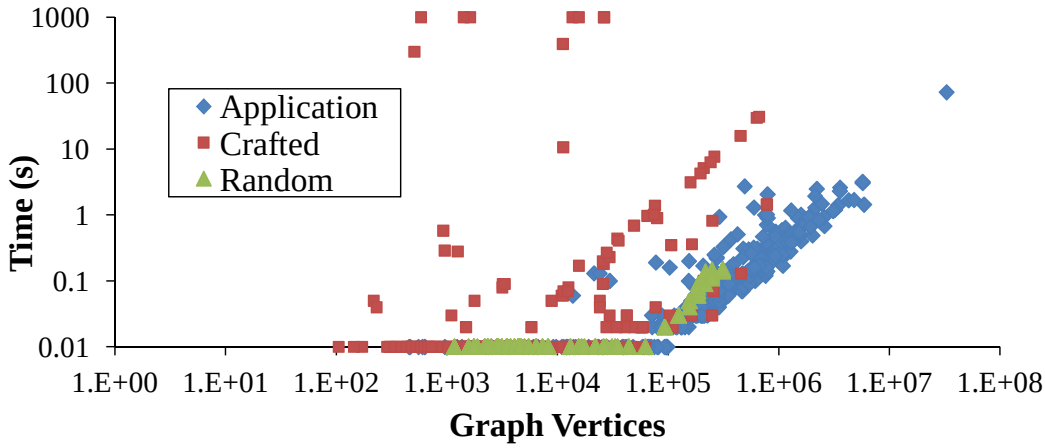
Of the 1200 SAT 2011 benchmarks, only 7 were reported unsolved. Of the remaining 1191 benchmarks, 1157 were completed in less than a second (279 Application, 278 Crafted, and all 600 Random benchmarks). There were only two benchmarks (both from the Crafted category) that took more than 100 seconds to complete. In general, instances from the Crafted category were more challenging for **saucy** than similarly-sized instances from the Random or Application suites.

Of the 27 binary networks, all were processed in less than two seconds. The largest runtime for these graphs was 1.6 seconds, which was reported for the largest binary network with 9 million vertices.

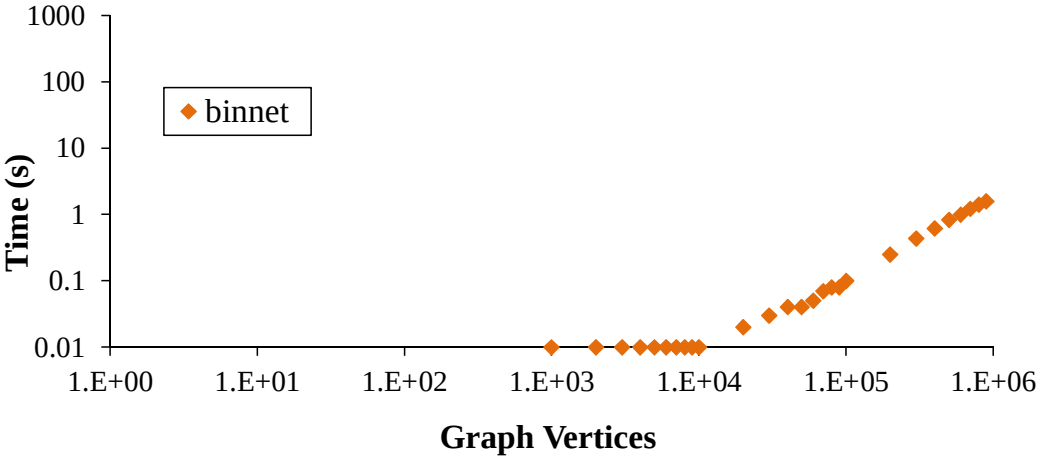
Of the 120 Miyazaki graphs, all were solved in less than a second. Miyazaki graphs are fairly small graphs, but are known to impede leading graph symmetry-detection algorithms. The results here show that those graphs are not challenging for **saucy**.



(a) Runtime results for **saucy** benchmarks

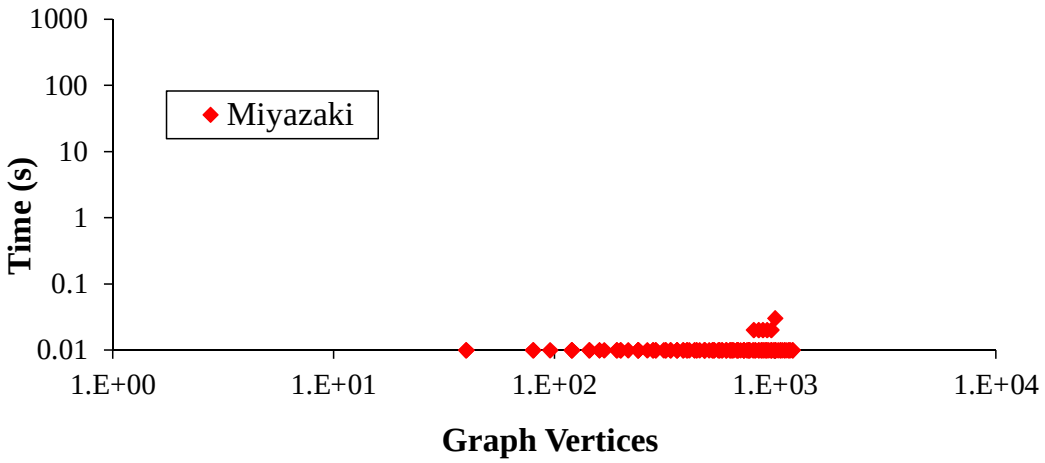


(b) Runtime results for SAT 2011 CNFs



(c) Runtime results for binary networks

Figure 7.1: This figure continues on the next page.

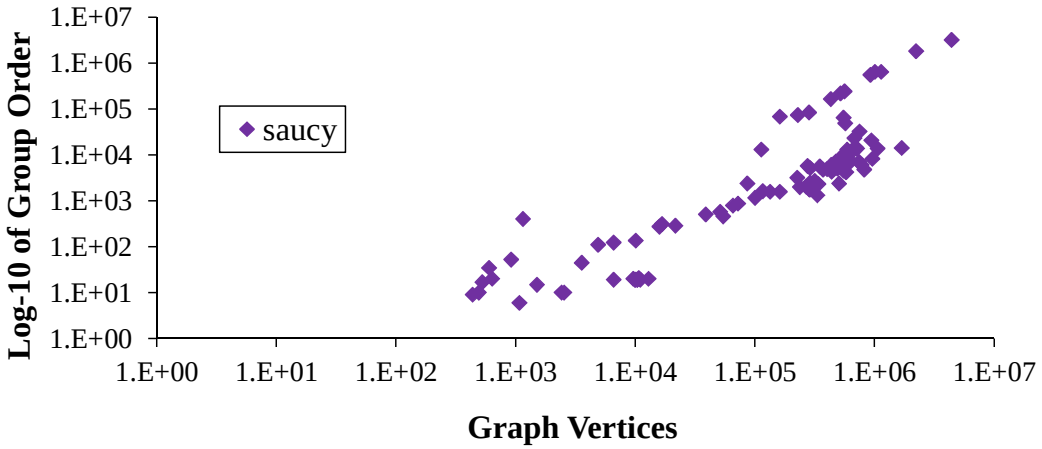


(d) Runtime results for Miyazaki graphs

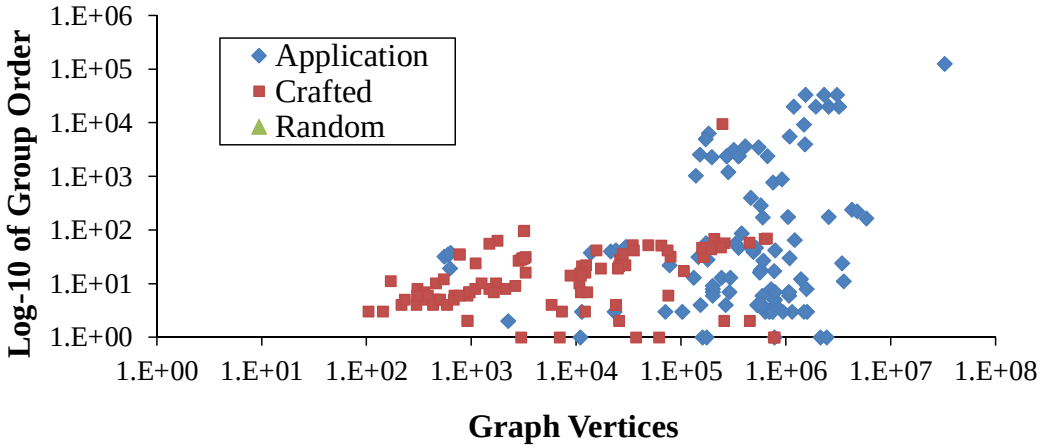
Figure 7.1: **saucy** 3.0 runtime, in seconds, as a function of graph size for the (a) **saucy** benchmarks, (b) SAT 2011 CNFs, (c) binary networks, (d) Miyazaki graphs. A time-out of 1000 seconds was applied.

It can also be inferred from Figure 7.1 that there is a weak trend towards larger runtimes for larger graphs (this trend is much stronger for binary networks and **saucy** graphs). However, runtime seems to also depend on other attributes of a graph besides its absolute size (number of vertices.) In any case, **saucy** is extremely fast, finishing in less than one second on 96% (1391) of all benchmarks.

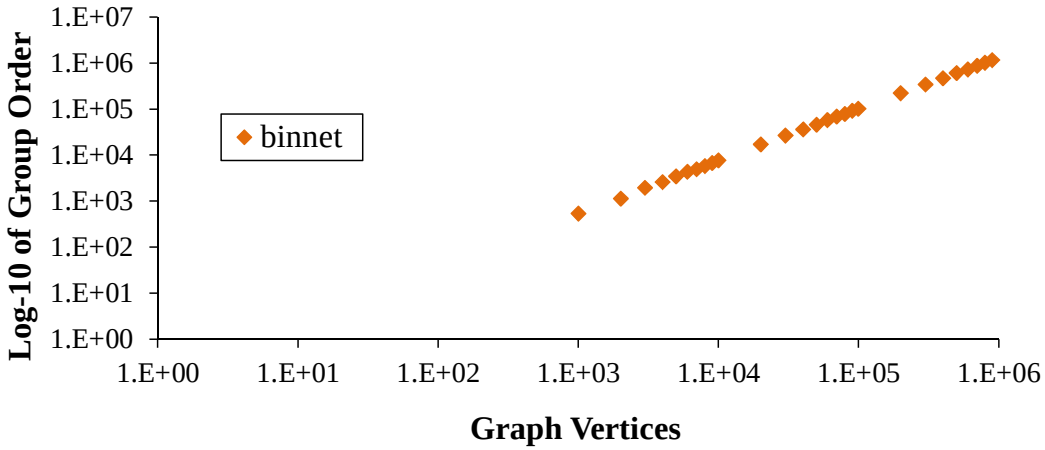
The "amount" of symmetry present (order of the automorphism group) in each benchmark is shown in Figure 7.2. This figure depicts the base-10 logarithm of symmetry group order as a function of graph size. In total, 508 out of 1439 (35%) benchmarks exhibited non-trivial symmetry. These included all **saucy** benchmarks, binary networks, Miyazaki graphs, and 269 out of 1200 SAT 2011 CNF instances (153 from Application and 116 from Crafted). All Random CNF instances showed only one trivial symmetry, and hence, are not listed in Figure 7.2. The order of the largest automorphism group was an astronomical 4×10^{3232782}, reported for a graph from the **saucy** suite. Note that the 7 SAT 2011 benchmarks that were reported unsolved are not listed in Figure 7.2.



(a) Group order for **saucy** benchmarks

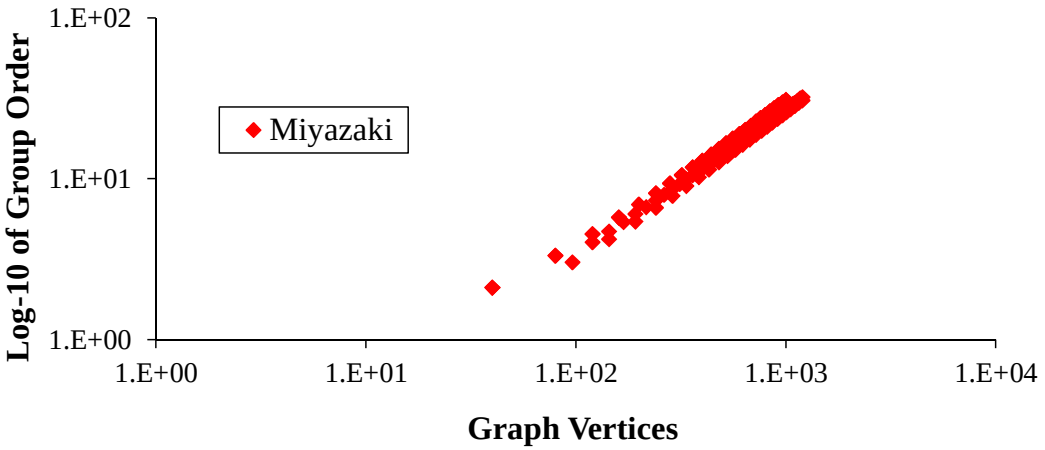


(b) Group Order for SAT 2011 CNFs



(c) Group Order for binary networks

Figure 7.2: This figure continues on the next page.



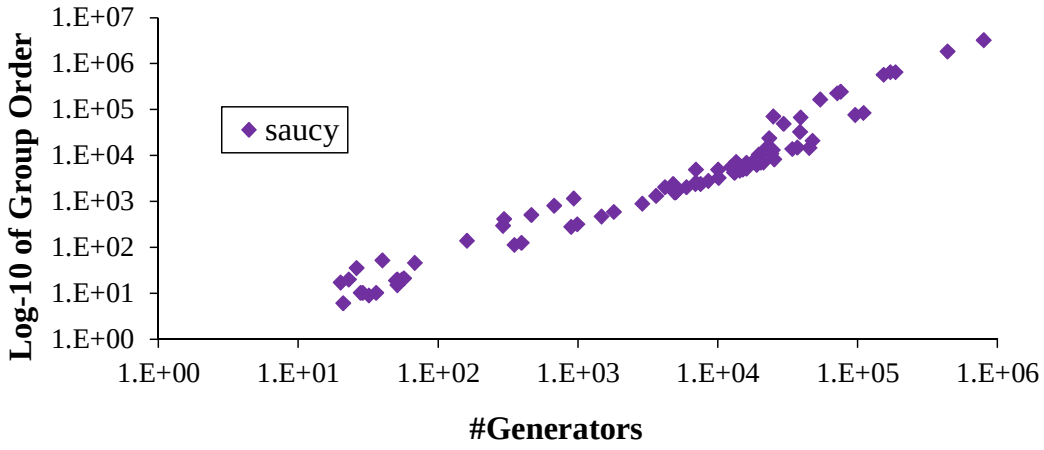
(d) Group Order for Miyazaki graphs

Figure 7.2: **saucy** 3.0 group order, as a function of graph size for the (a) **saucy** benchmarks, (b) SAT 2011 CNFs, (c) binary networks, and (d) Miyazaki graphs.

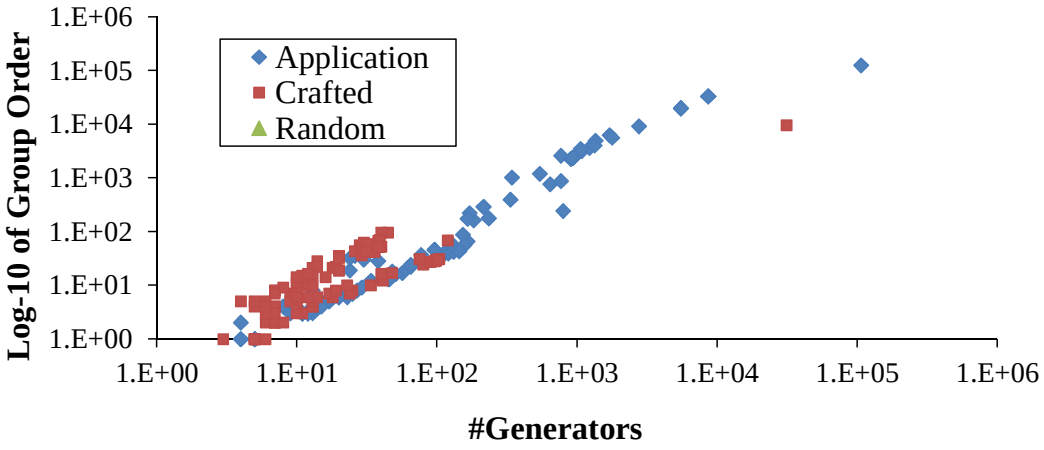
Figure 7.3 shows the relation between the order of the automorphism group and the number of generators returned by **saucy** for the 508 benchmarks that exhibited non-trivial symmetry. Symmetry-detection algorithms, including **saucy**, guarantee to produce no more than $n-1$ generators for an n-vertex graph. The number of reported generators in these results is significantly less than $n-1$. This, however, is not inconsistent with the well-known fact that the number of (irredundant) generators is exponentially smaller than the order of the corresponding symmetry group.

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 [11]) on all the 1439 benchmarks listed in Table 7.1, and compared its runtimes to those obtained from **saucy** 3.0. This comparison is shown in Figure 7.4.

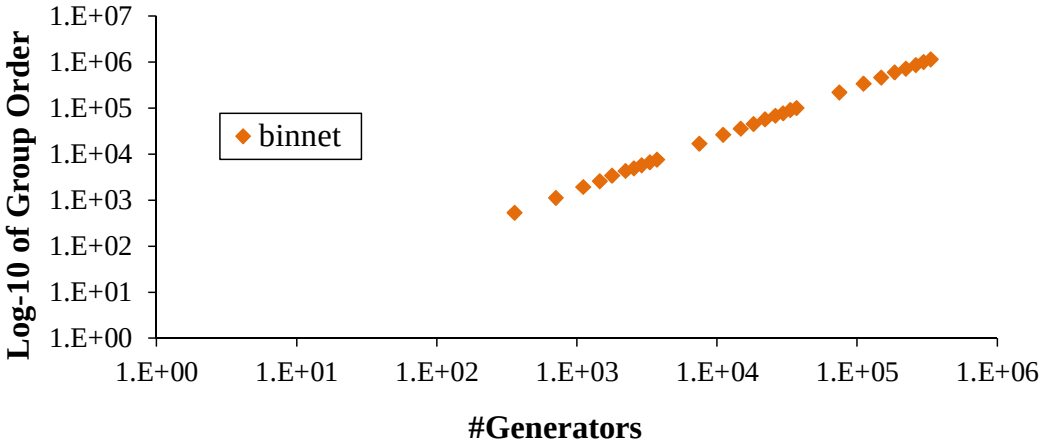
Of the four Miyazaki graph families, **bliss** showed difficulties in processing the instances of `cmz` (took up to 856 seconds to complete all those instances), but processed the remaining three families in less than a second. In particular, **bliss** spent from 10 to 856 seconds to solve 14 out of the 46 `cmz` instances, but managed to solve the rest in less than 10 seconds. In contrast, **saucy** solved all Miyazaki graphs in less than a second.



(a) Group order vs. #generators for **saucy** benchmarks

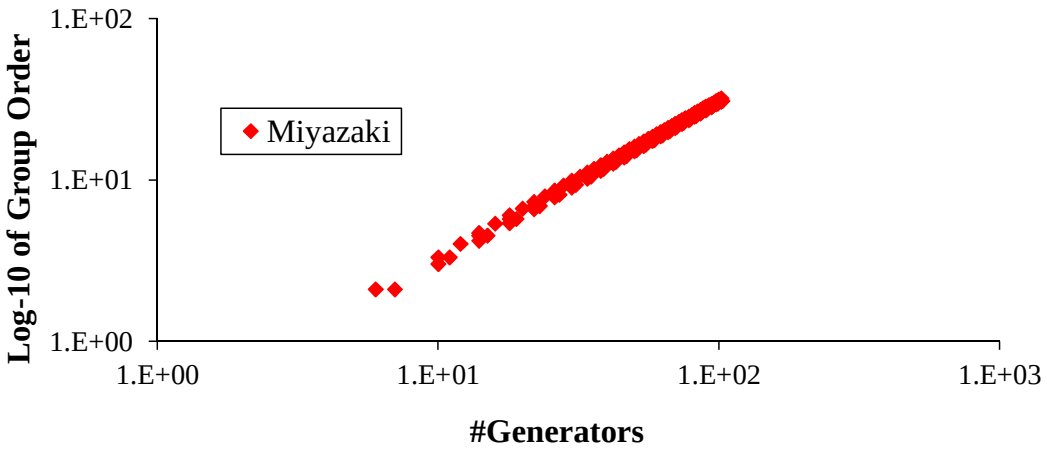


(b) Group Order vs. #generators for SAT 2011 CNFs



(c) Group Order vs. #generators for binary networks

Figure 7.3: This figure continues on the next page.



(d) Group Order vs. #generators for Miyazaki graphs

Figure 7.3: **saucy** 3.0 group order, as a function of number of generators for the (a) **saucy** benchmarks, (b) SAT 2011 CNFs, (c) binary networks, and (d) Miyazaki graphs.

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.

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 (287 Crafted and 254 Application) in less than 10 seconds, and 52 (9 Crafted and 43 Application) in 366 seconds, while **saucy** solved 577 (278 Crafted and 299 Application) in less than 10 seconds, and 6 (5 Crafted and 1 Application) in 300 seconds. Both **saucy** and **bliss** solved all Random benchmarks in less than a second. Overall, the results in Figure 7.4 indicate that **saucy** outperformed **bliss** on the majority of SAT

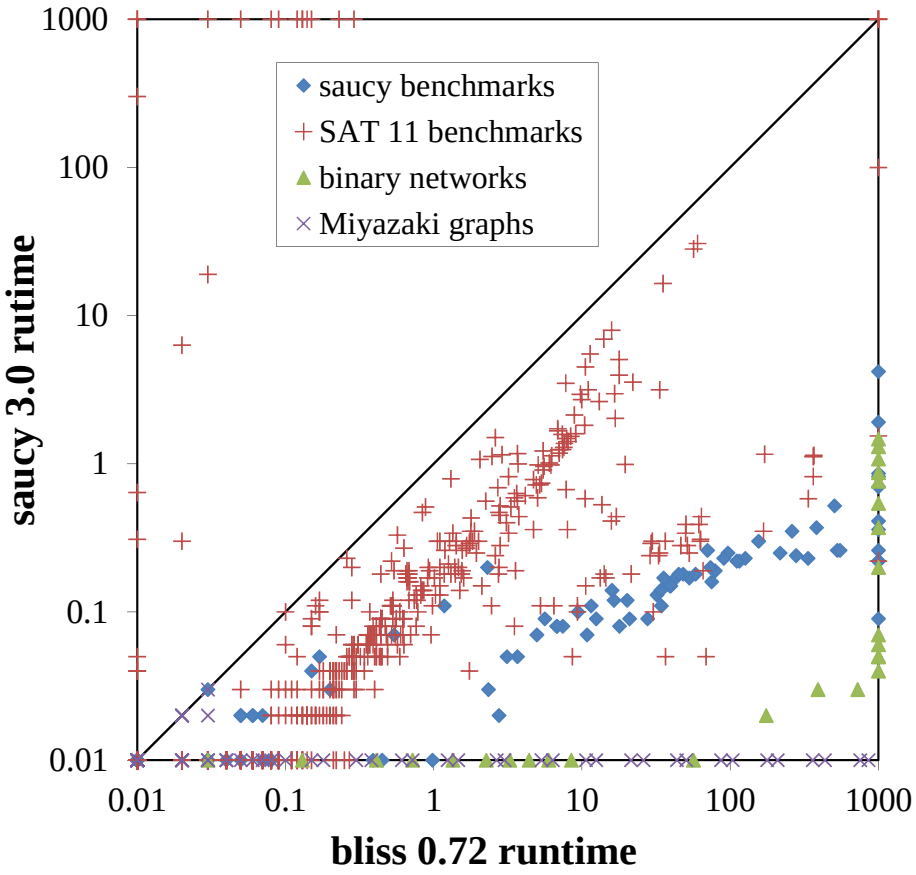


Figure 7.4: Runtime of **saucy** 3.0 versus **bliss** 0.72. A timeout of 1000 seconds was applied.

2011 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.

7.2 Empirical Results for Graph Canonical Labeling

We tested the performance of our proposed canonical-labeling approach on a subset of the benchmarks in Table 7.1 which were very large and sparse. This subset contained 432 benchmarks, which included all **saucy** benchmarks and binary net-

works, and the subset of the SAT 2011 competition CNFs [52] that had more than 10,000 variables (of the 1200 CNF instances, 313 had more than 10,000 variables). The choice of 10,000 was based on our observation that the modeled graphs for CNF benchmarks with that many variables tend to be very large and sparse.

All 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 to all experiments.

According to the results in Figure 7.1, 268 out of 432 large and sparse graphs exhibited non-trivial symmetry. These 268 benchmarks included all **saucy** graphs, all binary networks, and 149 out of 313 CNF instances. It should be mentioned that all the 268 graphs with at least one non-trivial symmetry were *highly symmetric*. Of those 268 graphs, 203 (75%) had group order of larger than 10^{10}.

To determine the amount of time that canonical-labeling algorithms spend on finding automorphisms, we ran **bliss** 0.72 [11] under two configurations; once, to just search for symmetries, and once, to also look for a canonical labeling. Figure 7.5 depicts the results. It can be seen that *the extra cost imposed by looking for a canonical labeling is negligible*. In other words, the canonical-labeling routines spend most of their time searching for symmetries.

To assess the performance of our proposed canonical-labeling approach versus state-of-the-art canonical labelers, we compared the results of our approach to that obtained from **bliss** 0.72 [11], **nauty** 2.4 (r2) [48], **nishe** 0.1 [49], and **traces** Nov09 [56]. Figure 7.6 depicts the results. These results clearly state that the combination of **saucy** and **bliss**, denoted by **saucy+bliss**, outperforms all the other four canonical labelers. Of the 432 total benchmarks, **saucy+bliss** solved 417, while **bliss** solved 404, **nauty** solved 58, **nishe** solved 130, and **traces** solved only 18. Furthermore, of the 432 benchmarks, 388 were solved by **saucy+bliss** in less than 10 seconds, while this number was reported to be 319 for **bliss**, 18 for **nauty**, 38 for **nishe**, and 7 for

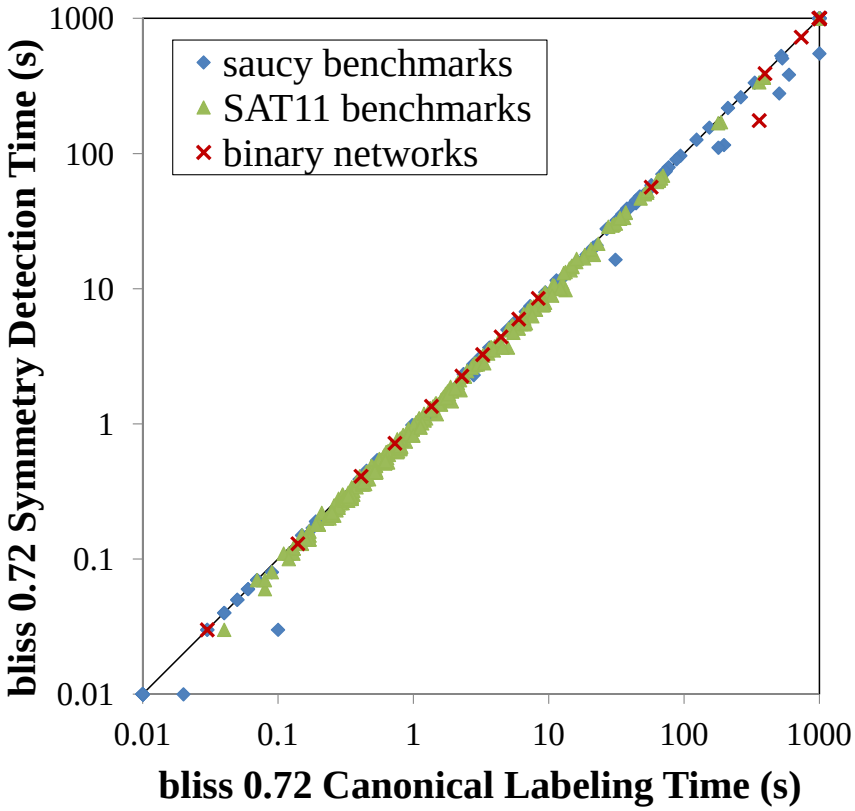
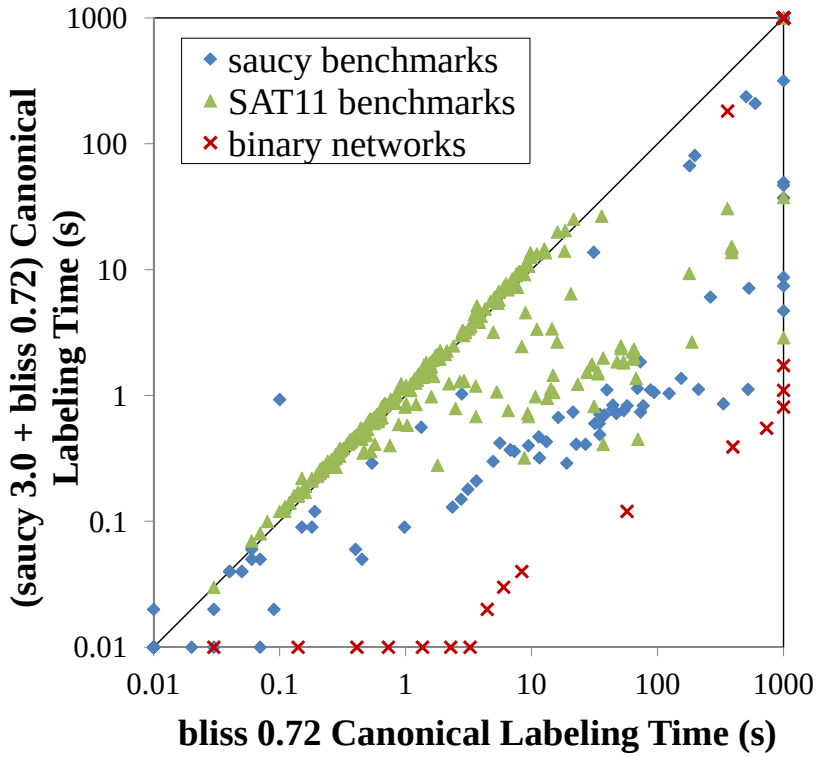


Figure 7.5: Runtime comparison of **bliss** symmetry detection vs. **bliss** canonical labeling.

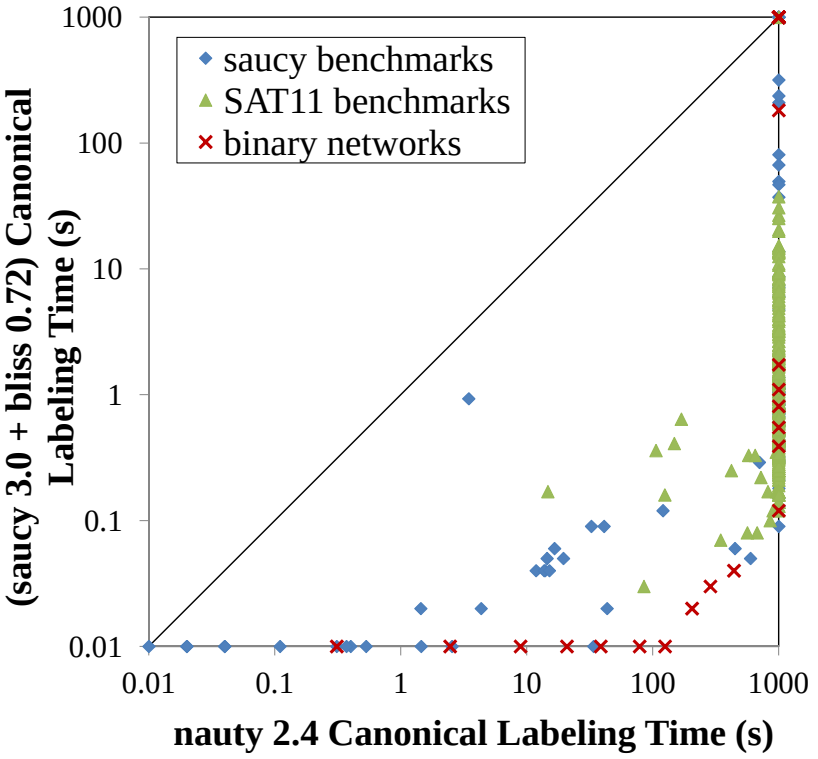
traces. Note that the 164 benchmarks that exhibited only one trivial symmetry did not benefit from our proposed canonical-labeling approach. Nevertheless, the extra overhead imposed by those benchmarks was insignificant, as they were all processed by **saucy** in less than 4 seconds.

The detailed comparison between **saucy**+**bliss** and each of the four mentioned canonical-labeling tools is provided below:

– Figure 7.6a compares the runtime of **saucy**+**bliss** to **bliss**. In total, **bliss** timed out on 28 benchmarks. (12 from the **saucy** suite, 3 from SAT11 benchmarks, and 13 from binary networks). Of those 28, **saucy**+**bliss** managed to solve 13. (8 **saucy** graphs, 2 SAT11 CNFs, and 3 binary networks). The remaining 404 benchmarks were solved by both **bliss** and **saucy**+**bliss**. Of those 404 benchmarks, 137 experienced a speed-up by **saucy**+**bliss**, whereas, 223 went

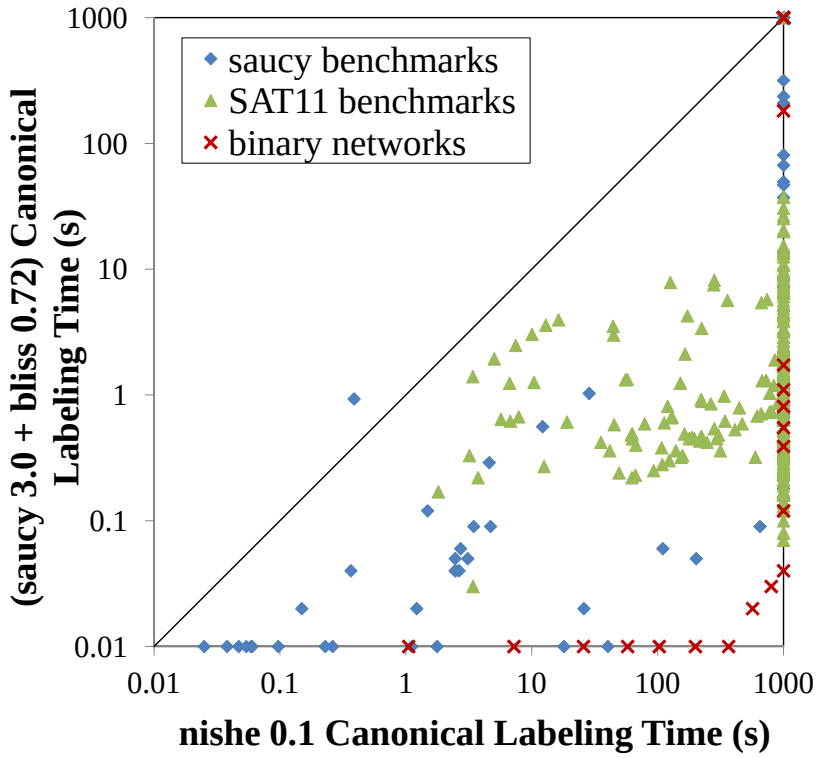


(a)

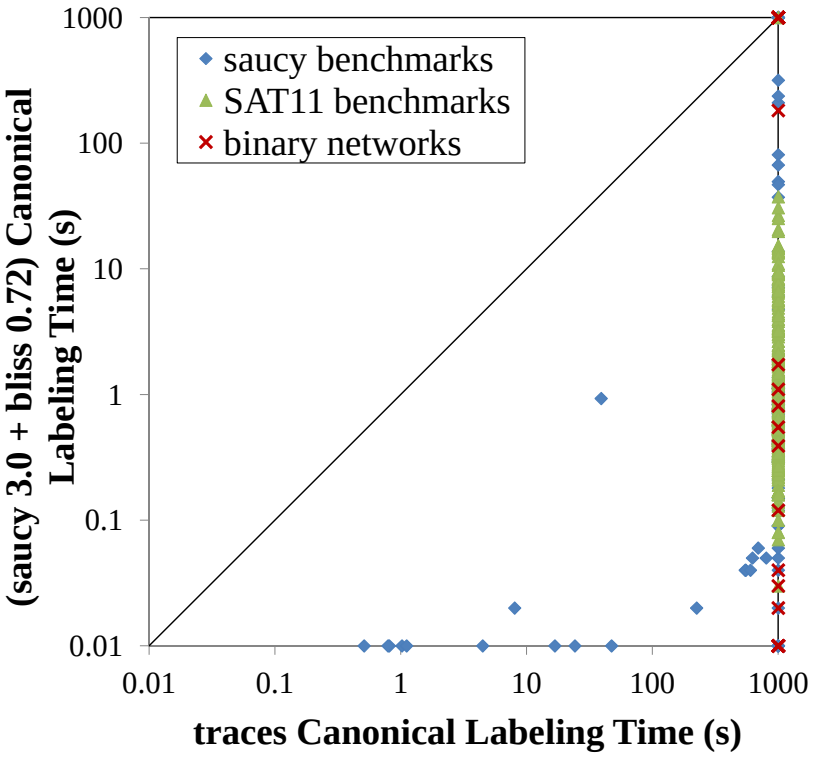


(b)

Figure 7.6: This figure continues on the next page.



(c)



(d)

Figure 7.6: Runtime comparison of our canonical-labeling approach (see Figure 5.2) vs. (a) **bliss** 0.72, (b) **nauty** 2.4 (r2), (c) **nishe** 0.1, and (d) **traces** Nov09.

through a slow-down (slow-down for 164 benchmarks with symmetry group of size 1 was expected). The highest speed-up was 1334x, which was reported for the binary network with 50,000 vertices (runtime was improved from 734 seconds to 0.55 seconds). The largest slow-down was 9x, which was reported for a **saucy** graph.

– Figure 7.6b compares the runtime of **saucy**+**bliss** to **nauty**. In total, **nauty** processed 58 benchmark (30 **saucy** graphs, 18 SAT11 CNFs, and 10 binary networks), but timed out or returned dynamic allocation failure on the remaining 374. All 58 benchmarks that were solved by **nauty** were also solved by **saucy**+**bliss**. The largest reported runtime from **nauty** for those benchmarks was 956 seconds. This was while **saucy**+**bliss** processed all those benchmarks in less than a second.

– Figure 7.6c compares the runtime of **saucy**+**bliss** to **nishe**. In total, **nishe** failed to process 302 benchmarks, on which it either timed out, or returned a segmentation fault. These benchmarks included 59 **saucy** graphs, 18 binary networks, and 225 SAT11 CNFs. Of these 302 benchmarks, **saucy**+**bliss** solved 287, but failed to process 4 **saucy** benchmarks, 10 binary networks, and one SAT11 CNF. All the benchmarks that were solved by **nishe** were also solved by **saucy**+**bliss**, but the runtimes of **saucy**+**bliss** were superior (speed-up of up to 36650x was reported). There was only one benchmark from the **saucy** suite which was processed by **saucy**+**bliss** in 0.93 seconds, but was completed by **nishe** in 0.38 seconds.

– Figure 7.6d compares the runtime of **saucy**+**bliss** to **traces**. Of the 432 benchmarks, **traces** only solved 18, all from the **saucy** suite. The poor performance of **traces** was due to the fact that it could not handle graphs with more than 18,000 vertices, and only 36 graphs in our suite (26 from **saucy** benchmarks,

and 10 from binary networks) exhibited less than 18,000 vertices. The 18 benchmarks that were processed by **traces** were also processed by **saucy+bliss**, but a speed-up of up to 16025x was observed in **saucy+bliss** runtimes.

In summary, the number of instances solved by each of the discussed canonical-labeling tools suggests the following ordering of performance: **saucy+bliss** > **bliss** > **nishe** > **nauty** > **traces**. This ordering is obtained by testing each tool on a considerable number of large and sparse graphs. However, such an ordering is subject to a change if graphs with fewer vertices and higher edge concentration are used for benchmarking.

7.3 Empirical Results for Finding Symmetries of Boolean Functions

We integrated our proposed symmetry-detection algorithms for Boolean functions in the ABC package [9]. We tested the performance of our algorithms on a collection of benchmarks from ISCAS'85 [14], ISCAS'89 [13], MCNC [58], and ITC'99 [25]. We also invoked our algorithms to solve several instances of Boolean matching.

Our experiments were conducted on an HP workstation equipped with a 3.2GHz Intel Quad-Core CPU, an 8MB cache and an 8GB RAM, running 64-bit Windows 7. We applied a time-out of 2000 seconds to all our experiments.

Table 7.2 demonstrates the runtime results. In this table, the first column lists the name of the benchmarks. The next three columns list the number of inputs, number of outputs, and the size of AIG for each benchmark, respectively. Column #Symms shows the order of symmetry group, and Column #Gen shows the number of generators. Information on constructed search trees, such as the number of nodes, number of levels, and number of conflicts, are drawn in Columns #Node, #Lev, and #Conf, respectively. The last column shows the runtimes in second.

Table 7.2: The results of our symmetry-detection algorithm for Boolean functions.

Circuit	#I	#O	|AIG|	#Symms	#Gen	#Node	#Lev	#Conf	Time (s)
mux-4*	6	1	19	2	1	3	2	0	0.01
mux-8	11	1	57	6	2	7	3	0	0.02
mux-16	20	1	158	24	3	13	4	0	0.05
mux-32	37	1	419	120	4	21	5	0	0.10
mux-64	70	1	1085	720	5	31	6	0	0.28
mux-128	135	1	2777	5040	6	43	7	0	1.10
mux-256	264	1	7044	40320	7	57	8	0	5.85
adder-1*	3	2	19	6	2	7	3	0	0.01
adder-16	33	17	144	196608	17	307	18	0	0.19
adder-40	81	41	760	3.298535E12	41	1723	42	0	1.63
b01	6	7	25	2	1	3	2	0	0.01
b02	4	5	20	1	0	1	1	0	0.01
b03	33	34	84	3.185050E7	15	273	17	1	0.15
b04	76	74	443	1.000000E7	0	1	1	0	0.11
b05	34	70	793	1.741824E7	12	211	15	2	0.20
b06	18	15	19	2880	7	57	8	0	0.02
b07	49	57	351	24	3	13	4	0	0.28
b08	29	25	154	1	0	3	2	1	0.03
b09	28	29	84	3.556874E14	16	273	17	0	0.15
b10	27	23	176	1	0	1	1	0	0.02
b11	37	37	610	1	0	1	1	0	0.04
b12	125	127	1002	960	7	57	8	0	1.08
b13	62	63	256	6	2	7	3	0	0.07
b14	276	299	6061	2.652529E32	29	871	30	0	7.08
b15	484	519	8384	2.652529E32	29	871	30	0	267
b17	1451	1512	27514	1.493036E98	90	8191	91	0	1843
b18	3357	3343	71878	9.802584E259	234	54991	235	0	1488
b20	521	512	12186	7.035908E64	58	3423	59	0	73
b21	521	512	12743	7.035908E64	58	3423	59	0	76

b22	766	757	18450	3.732589E97	88	7833	89	0	138
c499	41	32	400	384	4	64	5	37	0.30
c880	60	26	327	16	4	21	5	0	0.06
c5315	178	123	1773	5.662310E8	24	601	25	0	0.67
c7552	207	108	2074	1.460814E19	45	2376325	60	533198	1141
s953	45	52	347	2.585202E22	22	553	24	1	0.22
s1423	91	97	462	2	1	3	2	0	0.19
s5378	214	228	1389	1.431598E22	49	2551	51	1	2.28
s9234	247	250	1958	2.626993E29	50	83070	59	15541	370
s13207	700	790	2719	1.291078E213	294	86731	295	0	56
s15850	611	684	3560	3.759006E87	112	663078	114	4756	165
s38584	1464	1730	12400	8.200341E116	253	198045	255	2415	141
9symml	9	1	211	362880	8	73	9	0	0.12
apex6	135	99	659	2	1	7	3	1	0.07
cht	47	36	185	120	4	21	5	0	0.02
frg2	143	139	1164	240	5	43	7	1	0.13
i2	201	1	232	2.038573E222	180	42343	184	8985	3.83
i7	199	67	904	3.850825E66	62	3907	63	0	1.54
i10	275	224	1818	1658880	13	183	14	0	1.25
k2	45	45	1998	4	2	13	4	1	0.12
lal	26	19	109	768	5	31	6	0	0.05
pm1	16	13	47	864	7	57	8	0	0.06
rot	135	107	550	1658880	15	241	16	0	0.38
term1	34	10	311	480	6	43	7	0	0.11
vda	17	39	924	6	2	13	4	1	0.06
x1	51	35	377	8	3	13	4	0	0.04
x2	10	7	54	2	1	3	2	0	0.03
x3	135	99	833	2	1	3	2	0	0.03
x4	94	71	439	7257600	12	157	13	0	0.19

$*$ mux-n is an n-to-1 multiplexer, and adder-n is an n-bit ripple-carry adder with a carry in.

Table 7.3: The results of solving the PP-equivalence checking problem.

Circuit	#Node	#Lev	#Conf	Time (s)	Time (s) from [37]
mux-64	68	13	32	0.29	2.51
mux-128	4144	17	754	46	22
adder-16	141	36	36	0.11	0.05
adder-40	333	84	84	1.02	0.84
b05	161	26	86	0.22	0.19
b12	75	16	30	2.32	> 2000
b14	1057	62	874	12	10
b20	2096	119	1742	190	126
b21	2096	119	1742	210	145
s5378	272730	104	6785	173	1.45
s13207	11201	609	9377	78	> 2000
s15850	4893	232	4200	83	> 2000
s38584	5459	514	3504	188	> 2000
frg2	60	13	24	0.24	0.47
i10	166	30	79	3.25	2.20
rot	206	35	104	0.75	0.4

The symmetry group orders in Table 7.2 range from one trivial symmetry up to approximately 10^{260} symmetries. The largest group order was reported for b19, i.e., the largest benchmark in our collection. In our experiments, we observed that the number of symmetries of an n-to-1 multiplexer was reported to be $(log\, n)!$ This number corresponds to all permutations of the multiplexer's control signals.

The largest runtime in Table 7.2 is 1843 seconds (reported for b17). Of the 55 total benchmarks, only 3 took more than 1000 seconds to finish. The remaining were solved in less than 400 seconds (including s38584 which has more than a thousand I/Os). The least challenging benchmarks were those with less than a hundred I/Os, which were all processed in less than two seconds. All benchmarks with less than a

hundred I/Os were processed in less than two seconds. We also observed that b19 was not processed within the time-out limit. This is the reason why b19 is excluded from Table 7.2.

In our experiments, the majority of the benchmarks (76%) did not encounter any conflict. This suggests that our refinement techniques were effective enough to prune away unpromising branches of search. We also assessed the effect of learning by disabling it, and re-running our algorithm on all benchmarks that showed conflicts. We observed that, without learning, our algorithm failed to solve 61% of the benchmarks that showed conflicts.

As part of our study, we encoded several Boolean-matching benchmarks as symmetry-detection instances, and used our algorithm to solve them. Table 7.3 demonstrates the results, and compares the runtimes of our matcher to that proposed in [37]. All the results in Table 7.3 are averaged over 10 re-runs, where each re-run 1) randomly permuted I/Os of the circuits, and 2) reconstructed the circuits using ABC's synthesis commands to ensure structural difference.

Of the 16 benchmarks in Table 7.3, our Boolean matcher managed to solve all 16 in less than 210 seconds, but the matcher from [37] failed to process 4 within the time-out limit. On the other hand, the matcher from [37] solved one instance (s5378) in less than 2 seconds, but our matcher took 173 seconds to process it. For the remaining benchmarks, both matchers exhibited comparable results.

CHAPTER VIII

Conclusions and Future Work

In this chapter, we provide a summary of our work, and discuss conclusions. We also provide future directions for our work which can benefit from our previous work, and can have impact on both academic research and industrial applications.

8.1 Summary and Conclusions

In this thesis, we proposed scalable algorithms to detect symmetries of graphs. Our algorithms, referred to as **saucy**, find a set of generators for the symmetry group of a given graph, and return the size of its symmetry group. They accomplish this by refining the graph permutation space through nested partition refinement. They expedite the search by performing simultaneous partition refinement which allows conflict anticipation and early termination of futile subtrees. Furthermore, they incorporate four different pruning techniques; two algorithmic, namely, non-isomorphic OPP pruning and matching OPP pruning, and two group-theoretic, namely, coset pruning and orbit pruning. Our empirical results confirmed that **saucy** algorithms were scalable to graphs with million of vertices and edges.

Moreover, we proposed a new graph canonical-labeling approach that separated the search for symmetries from that for a canonical labeling. Our work was motivated by the observation that publications on graph automorphism and canonical labeling

typically focused on one of these problems and neglected the other. Canonical-labeling algorithms produce symmetries as a byproduct, but are not as efficient as graph-automorphism algorithms, which however, do not produce canonical labelings. We presented comparative analysis of relevant algorithms, highlighting the differences and exploring possible synergies. In particular, we showed that canonical-labeling algorithms can be more effective when symmetries are found in one dedicated pass and conveyed to these algorithms. We therefore developed an appropriate group-theoretic interface between **saucy** — the fastest symmetry finder — and **bliss** — the fastest canonical labeler. Extensive empirical results convincingly demonstrated the benefits of our approach.

Furthermore, we proposed new algorithms that searched for symmetries of Boolean functions under permutation of inputs and outputs. We used functional dependency, random simulation and satisfiability to facilitate the search. Specifically, we built a number of abstraction graphs that partially captured the functionality of the Boolean function, and used those graphs to prune unpromising branches of the search. Once refinement was exhausted, we invoked SAT to test candidate permutations for symmetries. In cases where SAT disproved functional equivalence under candidate permutations, we learned from SAT counterexamples to avoid similar conflicts in the future. As part of our study, we formulated instances of PP-equivalence checking as symmetry-detection problems, and invoked our algorithms to solve those instances. Empirical results confirmed the scalability of our proposed algorithms to combinational circuits with hundreds of I/Os.

8.2 Future Work

With the contributions of the current work, there are still unanswered questions that can give direction to our future research. Some promising directions include:

1. **Learning from conflicts that arise in the search for graph symmetries**: Our experimental results show a correlation between conflicts that arise during the search for graph symmetries. Specifically, they show that, in many cases, the number of constraint propagations before reaching a conflict is the same for different conflicts. By identifying such a correlation, one can develop a mechanism that helps avoid similar conflicts during the search.

2. **Devising an adaptive branching heuristic**: Branching heuristics significantly affect the performance of branching and backtracking algorithms, including the ones presented in this thesis. By examining the effect of different branching heuristics, one can develop a mechanism that chooses the "best" heuristic based on the properties of the combinatorial object in question. More ideally, one can devise an adaptive branching heuristic that analyzes conflicts and learns isomorphism invariants along the way during the search.

3. **Parallelizing graph symmetry-detection algorithms**: The sub-problems that arise in the search for graph symmetries are independent from each other. This suggests that we can parallelize the execution of such sub-problems, and gain instant speedup in runtime. The main challenge of such an algorithm is to identify the data structures that should be duplicated for each thread. It also needs to orbit prune (i.e., stop) the execution of the threads that lead to redundant generators based on the generators that are found by other threads.

4. **Detecting symmetries of Boolean functions under permutation and negation of I/Os**: Our proposed symmetry-detection algorithms for Boolean functions only allow permutation of I/Os. One can extend those algorithms to find symmetries under the permutation and negation of I/Os. Developing such algorithms can help solve the general Boolean matching problem [2], and reduce samples for logic simulation [59].

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] Giovanni Agosta, Francesco Bruschi, Gerardo Pelosi, and Donatella Sciuto. A unified approach to canonical form-based Boolean matching. In *Proceedings of the 44th annual Design Automation Conference*, pages 841–846, New York, NY, USA, 2007. ACM.

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

[4] 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.

[5] 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.

[6] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 271–282, Acapulco, Mexico, 2003.

[7] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

[8] László Babai, D. Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 310–324, New York, NY, USA, 1982. ACM.

[9] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. *http://www.eecs.berkeley.edu/ alanmi/abc*.

[10] binary networks. *https://sites.google.com/site/santofortunato/inthepress2*.

[11] bliss 0.72. *http://www.tcs.hut.fi/Software/bliss/bliss-0.72.zip*, 2011.

[12] Kellogg S. Booth and Charles J. Colbourn. Problems polynomially equivalent to graph isomorphism. Technical report, Computer Science Department, University of Waterloo, 1979.

[13] Franc Brglez, David Bryan, and Krzysztof Koiminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, pages 1929–1934, 1989.

[14] Franc Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *ISCAS*, pages 663–698, 1985. Special Session: "Recent Algorithms for Gate-Level ATPG with Fault Simulation and their Performance Assesment".

[15] Andrew E. Caldwell, Andrew B. Kahng, Andrew A. Kennings, and Igor L. Markov. Hypergraph partitioning for vlsi cad: Methodology for heuristic development, experimentation and reporting. In *in Proc. ACM/IEEE Design Automation Conf*, pages 349–354, 1999.

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

[17] Kaihui Chang, Igor L. Markov, and Valeria Bertacco. Post- placement rewiring by exhaustive search for functional symmetries. *ACM Trans. on Design Automation of Electronic Systems, Article*, 32:10–1145, 2007.

[18] Bill Cheswick, Hal Burch, and Steve Branigan. Mapping and visualizing the internet. In *USENIX Annual Technical Conference*, pages 1–13, 2000.

[19] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, 1996.

[20] 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.

[21] 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.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

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

[24] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[25] Electronic CAD and Reliability Group at Politecnico di Torino. ITC'99 benchmarks. *http://www.cad.polito.it/downloads/tools/itc99.html*.

[26] I. S. Filotti and Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 236–243, New York, NY, USA, 1980. ACM.

[27] A.A. Fraenkel and A. Lévy. *Abstract set theory*. Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1976.

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

[29] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint programming*. Elsevier, 2006.

[30] Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, pages 599–603, 1999.

[31] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 431–437, Menlo Park, CA, USA, 1998.

[32] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM*, pages 1371–1380, 2000.

[33] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 172–184, New York, NY, USA, 1974. ACM.

[34] 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.

[35] 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.

[36] Petteri Kaski. *http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml*.

[37] Hadi Katebi and Igor L. Markov. Large-scale Boolean matching. In Sunil P. Khatri and Kanupriya Gulati, editors, *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 771–776. Springer, 2011.

[38] Hadi Katebi, Karem A. Sakallah, and Igor 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.

[39] Victor N. Kravets and Karem A. Sakallah. Generalized symmetries in Boolean functions. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 526–532, San Jose, California, 2000.

[40] 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.

[41] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[42] Rudolf Mathon. A note on the graph isomorphism counting problem. *Inf. Process. Lett.*, 8(3):131–132, 1979.

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

[44] Gary Miller. Isomorphism testing for graphs of bounded genus. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 225–235, New York, NY, USA, 1980. ACM.

[45] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, UC Berkeley, 2005.

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

[47] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535, Las Vegas, 2001.

[48] nauty 2.4 (r2). *http://cs.anu.edu.au/ bdm/nauty/nauty24r2.tar.gz*.

[49] nishe 0.1. *http://gregtener.com/media/upload/nishe-0.1.tar.bz2*.

[50] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008.

[51] Jean-François Puget. On the satisfiability of symmetrical constrained satisfaction problems. *Lecture Notes in Computer Science*, 689:350–361, 1993.

[52] SAT-Competition. *http://www.satcompetition.org*.

[53] C. Scholl, D. Moller, P. Molitor, and R. Drechsler. BDD minimization using symmetries. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 18(2):81–100, November 2006.

[54] João P. Marques Silva and Karem A. Sakallah. GRASP—a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, 1996.

[55] Greg Tener and Narsingh Deo. Efficient isomorphism of miyazaki graphs. In *39th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, 2008.

[56] traces Nov09.
http://www.dsi.uniroma1.it/ piperno/pers/Download_files/TracesNov2009.zip.

[57] 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, 2001.

[58] Saeyang Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

[59] Chien-Chih Yu, Armin Alaghi, and John P. Hayes. Scalable sampling methodology for logic simulation: Reduced-ordered monte carlo. In *ICCAD*, 2012.

[60] Ellen W. Zegura, Kenneth L. Calvert, and Michael J. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Trans. Netw.*, 5(6):770–783, December 1997.

[61] Jin S. Zhang, Alan Mishchenko, Robert Brayton, and Malgorzata Chrzanowska-Jeske. Symmetry detection for large Boolean functions using circuit representation, simulation, and satisfiability. In *Design Automation Conference*, pages 510–515, 2006.