ATTACKS ON DIFFICULT INSTANCES OF GRAPH ISOMORPHISM:
SEQUENTIAL AND PARALLEL ALGORITHMS

by

GREG DANIEL TENER
B.S. University of Central Florida, 2005

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2009

Major Professor:
Narsingh Deo
ABSTRACT

The graph isomorphism problem has received a great deal of attention on both theoretical and practical fronts. However, a polynomial algorithm for the problem has yet to be found. Even so, the best of the existing algorithms perform well in practice; so well that it is challenging to find hard instances for them.

The most efficient algorithms, for determining if a pair of graphs are isomorphic, are based on the individualization-refinement paradigm, pioneered by Brendan McKay in 1981 with his algorithm nauty. Nauty and various improved descendants of nauty, such as bliss and saucy, solve the graph isomorphism problem by determining a canonical representative for each of the graphs. The graphs are isomorphic if and only if their canonical representatives are identical. These algorithms also detect the symmetries in a graph which are used to speed up the search for the canonical representative—an approach that performs well in practice. Yet, several families of graphs have been shown to exist which are hard for nauty-like algorithms. This dissertation investigates why these graph families pose difficulty for individualization-refinement algorithms and proposes several techniques for circumventing these limitations.

The first technique we propose addresses a fundamental problem pointed out by Miyazaki in 1993. He constructed a family of colored graphs which require exponential time for nauty (and nauty’s improved descendants). We analyze Miyazaki’s construction to determine the source of difficulty and identify a solution. We modify the base individualization-refinement algorithm by exploiting the symmetries discovered in a graph to guide the search for its canonical representative. This is accomplished with the help of a novel data structure called
a *guide tree*. As a consequence, colored Miyazaki graphs are processed in polynomial time—thus obviating the only known exponential upper-bound on individualization-refinement algorithms (which has stood for the last 16 years).

The preceding technique can only help if a graph has enough symmetry to exploit. It cannot be used for another family of hard graphs that have a high degree of regularity, but possess few actual symmetries. To handle these instances, we introduce an adaptive refinement method which utilizes the guide-tree data structure of the preceding technique to use a stronger vertex-invariant, but only when needed. We show that adaptive refinement is very effective, and it can result in dramatic speedups.

We then present a third technique ideally suited for large graphs with a preponderance of sparse symmetries. A method was devised by Darga et al. for dealing with these large and highly symmetric graphs, which can reduce runtime by an order of magnitude. We explain the method and show how to incorporate it into our algorithm.

Finally, we develop and implement a parallel algorithm for detecting the symmetries in, and finding a canonical representative of a graph. Our novel parallel algorithm divides the search for the symmetries and canonical representative among each processor, allowing for a high degree of scalability. The parallel algorithm is benchmarked on the hardest problem instances, and shown to be effective in subdividing the search space.
Dedicated to my parents
ACKNOWLEDGMENTS

I would like to thank my adviser, Narsingh Deo, for granting me the freedom to pursue
my interests while simultaneously keeping me focused enough to finish; for his patience in
opening my eyes to the art of scholarly writing and his faith that I could produce such
writing. The members of my committee, Ron Dutton, Charles Hughes and David Workman
all have my profound gratitude for providing feedback which helped polished and add rigor
to this dissertation. I would also like to thank Yue Zhao for being on my committee and for
introducing me to combinatorics and graph theory from a mathematician’s point of view as
an undergraduate.

Thanks to all of my friends for their support and understanding throughout the years.
In particular I’d like to thank Elena Erbiceanu for many valuable discussions and her help
in proofreading; Mark Colbert for his incredibly useful bits of assistance along the way;
Hamed Ahmadi, Chen-Fu Chang, Stephen Fulwider, Yiu Yu Ho, and Mahadevan Vasudevan
for being excellent labmates; Josh Cox, Hieuhanh Huynh, Matt Janisz, Kellyn Janisz, and
Fareza Yacoob for listening and asking questions.

I’d like to thank those I have learned from, a small subset of which is: Mr. White, for an
early foundation in mathematics; Scott Johnson, for the joy of programming; Michael Reid,
for the importance of precision; Arup Guha and Charles Hughes, for the beauty of computer
science; Robert Brigham, Narsingh Deo, and Yue Zhao, for the complexities of graph theory;
and Ron Dutton and Pawel Wocjan, for the elegance of algorithms.

Finally, I thank my parents, without whom these words would clearly be impossible; and
my brother, whose faith helped make them a reality.
# TABLE OF CONTENTS

LIST OF FIGURES ....................................................... xii

LIST OF TABLES ......................................................... xiv

LIST OF SYMBOLS ....................................................... xv

CHAPTER 1 INTRODUCTION ............................................. 1

1.1 Theory ................................................................. 4

1.1.1 Equivalent problems ............................................ 5

1.1.2 Related problems ................................................ 7

1.2 Applications ........................................................ 8

1.3 Isomorphism algorithms in practice ............................. 10

1.4 Organization ........................................................ 11

CHAPTER 2 PRELIMINARIES ............................................ 12

2.1 Partitions and colorings ............................................ 12

2.2 Group actions ...................................................... 15
### Chapter 4

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Guide trees</td>
<td>57</td>
</tr>
<tr>
<td>4.3.1</td>
<td>An example using Miyazaki graphs</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Pseudocode for a canonical labeling algorithm using a guide tree</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Correctness of target-cell reevaluation</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Group target-cell functions</td>
<td>65</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Group search-trees</td>
<td>67</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Approximating the automorphism group</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Lazy target-cell reevaluation</td>
<td>75</td>
</tr>
<tr>
<td>4.6.1</td>
<td>An example of lazy target-cell reevaluation</td>
<td>76</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Implementing lazy reevaluation</td>
<td>77</td>
</tr>
<tr>
<td>4.7</td>
<td>A good group target-cell function</td>
<td>78</td>
</tr>
<tr>
<td>4.8</td>
<td>Empirical study</td>
<td>79</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Experimental setup</td>
<td>80</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Benchmarking on Miyazaki graphs</td>
<td>80</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Benchmarking on non-Miyazaki graphs</td>
<td>88</td>
</tr>
</tbody>
</table>

### Chapter 5

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Taking advantage of node-invariants</td>
<td>92</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>Runtime considerations</td>
<td>99</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.4 A refinement-trace invariant</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>5.5 Empirical study</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td><strong>CHAPTER 6  SHORT-CIRCUITING GENERATOR DISCOVERY</strong></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>6.1 The motivation</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>6.2 Singleton permutations</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>6.3 Implementation using a guide tree</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td><strong>CHAPTER 7  A PARALLEL GRAPH ISOMORPHISM ALGORITHM</strong></td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>7.1 Design</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>7.2 Communication</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>7.2.1 Generator discovery</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>7.2.2 Canonical path updates</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>7.2.3 Load balancing</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>7.3 Empirical study</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>7.3.1 Observations</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td><strong>CHAPTER 8  CONCLUSION</strong></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>8.1 Future work</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>8.1.1 Group target-cell functions</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>8.1.2 Adaptive refinement</td>
<td>122</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1. Two isomorphic graphs. .............................................. 1

2. The graph in Figure 1(x) ............................................ 6

3. Three different views of a partition $\pi$. ............................ 14

4. A graph $G$ with four automorphisms. ............................... 18

5. The 6 isomorphs of $G$ in lexicographic order ....................... 20

6. Three isomorphic partitions. ........................................... 20

7. A graph $G$ on eight vertices with twelve edges and four automorphisms. ... 38

8. The search tree for the graph $G$ defined in Figure 7 ............. 39

9. The invariant values for the first and third leaves. .................... 40

10. The multigraph $Y_2$. .................................................. 47

11. The partitioned F"urer gadget $\Xi_3$. ................................. 49

12. The effect of applying F"urer gadgets to $Y_2$ to get $(\mathcal{X}(Y_2), \vartheta_2)$. .............. 49

13. Generators of $\text{Aut}(\mathcal{X}(Y_2), \vartheta_2)$. .......................... 50

14. The Miyazaki graph for $k = 2$ and three ordered colorings. .......... 51
15 The root node of \((\mathcal{X}(Y_2), \vartheta_{b,2})\) after discovering the first generator.  

16 The search tree for the graph defined in Figure 7, using min-orbits.  

17 Two different search trees for \((\mathcal{X}(Y_3), \vartheta_{b,3})\).  

18 The guide tree for the search tree in Figure 17(y).  

19 The search and guide trees using lazy reevaluation on \((\mathcal{X}(Y_3), \vartheta_{b,3})\).  

20 A complication with recovering the canonical labeling using lazy reevaluation.  

21 The runtimes and refines for nauty, bliss, and nishe on \((\mathcal{X}(Y_k), \vartheta_{c,k})\).  

22 The min and max runtimes and refines for nishe on random orderings of \(\vartheta_k\).  

23 The runtimes and refines using nauty, bliss, and nishe for mz-aug2-k.  

24 The smallest connected graph for which shape-invar has an effect.  

25 The search tree for the graph in Figure 24.  

26 The modified search tree for the graph in Figure 24.  

27 The guide tree for the search tree in Figure 26.  

28 A graph for illustrating generator discovery short circuiting.  

29 The search tree for the graph in Figure 28.  

30 The search tree for the graph in Figure 28 when short-circuiting generator discovery.  

LIST OF TABLES

1. Six problems polynomially equivalent to graph isomorphism. 7
2. Results for the $c$ coloring of Miyazaki graphs. 83
3. Runtimes for nishe on random orderings of $\vartheta_k$. 85
4. Refinements for nishe on random orderings of $\vartheta_k$. 85
5. Results for the $\texttt{mz-aug2-k}$ series. 87
6. Results comparing nishe to bliss and nauty on non-Miyazaki graphs. 88
7. Results for adaptive refinement. 103
8. Sequential results for the benchmark graphs using the target-cell function $\texttt{max-joins}$. 116
9. Parallel results for the selected Hadamard matrices. 117
10. Parallel results for the selected projective planes. 118
LIST OF SYMBOLS

$G$ a simple, undirected graph, page 2

$V$ the vertex set of a graph, page 2

$u, v$ vertices (elements of $V$), page 2

$E$ the edge set of a graph (unordered pairs $\{u, v\} \subseteq V$), page 2

$\gamma$ a bijection between two sets; usually a permutation of $V$, page 2

$\text{Aut}(G)$ the automorphism group of $G$, page 6

$\pi$ an ordered partition (or equivalently, coloring) of a set $V$, page 12

$W$ a cell in an ordered partition, page 12

$\mathbb{N}$ the natural numbers $\{1, 2, 3, \ldots\}$, page 14

$\text{indices}(\pi)$ the set of start indices when representing $\pi$ as an array, page 14

$\pi[k]$ the cell of $\pi$ with index $k$, page 14

$(G, \pi)$ a colored graph, page 15

$\mathcal{G}(V)$ the set of all graphs with vertex set $V$, page 15
$\Pi(V)$ the set of all colorings of the set $V$, page 15

$G(V) \times \Pi(V)$ the set of all colored graphs with vertex set $V$, page 15

$A$ a group (in the algebraic sense), page 15

$\Omega$ an arbitrary set of objects, page 15

$x^a$ the group action of $a \in A$ on $x \in \Omega$, page 15

$\text{Sym}(V)$ the group of all permutations of $V$, page 15

$\pi_1 \preceq \pi_2$ $\pi_1$ is finer than $\pi_2$, page 16

$\pi_1 \prec \pi_2$ $\pi_1$ is strictly finer than $\pi_2$, page 16

$\cong$ isomorphic to, page 17

$\text{Aut}(G, \pi)$ the color preserving automorphism group, page 18

$\Lambda$ a set, usually totally ordered, page 19

$\text{CF}(X)$ a function that is a canonical form, page 19

$\text{CL}(X)$ a function that is a canonical labeling, page 19

$\theta$ a subset of $\text{Aut}(G, \pi)$, usually a generating set, page 21

$R(G, \pi)$ a partition refiner (usually equitable), page 22

$\deg_G(v, W)$ the number of neighbors of $v$ that are in $W$, page 22
split(\(\pi, f\)) a function which sorts the cells of \(\pi\) using \(f\) as the comparator and splits where \(f\) values change, page 23

tc(\(G, \pi\)) a target-cell function, page 26

\(T(G, \pi)\) the search tree for the colored graph \((G, \pi)\), page 28

\(\mathcal{N}\) the node set in a search tree, page 28

\(\mathcal{E}\) the vertex-labeled edges in a search tree, page 28

\(r\) the root node of a search tree, page 28

\(\pi \vdash u\) the operation of individualizing \(u\) in \(\pi\), page 28

\(A \smallsetminus B\) the difference of the set \(A\) and \(B\), page 28

\(\nu\) a node in a search tree, page 28

\(\nu \xrightarrow{u} \nu_u\) \(\nu\)'s child when branching on \(u\) is \(\nu_u\), page 28

\(\mathcal{L}(G, \pi)\) the leaves of the search tree \(T(G, \pi)\), page 30

\(\lambda\) a leaf node in a search tree (and thus a permutation), page 30

\(I(G, \pi, \nu)\) a node invariant, page 31

\(\bar{I}(G, \pi, \nu_\ell)\) abbreviation for \((I(G, \pi, \nu_0), I(G, \pi, \nu_1), \ldots, I(G, \pi, \nu_\ell))\), page 33

\(\mathcal{L}_I(G, \pi)\) the incremental leaf-certificate values in \(T(G, \pi)\), page 33

\(\langle \theta \rangle\) the group generated by \(\theta\), page 35
Y\_k \quad \text{an undirected multigraph used to construct Miyazaki graphs, page 47}

Ξ\_n \quad \text{the F"urer gadget with parameter } n, \text{ replaces a vertex of degree } n, \text{ page 47}

\mathcal{X}(Y\_k) \quad \text{the Miyazaki graph with parameter } k, \text{ page 48}

\vartheta\_k \quad \text{the unordered partition of the vertices in } \mathcal{X}(Y\_k), \text{ page 48}

\vartheta\_a,\_k \quad \text{the } a\text{-coloring, orders the cells (cycle)}^{4k}(\text{internal})^{2k}(\text{bridge})^{2k}, \text{ page 51}

\vartheta\_b,\_k \quad \text{the } b\text{-coloring, orders the cells (bridge)}^{2k}(\text{cycle, cycle, internal})^{2k}, \text{ page 51}

\vartheta\_c,\_k \quad \text{the } c\text{-coloring, orders the cells (internal)}^{2k}(\text{bridge})^{2k}(\text{cycle})^{4k}, \text{ page 51}

\delta \quad \text{an element of } \text{Sym}(V), \text{ usually not an automorphism, page 52}

\delta^{-1}\text{Aut}(G, \pi)\delta \quad \text{abbreviation for } \{\delta^{-1}\gamma\delta : \gamma \in \text{Aut}(G, \pi)\}, \text{ page 53}

tc(G, \pi, A) \quad \text{a group target-cell function, page 65}

A\_\pi \quad \text{the partition stabilizer of } \pi \text{ in the group } A, \text{ page 67}

\mathcal{T}(G, \pi, A) \quad \text{a group search tree, page 68}

\mathcal{I}_I(G, \pi, \nu, u) \quad \text{a vertex-invariant defined by } I(G, \pi, R(G, \nu \vdash u)), \text{ page 94}
CHAPTER 1
INTRODUCTION

The graph isomorphism problem asks a seemingly simple question: Are two graphs structurally equivalent? In other words, the names of the vertices are unimportant. What matters is how they are connected.

Figure 1: Two isomorphic graphs. An isomorphism from (x) to (y) is $a \mapsto 2, b \mapsto 3, c \mapsto 4, d \mapsto 5, e \mapsto 6, f \mapsto 1$.

Figure 1 shows that this question is not as simple as it seems; the two graphs are isomorphic, yet not obviously so. The graph in Figure 1(x) is drawn without crossing edges (it is planar) and has a clear symmetry about the vertical axis. The graph in Figure 1(y) is drawn with three edge-crossings and one of the symmetries (swapping vertex 6 with vertex 2 and keeping the others fixed) is easier to visualize. Despite the differences in presentation, they are essentially the same object. The graph-isomorphism problem is nontrivial, and a formal definition helps to understand it better.
**Definition (graph).** A graph is a pair $G = (V, E)$ where $V$ is the finite set of vertices (with an assumed total ordering $v_1, v_2, \ldots v_n$) and the edge set $E$ consists of unordered pairs of vertices.

**Example.** The six vertices of the graph in Figure 1(x) are $a$, $b$, $c$, $d$, $e$, and $f$ and its ten edges are $\{a,b\}$, $\{a,d\}$, $\{a,f\}$, $\{b,c\}$, $\{b,e\}$, $\{c,d\}$, $\{c,e\}$, $\{d,e\}$ and $\{e,f\}$.

**Definition (graph isomorphism).** An isomorphism between two graphs is a one-to-one and onto function $\gamma$ from the vertices of a graph $G$ to the vertices of a graph $H$ such that $\gamma$ preserves adjacency (and nonadjacency). That is, $\{u,v\}$ is an edge in $G$ if and only if $\{\gamma(u), \gamma(v)\}$ is an edge in $H$. If such an isomorphism exists then $G$ and $H$ are said to be isomorphic.

**Example.** One isomorphism between the graphs in Figure 1 is

$$\gamma = \begin{pmatrix} a & b & c & d & e & f \\ 2 & 3 & 4 & 5 & 6 & 1 \end{pmatrix}.$$

**Definition (graph isomorphism problem).** Given two graphs $G$ and $H$, are they isomorphic?

Graph isomorphism is a highly-studied problem in computer science. It is strongly suspected not to be NP-complete, yet no polynomial-time algorithm for it has been found despite much effort.

There are two main approaches to solving this problem. One approach, called exact matching, searches for the isomorphism between the two graphs. The other approach, called canonical labeling, takes one graph and transforms it into a canonical representative of all
graphs isomorphic to it, and does the same for the other graph. If (and only if) the two canonical representatives are identical, then the graphs are isomorphic.

Exact-matching algorithms have the advantage that they are usually easier to implement and understand than canonical-labeling algorithms. In some cases they are adaptable to solving the subgraph-isomorphism problem: see [20] and [83] for example.

Canonical-labeling algorithms tend to be more complicated and difficult to implement than exact-matching algorithms. Although they are not easily adaptable to solving subgraph isomorphism, they have some advantages. For instance, once a canonical representative of a graph is found, it can be reused for future isomorphism tests against this graph. In addition to computing a canonical representative, most modern general-purpose canonical-labeling algorithms also output the symmetries in a graph. These symmetries have a wide variety of applications as will be seen in Section 1.2.

The most efficient canonical-labeling/symmetry-detecting algorithms are based on the individualization-refinement method pioneered by McKay’s nauty [62, 64]. Despite the success of these algorithms (saucy by Darga, Skallah, and Markov [23, 24] and bliss by Junttila and Kaski [44] are two other examples) there still exist graphs for which they perform poorly. These algorithms are the basis for the state of the art approaches introduced in this dissertation used to combat these difficult graphs.
1.1 Theory

The graph isomorphism problem occupies a special position in complexity theory. It has resisted all attempts to be classified into any of the standard complexity classes. Because of this, it is placed in its own complexity class, called *graph isomorphism complete*. It is one of the few “natural” problems that is a candidate for being in NP, but not in NP-complete or P [35]. Two other problems which were thought to occupy this position, linear programming and determining if a number is prime or not, have been shown to be in P. Linear programming was shown to be in P by Khachian in 1979 and determining primality was shown to be in P by Agrawal, Kayal, and Saxena [49, 4] in 2002.

Several subproblems of graph isomorphism are known to have polynomial algorithms. These include, but are not limited to: graphs of bounded genus [67] (which includes planar graphs [41] and trees [47]), graphs with bounded valence [59, 68] (valence is a synonym for degree), graphs with bounded eigenvalue multiplicity [9], interval graphs [13], permutation graphs [19], and graphs of bounded color-class size [11]. A survey of the current status of several subproblems and other problems related to graph isomorphism can be found in [51].

For the general problem, the best upper-bound as given by Luks and Zemlyachenko [11] is $\exp(\sqrt{cn \log n})$ where $c > 0$. The bound was obtained via a canonical-labeling algorithm (unrelated to *nauty*). The graph isomorphism problem has been described as a “disease” by Read and Corneil in an early survey paper on the problem [75, 36]. It was observed that the ease of creating seemingly-strong graph invariants combined with the property that most
randomly generated instances will be easy for even naive solutions contributes to premature publishing and the disease-like nature of the problem.

### 1.1.1 Equivalent problems

Several problems are polynomially equivalent to graph isomorphism. These problems are called *isomorphism complete*. These include, but are not limited to: isomorphism of directed acyclic graphs, regular graphs, line graphs, chordal graphs, semigroups, finite automata, and regular self-complementary graphs. More comprehensive treatments of isomorphism-complete problems can be found in [12] by Kellogg and [87] by Zemlyachenko.

The primary purpose of some individualization-refinement algorithms is to produce symmetries (notably *saucy*). Producing the canonical representative is usually optional, and secondary to producing the symmetries. This is no coincidence, as many isomorphism-complete problems involve the symmetries of a graph.

**Definition (symmetry).** A symmetry (formally called an *automorphism*) of a graph is an isomorphism from a graph to itself. Given a graph $G = (V, E)$, a permutation $\gamma: V \rightarrow V$ which is an isomorphism from $G$ to $G$ is an automorphism of $G$.

**Example.** The automorphisms of the graph in Figure 2, are

1. $\gamma_1 = \begin{pmatrix} a & b & c & d & e & f \\ a & d & c & b & e & f \end{pmatrix}$ which swaps $b$ with $d$, 

2. \( \gamma_2 = \begin{pmatrix} a & b & c & d & e & f \\ e & b & c & d & a & f \end{pmatrix} \) which swaps \( a \) with \( e \),

3. the composition of \( \gamma_1 \) and \( \gamma_2 \) which swaps \( a \) with \( e \) and \( b \) with \( d \),

4. and the identity permutation which maps each vertex to itself.

The set of all automorphisms of a graph \( G \) is called its automorphism group, denoted \( \text{Aut}(G) \) (the group operation being function composition). The automorphism group partitions the vertices of \( G \) into a set of equivalence classes called the automorphism partition via the equivalence \( u \sim v \) if and only if there is a symmetry in \( \text{Aut}(G) \) which moves \( u \) to \( v \).

**Example.** The automorphism partition of the graph in Figure 2 is

\[
\{\{a, e\}, \{b, d\}, \{c\}, \{f\}\}.
\]

In [61], Mathon showed the equivalence of the problems in Table 1. Of particular note is that ICOUNT is equivalent to ISO, a result also found in [8]. No NP-complete problem is known to be equivalent to its counting counterpart. This is good evidence that the graph isomorphism problem is not NP-complete. More evidence lies in the fact that if graph
isomorphism is NP-complete then the polynomial hierarchy would collapse to the second level, which is unlikely [14, 77].

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO($G_1, G_2$)</td>
<td>are $G_1$ and $G_2$ isomorphic?</td>
</tr>
<tr>
<td>IMAP($G_1, G_2$)</td>
<td>an isomorphism from $G_1$ to $G_2$ if it exists</td>
</tr>
<tr>
<td>ICOUNT($G_1, G_2$)</td>
<td>number of isomorphisms from $G_1$ to $G_2$</td>
</tr>
<tr>
<td>ACOUNT($G$)</td>
<td>$</td>
</tr>
<tr>
<td>AGEN($G$)</td>
<td>generators of $\text{Aut}(G)$</td>
</tr>
<tr>
<td>APART($G$)</td>
<td>automorphism partition of $G$.</td>
</tr>
</tbody>
</table>

Table 1: Six problems polynomially equivalent to graph isomorphism.

1.1.2 Related problems

Definition (graph automorphism problem). Does a graph $G$ have a nontrivial automorphism?

Determining if a graph has a nontrivial automorphism is believed to be easier than determining its full automorphism group [81]. A graph with only one automorphism (the identity permutation) is called rigid. Almost all graphs (even when restricted to regular graphs [54]) are rigid [10]. This was shown by proving that almost every graph can be canonically labeled in linear time. It was also shown that almost all $d$-regular graphs on $n$ vertices for $3 \leq d \leq n - 4$ are rigid [50].

Canonically labeling a graph certainly helps with determining isomorphism, but does determining isomorphism help with canonically labeling a graph? The following observation was made by Babai and Luks in [11]:

7
If, as is almost always the case, \( \text{Aut}(G) \) is trivial, the number of such representations is \( n! \). How do we select [the canonical representative]?

It seems that canonically labeling a graph is more difficult than determining its automorphism group, while determining if a graph is rigid is assumed to be an easier problem than graph isomorphism [81, 51].

### 1.2 Applications

A graph isomorphism algorithm can be used to test for isomorphism between any two explicitly presented algebraic or combinatorial structures (i.e. not presented via generators) since they can be encoded as graphs [39, 66]. This means graph isomorphism can be used to determine isomorphism of objects like Hadamard matrices, projective planes, Latin squares, combinatorial designs, and hypergraphs; but it cannot be used to determine isomorphism of groups specified via generators or vector spaces specified via a basis. Canonical-labeling algorithms can be used to store a database of canonical representatives which can be indexed and searched. Symmetry-detection algorithms help avoid enormous amounts of redundant work, making completely hitherto intractable problems tractable. A small sampling of graph isomorphism applications is: helping robots recognize objects [3], optimal routing for networks [25], image analysis [29], isomorphism of molecules (for chemistry) [31, 32, 74], and data mining [86].
In recent years, much of the research in graph isomorphism algorithms has focused on how to efficiently detect symmetries in large, sparse graphs. Two recent algorithms, bliss [44] and saucy [23, 24], have overtaken nauty as the symmetry-detectors of choice for large graphs. Some symmetry-detection applications include: circuit diagnosis and layout correction [18], model checking and relational specifications [28, 82, 43], satisfiability solving for electronic design automation [23, 60], search problems [78], determining the entropy of complex networks [85], and solving linear integer-programming [55].

The automorphism group of an object is also an indispensable tool for combinatorial mathematics and is used to create [21, 57, 56, 65, 70], enumerate [30, 63, 48], and classify [46, 5, 76, 45], combinatorial objects.

As a specific example of the great utility of symmetry-detection for industrial applications, recent advances in boolean-satisfiability solving take advantage of symmetries in conjunctive normal form formula to achieve exponential speedup on practical benchmark problems. This is accomplished by determining the symmetries in a conjunctive normal form formula, appending some symmetry breaking predicates to the formula and then using a conventional backtracking satisfiability solver on the augmented formula (see [6] and [7] by Aloul et. al. for details). Many of the benchmark conjunctive normal form formulas arising from practical instances of satisfiability are very large and related to circuit and program-run verification. The tool saucy was created specifically to find symmetries in these large instances.
For an example from the realm of combinatorics, *nauty* was the basis of the first program to generate all nonisomorphic graphs with 11 vertices. The theory behind McKay’s generation method can be found in [63]. A recent dissertation by Al-Azemi [5] discusses how to exhaustively generate graphs, digraphs, and linear spaces with given properties (e.g. regularity, girth, etc.) with the aid of canonical-labeling algorithms.

### 1.3 Isomorphism algorithms in practice

Several large software packages make use of isomorphism algorithms. Some systems currently employing exact matching algorithms are:

1. BOOST [1] – a large collection of open source portable C++ libraries, uses the algorithm of Deo, Davis, and Lord described in [27].

2. Combinatorica [71] – a combinatorics package for Mathematica [84].

3. NetworkX [38] – an open source python package for the “creation, manipulation, and study of the structure, dynamics, and functions of complex networks.”, uses *vf2* by Cordella et al. [20].

Some systems currently using canonical-labeling algorithms are:

1. GRAPE [79] – a GAP (Groups, Algorithms, Programming [34]) package for computing with graphs and groups, uses *nauty*. 

10
2. igraph [22] – a C/C++ library (with bindings for R and Python) for complex network research, uses bliss [44].


1.4 Organization

The rest of the dissertation is organized as follows: Chapter 2 provides the background necessary to present the basic canonical-labeling algorithm and Chapter 3 presents the basic canonical-labeling algorithm. Chapter 4 describes the exponential upper-bound for nauty (colored Miyazaki graphs) and demonstrates how to avoid it efficiently. Chapter 5 introduces an adaptive refinement operation which can help alleviate the difficulty posed by highly-regular graphs with few automorphisms. Chapter 6 shows how to take advantage of sparse symmetries (as introduced in [24]) to speed up the search when a graph has many symmetries. Chapter 7 describes the design and implementation of a parallel canonical labeling algorithm and Chapter 8 summarizes this dissertation and discusses avenues for future work.
CHAPTER 2
PRELIMINARIES

The concepts required for describing the canonical-labeling algorithm come from graph theory and group theory. For a background on graph theory concepts, see [26, 37]. The group theory required does not go very deep; the following books are recommended for an introduction to group theory [42, 40].

2.1 Partitions and colorings

Definition (ordered partition of a set $V$). An ordered partition $\pi$ of a set $V$ is an ordered sequence of nonempty disjoint subsets $\pi = (W_1, W_2, \ldots, W_k)$ whose union is $V$. The subset $W_i \subseteq V$ is called the $i^{th}$ cell of $\pi$. Without qualification the term partition means an ordered one.

Example. Let $V = \{a, b, c, d, e, f\}$ and $\pi = (\{b\}, \{a, d, e\}, \{c, f\})$. The cells of $\pi$ union to $V$ and are disjoint, therefore $\pi$ is an ordered partition of $V$.

A partition $\pi$ of the set $V$ may be regarded as a coloring of $V$ which assigns the same color to two vertices if and only if they are in the same cell. The same variable, $\pi$, will be used to denote both partitions and colorings because the concepts are essentially the same.
**Definition** (coloring of a set). A *coloring* of a set $V$ is a function $\pi : V \rightarrow \{1, 2, \ldots, k\}$ which assigns each $v \in V$ a “color”. Here $k$ is known as the number of *color classes* of $\pi$, and $\pi$ is called a $k$–coloring of $V$.

A $k$–coloring, $\pi$, partitions the set $V$ into a sequence of $k$ disjoint subsets $W_1, W_2, \ldots, W_k$ such that each vertex in $W_i$ is assigned color $i$ by $\pi$, for $1 \leq i \leq k$. The coloring $\pi$ corresponds to the partition $(W_1, W_2, \ldots, W_k)$ of $V$.

**Example.** The ordered partition of $V = \{a, b, c, d, e, f\}$, $\pi = (\{b\}, \{a, d, e\}, \{c, f\})$, corresponds to the coloring $\pi : V \rightarrow \{1, 2, 3\}$ defined by

$$
\pi = \begin{pmatrix}
a & b & c & d & e & f \\
2 & 1 & 3 & 2 & 2 & 3
\end{pmatrix}.
$$

**Remark.** Instead of coloring $V = \{v_1, v_2, \ldots, v_n\}$ with the first $k$ natural numbers, it is convenient to use $\{v_1, v_2, \ldots, v_k\}$ as the set of colors. This has the useful property that when $k = n$, the coloring is a permutation of $V$. This meaning of the word coloring is used henceforth.

**Example.** The ordered partition of $V = \{a, b, c, d, e, f\}$, $\pi = (\{b\}, \{a, d, e\}, \{c, f\})$, now corresponds to the function $\pi : V \rightarrow \{a, b, c\}$ defined by

$$
\pi = \begin{pmatrix}
a & b & c & d & e & f \\
b & a & c & b & b & c
\end{pmatrix}.
$$
As a third and final representation, a partition can also be expressed as a zero-indexed array of length $|V|$, and a set of indices which contains the start index of each cell (see Figure 3). The starting index of each cell is the sum of the sizes of all previous cells.

**Definition (indices of a partition).** The set of indices of a partition $\pi$ is formally defined as

$$\text{indices}(\pi) = \left\{ j \in \mathbb{N} \mid j = \sum_{i=1}^{m} |W_i| \text{ for } 0 \leq m \leq k - 1 \right\}.$$

**Notation.** To retrieve a cell based on its index $j \in \text{indices}(\pi) = \{j_1, j_2, \ldots, j_k\}$, let $\pi[j_i] = W_i$, where $j_i$ is the index of the $i^{th}$ cell of $\pi$.

**Example.** Let $V = \{a, b, c, d, e, f\}$ and $\pi : V \to \{a, b, c\}$ be the 3-coloring defined in Figure 3. Its indices are $\{0, 1, 4\}$ and its cells are $\pi[0] = \{b\}$, $\pi[1] = \{a, d, e\}$, $\pi[4] = \{c, f\}$. The cell $\{b\}$ has size one and is called *trivial*. The cells $\{a, d, e\}$ and $\{c, f\}$ have size greater than one and are called *nontrivial*.

$$\begin{align*}
\pi(b) &= a \\
\pi(a) &= \pi(d) = \pi(e) = b \\
\pi(c) &= \pi(f) = c
\end{align*}$$

Figure 3: Three different views of a partition $\pi$: functional (left), partition (middle) and array based (right). The indices of $\pi$ are $\text{indices}(\pi) = \{0, 1, 4\}$.

The terms “partition” and “coloring” are used interchangeably throughout.

**Definition (colored graph).** Given a graph $G = (V, E)$ and a coloring $\pi$ of $V$, the pair $(G, \pi)$ is called a *colored graph*. 

14
Notation. The set of all graphs with vertex set $V$ is denoted $\mathcal{G}(V)$. The set of all colorings of a set $V$ is denoted $\Pi(V)$. The set of all colored graphs with vertex set $V$ is then $\mathcal{G}(V) \times \Pi(V)$, the cartesian product of $\mathcal{G}(V)$ and $\Pi(V)$.

2.2 Group actions

Definition (group action). The action of a group $A$ on a set $\Omega$ is a function (expressed using exponent notation) from $A \times \Omega$ to $\Omega$ such that for all $x \in \Omega$:

(i) $(x^a)^b = x^{(ab)}$ for all $a, b \in A$, and

(ii) $x^e = x$ where $e$ is the identity element of $A$.

Example. Let $G = (V, E)$, and $\text{Sym}(V)$ be the group of all permutations of $V$. The group actions of $\text{Sym}(V)$ on (i) vertices, (ii) edges, (iii) subsets of vertices, (iv) graphs, (v) colorings, and (vi) colored graphs are defined as follows:

(i) $v^\gamma = \gamma(v)$, for $v \in V$

(ii) $\{u, v\}^\gamma = \{u^\gamma, v^\gamma\}$, for $\{u, v\} \in E$

(iii) $W^\gamma = \{u^\gamma : u \in W\}$, where $W \subseteq V$

(iv) $G^\gamma = (V^\gamma, E^\gamma) = (V, E^\gamma)$, for $G \in \mathcal{G}(V)$

(v) $\pi^\gamma = (W_1^\gamma, W_2^\gamma, \ldots, W_k^\gamma)$, where $\pi \in \Pi(V)$ and $\pi = (W_1, W_2, \ldots, W_k)$
(vi) \((G, \pi)^\gamma = (G^\gamma, \pi^\gamma)\), for \((G, \pi) \in \mathcal{G}(V) \times \Pi(V)\)

In general, the group action of \(\text{Sym}(V)\) on a structure is applied to each component in the structure recursively, until elements of \(V\) are reached.

**Definition** (discrete partition). A partition \(\pi\) for which each cell is trivial is called **discrete**.

Because partitions are colorings, and colorings use the first \(k = n\) elements of \(V\) as colors, \(\pi \in \text{Sym}(V)\).

**Example.** If \(V = \{a, b, c, d, e, f\}\) and

\[
\pi = \begin{array}{cccc}
b & d & e & a \\
f & c & d & e \\
a & b & c & d \\
\end{array}, \text{ then } \\
\pi = \left( \begin{array}{cccc}
b & d & e & a \\
a & b & c & d \\
f & c & e & d \\
\end{array} \right). \\
\]

In disjoint cycle notation, \(\pi = (a \ d \ b)(c \ f \ e)\). Observe that the action of any discrete partition \(\pi\) on itself is the identity permutation. For example,

\[
\begin{array}{cccc}
b & d & e & a \\
f & c & d & e \\
a & b & c & d \\
\end{array} \pi = \begin{array}{cccc}
b^\pi & d^\pi & e^\pi & a^\pi \\
f^\pi & c^\pi & d^\pi & e^\pi \\
a^\pi & b^\pi & c^\pi & d^\pi \\
\end{array} = \begin{array}{cccc}
a & b & c & d \\
f & c & e & d \\
b & d & e & a \\
\end{array}. \\
\]

**Definition** (finer, \(\preceq\)). Given two partitions \(\pi_1\) and \(\pi_2\) of a set \(V\), \(\pi_1\) is **finer** than \(\pi_2\) \((\pi_1 \preceq \pi_2)\) if \(\pi_1\) can be formed by splitting zero or more cells of \(\pi_2\). Formally, \(\pi_1 \preceq \pi_2\) if and only if for all \(u, v \in V\), \(\pi_2(u) \leq \pi_2(v)\) implies \(\pi_1(u) \leq \pi_1(v)\). If \(\pi_2(u) < \pi_2(v)\) implies \(\pi_1(u) < \pi_1(v)\)
then $\pi_1$ is said to be strictly finer than $\pi_2$ ($\pi_1 \prec \pi_2$). For example,

\[
\begin{array}{cccc}
    b & d & e & a \\
    \end{array}
\prec
\begin{array}{cccc}
    b & a & d & e \\
    \end{array}
\prec
\begin{array}{cccc}
    a & b & c & d & e & f \\
    \end{array}
\]

### 2.3 Isomorphisms, automorphisms and orbits

Let the group $A$ act on a set $\Omega$. The concept of isomorphism can be generalized (and later applied to search trees and search nodes).

**Definition** (general isomorphism). Given objects $X, Y \in \Omega$, $X$ is isomorphic to $Y$ ($X \cong Y$) if there exists a $\gamma \in A$ (called an isomorphism) such that $X^\gamma = Y$. If $X \cong Y$ then $Y$ is said to be an isomorph of $X$. Of particular interest is the case when $A = \text{Sym}(V)$ and $\Omega = \mathcal{G}(V)$; the graph isomorphism problem.

**Definition** (automorphism). An automorphism of an object is an isomorphism from an object to itself. The automorphism group of a graph $G \in \mathcal{G}(V)$ is

\[
\text{Aut}(G) = \{ \gamma \in \text{Sym}(V) : G^\gamma = G \}.
\]

**Definition** (color preserving automorphism group). Given a partition $\pi \in \Pi(V)$, the color preserving automorphism group is

\[
\text{Aut}(G, \pi) = \{ \gamma \in \text{Sym}(V) : (G, \pi)^\gamma = (G, \pi) \}.
\]
Definition (orbit of a vertex). The orbit of a vertex $u \in V$ in a colored graph $(G, \pi)$ is the set $\{u^\gamma : \gamma \in \text{Aut}(G, \pi)\}$. This is also known as the orbit of $u$ in $\text{Aut}(G, \pi)$.

Definition (orbits of a graph). The orbits of a graph are the orbits of its vertices under the action of the automorphism group.

Example. Consider the graph $G$ in Figure 4. It has four automorphisms, and the orbits of its vertices under the automorphism group are $\{\{a, c\}, \{b, d\}\}$.

```
Figure 4: A graph $G$ with edge set $E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}, \{d, b\}\}$ and the four automorphisms ( ), (a c), (b d), (a c)(b d).
```

2.4 Invariants, canonical forms and canonical labelings

Let the group $A$ act on a set $\Omega$, and $\Lambda$ be any set.
**Definition** (isomorphism invariant). A function $I: \Omega \rightarrow \Lambda$ is called an (isomorphism) *invariant* if it is constant on all isomorphic objects, so that $X \cong Y$ implies $I(X) = I(Y)$ for all $X, Y \in \Omega$. Equivalently, for all $\gamma \in A$, $I(X) = I(X^\gamma)$.

**Definition** (isomorphism certificate). If it also holds that $I(X) = I(Y)$ implies $X \cong Y$ then $I$ is called an (isomorphism) *certificate*. Equivalently, if $I(X) = I(Y)$ then there exists a $\gamma \in A$ such that $Y = X^\gamma$.

**Definition** (canonical form). A certificate whose range and domain are the same set ($\Lambda = \Omega$) such that $I(X) \cong X$ for all $X \in \Omega$ is called a *canonical form*.

**Definition** (canonical labeling). If $CF: \Omega \rightarrow \Omega$ is a canonical form then for all $X \in \Omega$ there exists a $\gamma \in A$ such that $CF(X) = X^\gamma$. A function $CL: \Omega \rightarrow A$ which given an object $X$ returns a corresponding $\gamma$ is called a *canonical labeling*. Equivalently, a function $CL$ is a canonical labeling when the function that maps an object $X$ to $X^{CL(X)}$ is a canonical form.

**Example** (a canonical form for graphs). In the case of canonizing graphs, $\Omega = \Lambda = G(V)$ and $CL: G(V) \rightarrow Sym(V)$. As an example of a canonical form, take the minimum graph of all those in its isomorphism class: if graphs are ordered by first taking the adjacency matrix, concatenating its rows together from top to bottom and treating this as an $n^2$-bit binary number, then the minimum graph is the one corresponding to the smallest $n^2$-bit binary number. Figure 5 illustrates this concept for the graph $G$ in Figure 4.
Figure 5: The $4!/4 = 6$ isomorphs of $G$ in lexicographic order. Using the canonical form 
“isomorph with smallest adjacency matrix” would return the labeling $(b\;c)$.

**Example** (a canonical form for colored graphs). An analogous example applies to colored graphs. The set of colorings, $\Pi(V)$, is ordered lexicographically. Figure 6 illustrates the concept of minimum element in the set $\Pi(V)$.

![Diagrams](image.png)

Figure 6: Three isomorphic partitions. The minimum is the middle partition, because all of its elements are ordered lexicographically.

The minimum element of an isomorphism class is only an illustrative example of a canonical form and is seldom used in practice. The terms canonical form and canonical labeling are functions, but will commonly be used to refer to the output of the function.
CHAPTER 3
CANONICAL-LABELING ALGORITHM

This chapter gives a high-level description of the ideas behind the individualization-refinement technique pioneered by *nauty* which computes a canonical labeling and generators of the automorphism group of a colored graph. The coloring is user-supplied (any coloring is allowed) and is often the unit partition, which colors each vertex the same.

The algorithm takes as input a colored graph \((G, \pi) \in G(V) \times \Pi(V)\) and outputs a permutation \(\gamma \in \text{Sym}(V)\) such that \((G, \pi)^\gamma\) is a canonical representative of its isomorphism class. It also outputs a set \(\theta \subseteq \text{Sym}(V)\) which generators \(\text{Aut}(G, \pi)\) (i.e. each element of \(\text{Aut}(G, \pi)\) can be expressed in terms of elements of \(\theta\)). The description of the algorithm is a combination of the explanations found in [62, 52, 69], although it most closely follows the description given by Junttila and Kaski used to present *bliss* [44].

The algorithm performs a depth-first search of a tree where each node is a partition of the vertex set \(V\). Each child of a node is formed by choosing a vertex in a nontrivial cell, placing it in its own cell (called *individualizing* the vertex), then refining the resultant partition. Thus, each child is strictly finer than its parent. Eventually this process will yield discrete partitions, which are the leaves of the search tree. Each leaf corresponds to a possible canonical labeling. The leaves are ordered by a function called a node invariant, and a leaf whose node invariant is minimum is returned by the algorithm as the canonical labeling.
If two leaves are found to have the same invariant value, a generator of the automorphism group is discovered and used to prune the search tree.

The fundamental operation used to create each node in the search tree is called refinement, and is introduced in the next section.

### 3.1 Partition refinement

Distinguishing between vertices which cannot be in the same orbit is a fundamental operation of many isomorphism algorithms. This is accomplished via a refinement function \( R: \mathcal{G}(V) \times \Pi(V) \rightarrow \Pi(V) \) which splits the cells of its input based on some vertex invariant. A common choice for the invariant uses vertex neighborhoods.

**Definition** (neighborhood of a vertex). Given a graph \( G \in \mathcal{G}(V) \), the neighborhood of a vertex \( u \in V \) is the set of all vertices adjacent to it, or \( N(u) = \{ v : \{u, v\} \in E \} \).

**Definition** (degree of a vertex within a set). Given a graph \( G \in \mathcal{G}(V) \), the degree of a vertex \( u \in V \) in a set \( W \subseteq V \) is the number of neighbors of \( u \) contained in \( W \), or \( \deg_G(u, W) = |N(u) \cap W| \).

**Definition** (equitable partition). A partition \( \pi \) is equitable with respect to a graph \( G \) if and only if for every pair of vertices \( u \) and \( v \) in the same cell and for every cell \( W \) of \( \pi \), \( \deg_G(u, W) = \deg_G(v, W) \).
**Definition** (equitable refiner). An equitable refiner takes in a colored graph \((G, \pi)\) and outputs an equitable partition with respect to \(G\) which is finer than \(\pi\) (or as fine as).

An equitable refiner is very powerful. For instance, in [10], Babai and Kučera showed that for almost every graph \(G\) and unit coloring \(\pi\) (such that each vertex is colored the same), the weakest equitable refiner discretizes \(\pi\).

Pseudocode for an equitable refiner is given in Algorithm 1. It makes use of a function \(\text{split}(\pi, f)\), where \(\pi\) is a partition of \(V\) and \(f\) is a function which assigns to each vertex \(u \in V\) its degree into a set \(W \subseteq V\). The set \(W\) changes depending on the current active cell. The splitting function sorts the contents of each cell in \(\pi\) using the output of \(f\) as the key for comparing, then splits the cell where its contents differ with respect to \(f\). Thus, if \(f(u) < f(v)\), then in the output of \(\text{split}(\pi, f)\) the index of the cell containing \(u\) is smaller than the index of the cell containing \(v\).

**Algorithm 1** An equitable refinement function \(R\)

**Input:** colored graph \((G, \pi)\)

**Output:** equitable partition \(R(G, \pi)\)

\[f : V \rightarrow \mathbb{N}\]

active = indices(\(\pi\))

while (active \(\neq \emptyset\)) do

\[k = \min(\text{active})\]
\[\text{active} = \text{active} \setminus \{k\}\]

\[f(u) = \deg_G(u, \pi[k])\] for all \(u \in V\)
\[\pi' = \text{split}(\pi, f)\]

\[\text{active} = \text{active} \cup (\text{indices}(\pi') \setminus \text{indices}(\pi))\]

\[\pi = \pi'\]

return \(\pi\)
Let $n = |V|$. Observe that the while-loop in Algorithm 1 can be executed at most $n - 1$ times, because there are only $n - 1$ indices of a partition $\pi$ of $V$. Also, given that $\deg_G(u, W)$ can be computed in an amount of time proportional to $W$, the amount of time required to sort and split the cells based on an active index is $O(n \log n)$. Hence, the asymptotic runtime of Algorithm 1 is $O(n^2 \log n)$. In practice, its runtime fluctuates depending on the input colored graph. Indeed, for almost all graphs and colorings $\pi$ which color each vertex the same, Algorithm 1 returns a discrete partition after three rounds of the while-loop [10].

The initial set of active indices is only the whole set of indices for the root node. For all other nodes in the search tree, it is the index returned by the target cell function. For most of the examples given in this dissertation, the set of initial active indices will always be the whole set of indices. An example of a refiner stronger than Algorithm 1 will be given in Chapter 5.

### 3.1.1 Properties of refiners

The goal of a refiner is to split vertices which are not in the same orbit. The weakest refiner, $R(G, \pi) = \pi$, just returns the partition passed in without changing it, which is not very useful. The strongest refiner splits all vertices which are not in the same orbit, which is not very practical, since doing this in an efficient manner implies being able to solve the general canonical-labeling problem efficiently (it produces an ordering of the automorphism partition).
In general, a refiner must have the following properties:

(i) \( R(G, \pi) \preceq \pi \) and

(ii) \( R(G, \pi)^\gamma = R(G^\gamma, \pi^\gamma) \) for all \( \gamma \in \text{Sym}(V) \).

Property (i) enforces the idea that cells are split, never joined. Property (ii) ensures that \( R \) is independent of the vertex labeling. A consequence of Property (ii) is that vertices which are in the same orbit are not separated by the refiner.

**Observation.** Property (ii) of refiners ensures that if vertices \( u \) and \( v \) are in the same orbit of \( \text{Aut}(G, \pi) \) then \( u \) and \( v \) are in the same cell of \( R(G, \pi) \).

**Proof.** The proof is by contradiction. Suppose that there exist vertices \( u \) and \( v \) in the same orbit of \( \text{Aut}(G, \pi) \) and that \( u \) and \( v \) are in different cells of \( R(G, \pi) \). Let \( \gamma \in \text{Aut}(G, \pi) \) be a permutation which moves \( u \) to \( v \) (\( \gamma \) exists because \( u \) and \( v \) are in the same orbit). Since \( \gamma \) is an automorphism, \( R(G^\gamma, \pi^\gamma) = R(G, \pi) \). By Property (ii) of refines, \( R(G^\gamma, \pi^\gamma) = R(G, \pi)^\gamma \), but \( R(G, \pi) \neq R(G, \pi)^\gamma \) because \( u \) is not in the same cell of both \( R(G, \pi) \) and \( R(G, \pi)^\gamma \). This contradicts the assumption that \( R \) is a refiner. \( \square \)

**Definition** (refiner). Any function \( R: \mathcal{G}(V) \times \Pi(V) \to \Pi(V) \) satisfying properties (i) and (ii) is a refiner.

The refinement operation presented here is designed for undirected simple graphs. However, the concept of equitability can be extended to apply to directed graphs, graphs with weighted edges, graphs with weighted vertices, etc. This is accomplished by generalizing the
concept of degree. For instance, in a directed graph, the edges are directed (and called arcs)
so that \((u, v) \in E\) means that there exists an arc from \(u\) to \(v\). The degree of a vertex \(u \in V\)
in a set \(W \subseteq V\) is defined to be the triple \((d_{in}, d_{out}, d_{both})\) where

\[
d_{in} = |\{v \in W : (v, u) \in E \text{ and } (u, v) \notin E\}|,
\]
\[
d_{out} = |\{v \in W : (v, u) \notin E \text{ and } (u, v) \in E\}|, \text{ and}
\]
\[
d_{both} = |\{v \in W : (v, u) \in E \text{ and } (u, v) \in E\}|.
\]

### 3.2 Target cell selection

After refining a partition, if it is not discrete, then a nontrivial cell (index) is chosen from
which to individualize vertices. This is accomplished via a target cell function \(tc: \mathcal{G}(V) \times \Pi(V) \rightarrow \{1, \ldots, n\}\). A target-cell function takes as input a graph \(G\) and a nondiscrete
partition \(\pi\). It outputs an element of \(\text{indices}(\pi)\). If \(k = tc(G, \pi)\), then \(k\) must have the
following properties:

(i) \(k \in \text{indices}(\pi)\)

(ii) \(|\pi[k]| \geq 2\)

(iii) \(tc(G^\gamma, \pi^\gamma) = k = tc(G, \pi)\) for all \(\gamma \in \text{Sym}(V)\).
Properties (i) and (ii) ensure that \( tc(G, \pi) \) returns the index of a nontrivial cell. Property (iii) ensures that \( tc(G, \pi) \) is independent of the vertex labeling. The term index is commonly used to mean the cell with that index.

**Definition (target-cell function).** Any function \( tc: \mathcal{G}(V) \times \Pi(V) \to \{1, \ldots, n\} \) satisfying the above three properties is called a target cell function.

The choice for the target-cell function can vastly affect the number of nodes traversed in the search tree and thus the runtime of the algorithm, as demonstrated in [80]. A simple target-cell function is:

\[
\text{first-smallest}(G, \pi) = \text{the first smallest nontrivial index of } \pi.
\]

In early versions of nauty the target-cell function was first-smallest. However, the target-cell function for the current version of nauty (at version 2.47b [64]) is more complex and is described next.

**Definition (nontrivially joined cells).** Given a colored graph \((G, \pi)\) and cells \(U\) and \(W\) in \(\pi\), \(W\) is nontrivially joined to \(U\) if for all \(w \in W\), \(0 < \deg_G(w, U) < |U|\). If \(\pi\) is equitable, then it suffices to check only a single element \(w \in W\) to determine if \(W\) is nontrivially joined to \(U\).

The target-cell function used by current versions of nauty, bliss, and saucy is

\[
\text{max-joins}(G, \pi) = \text{the first nontrivial index that is nontrivially joined to the most cells}.
\]
The target-cell function \texttt{max-joins} attempts to choose the cell which will split the most other cells after individualizing an element of this cell and refining.

### 3.3 The search tree

Let $T(G, \pi) = (\mathcal{N}, \mathcal{E}, r)$ be a labeled rooted tree where $\mathcal{N} \subseteq \Pi(V)$ is the node set, $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times V$ is the vertex-labeled edge set and $r \in \mathcal{N}$ is the root. Given a partition $\pi$ and a vertex $u$, the notation $\pi \models u$ denotes the operation of \textit{individualizing} $u$. That is, if $u$ is in a nontrivial cell $W$ of $\pi = (\ldots, W, \ldots)$, then $\pi \models u = (\ldots, \{u\}, W \setminus \{u\}, \ldots)$.

**Definition** (search tree of a colored graph). Given a refiner $R$ and a target-cell function $\texttt{tc}$, the search tree is defined inductively as follows:

1. $r = R(G, \pi) \in \mathcal{N}$

2. If $\nu \in \mathcal{N}$ and $\nu$ is discrete, then $\nu$ is a leaf. Otherwise, let $k = \texttt{tc}(G, \nu)$. If $u \in \nu[k]$ and $\nu_u = R(G, \nu \models u)$ then $\nu_u \in \mathcal{N}$ and $(\nu, \nu_u, u) \in \mathcal{E}$.

The notation $\nu \xrightarrow{u} \nu_u$ means that $(\nu, \nu_u, u) \in \mathcal{E}$. The sequence of vertices which label the edges on the path from the root to a node $\nu$ is called its \textit{fixed path}. The \textit{level} of a node is the number of elements on its fixed path. For instance, let $\nu$ be a node at level $m$ with fixed path $u_1, u_2, \ldots, u_m$. Then there exists nodes $\nu_0, \nu_1, \ldots, \nu_{m-1}$ ($\nu_0 = r$) such that

$$
\nu_0 \xrightarrow{u_1} \nu_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{m-2}} \nu_{m-1} \xrightarrow{u_m} \nu.
$$
Definition (group action on search trees). The action of $\gamma \in \text{Sym}(V)$ on the search tree is $T(G, \pi)^\gamma = (N^\gamma, E^\gamma, r^\gamma)$.

An important effect of this definition is that $T(G, \pi)^\gamma = T(G^\gamma, \pi^\gamma)$, [62, 44], which is proven here for completeness. The proof uses the following lemma:

Lemma 1. A node $\nu$ with fixed path $u_1, u_2, \ldots, u_m$ where $m \geq 0$ is in $T(G, \pi)$ if and only if $\nu^\gamma$ is a node in $T(G^\gamma, \pi^\gamma)$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma$ for all $\gamma \in \text{Sym}(V)$.

Proof. This is shown by induction on $m$, the fixed path length of $\nu$. Observe that $R(G, \pi)^\gamma = R(G^\gamma, \pi^\gamma) = r^\gamma$ by Property (ii) of refiners, so the root nodes of $T(G, \pi)^\gamma$ and $T(G^\gamma, \pi^\gamma)$ are equal and the base case of $m = 0$ holds.

Assume that $m \geq 1$ and let $\nu_{m-1}$ be the parent of $\nu$. By hypothesis, $\nu_{m-1}$ is a node in $T(G, \pi)$ with fixed path $u_1, u_2, \ldots, u_{m-1}$ if and only if $\nu_{m-1}^\gamma$ is a node in $T(G^\gamma, \pi^\gamma)$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_{m-1}^\gamma$. The definition of the search tree will be used to show that $\nu_{m-1} \xrightarrow{u_m} \nu$ is in $T(G, \pi)$ if and only if $\nu_{m-1}^\gamma \xrightarrow{u_m^\gamma} \nu^\gamma$ is in $T(G^\gamma, \pi^\gamma)$.

Let $k = \text{tc}(G, \nu_{m-1})$, then by Property (iii) of target-cell functions, $k = \text{tc}(G^\gamma, \nu_{m-1}^\gamma)$. Therefore, $u_m \in \nu_{m-1}[k]$ if and only if $u_m^\gamma \in \nu_{m-1}^\gamma[k]$ by the defined group action on partitions. This means $\nu_{m-1}$ has a child $R(G, \nu_{m-1} \triangleright u_m) = \nu$ if and only if $\nu_{m-1}^\gamma$ has a child $R(G^\gamma, \nu_{m-1} \triangleright u_m^\gamma)$. Since $\nu_{m-1} \triangleright u_m^\gamma = (\nu_{m-1} \triangleright u_m)^\gamma$ by the definition of individualization, $R(G^\gamma, \nu_{m-1} \triangleright u_m^\gamma) = R(G^\gamma, (\nu_{m-1} \triangleright u_m)^\gamma)$. Therefore, by Property (ii) of refiners, $R(G^\gamma, (\nu_{m-1} \triangleright u_m)^\gamma) = R(G, \nu_{m-1} \triangleright u_m)^\gamma = \nu^\gamma$. Thus, $\nu_{m-1} \xrightarrow{u_m} \nu$ is in $T(G, \pi)$ if and only if $\nu_{m-1}^\gamma \xrightarrow{u_m^\gamma} \nu^\gamma$ is in $T(G^\gamma, \pi^\gamma)$, proving the lemma. \qed
**Theorem 1** (McKay [62]). Given a colored graph \((G, \pi)\), a refiner \(R\), and a target-cell function \(tc\), \(T(G, \pi)^\gamma = T(G^\gamma, \pi^\gamma)\). That is, two colored graphs are isomorphic if and only if their search trees are isomorphic.

**Proof.** Observe that \(\nu\) is a node in \(T(G, \pi)\) with fixed path \(u_1, u_2, \ldots, u_m\) where \(m \geq 0\) if and only if \(\nu^\gamma\) is a node in \(T(G, \pi)^\gamma\) with fixed path \(u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma\). This follows directly by the defined group action on search trees. Using Lemma 1, \(\nu^\gamma\) is a node in \(T(G, \pi)^\gamma\) with fixed path \(u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma\) if and only if \(\nu^\gamma\) is a node in \(T(G^\gamma, \pi^\gamma)\) with fixed path \(u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma\). Therefore the edges, nodes, and root node of \(T(G, \pi)^\gamma\) and \(T(G^\gamma, \pi^\gamma)\) coincide and thus \(T(G, \pi)^\gamma = T(G^\gamma, \pi^\gamma)\).

**Corollary 1.** A node \(\nu\) with fixed path \(u_1, u_2, \ldots, u_m\) is in \(T(G, \pi)\) if and only if the node \(\nu^\gamma\) with fixed path \(u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma\) is in \(T(G, \pi)\) for all \(\gamma \in \text{Aut}(G, \pi)\).

**Notation.** The set of leaf nodes of the tree \(T(G, \pi)\) is denoted \(\mathcal{L}(G, \pi)\). The variable \(\lambda\) is used to denote leaf nodes.

**Corollary 2.** Define an equivalence relation over \(\mathcal{L}(G, \pi)\) by \(\lambda_1 \sim \lambda_2\) if and only if there exists a \(\gamma \in \text{Aut}(G, \pi)\) such that \(\lambda_1^\gamma = \lambda_2\). Since the only permutation which fixes a discrete partition is the identity, the size of each equivalence class is \(|\text{Aut}(G, \pi)|\).

Properties (i) and (ii) of target-cell functions and Property (i) of refiners ensure that the search tree is finite.
3.4 The canonical labeling

**Definition** (node invariant). Given a colored graph \((G, \pi)\), a node \(\nu\) in \(T(G, \pi)\) and a totally ordered set, a *node invariant* is an invariant \(I(G, \pi, \nu)\) which outputs an element of the totally ordered set.

**Definition** (leaf certificate). A node invariant is called a *leaf certificate* if its restriction to the leaf nodes is a certificate; i.e. if \(\lambda_1\) and \(\lambda_2\) are leaves in \(T(G, \pi)\), then \(I(G_1, \pi_1, \lambda_1) = I(G_2, \pi_2, \lambda_2)\) if and only if \((G_1, \pi_1, \lambda_1) \cong (G_2, \pi_2, \lambda_2)\).

**Example.** Let the totally ordered set be \(\mathcal{G}(V) \times \Pi(V) \cup \{0\}\) and define \((G, \pi) < 0\) for all \((G, \pi) \in \mathcal{G}(V) \times \Pi(V)\). A simple, but weak, leaf certificate can be defined as follows:

\[
\text{weakest-invar}(G, \pi, \lambda) = \begin{cases} 
(G, \pi)^\lambda & \text{if } \lambda \text{ is discrete, and} \\
0 & \text{otherwise.} 
\end{cases}
\]

The invariant *weakest-invar* is demonstrated to be a leaf certificate in [44]. This is proven here for completeness.

**Proposition 1** (Junntila and Kaski [44]). The node-invariant *weakest-invar* is a leaf certificate.

*Proof.* It needs to be shown that *weakest-invar* satisfies the requirements of being a certificate:

(i) \(\text{weakest-invar}(G, \pi, \lambda) = \text{weakest-invar}(G^{\gamma}, \pi^{\gamma}, \lambda^{\gamma})\) for all \(\gamma \in \text{Sym}(V)\), and
(ii) weakest-invar\((G_1, \pi_1, \lambda_1) = weakest-invar\((G_2, \pi_2, \lambda_2)\) implies \((G_1, \pi_1, \lambda_1) \cong (G_2, \pi_2, \lambda_2)\).  

To help show Property (i), observe that the action of a permutation \(\gamma\) on a partition \(\lambda\) is \(\lambda^\gamma = (W_1^\gamma, \ldots, W_k^\gamma)\), so as a function,

\[
\lambda^\gamma(u) = \lambda(u^{\gamma^{-1}}) = u^{\gamma^{-1}\lambda}. \tag{3.1}
\]

If \(\lambda\) is discrete and therefore in Sym\((V)\), then \(\lambda^\gamma = \gamma^{-1}\lambda\). Property (i) follows because

\[
\text{weakest-invar}(G^\gamma, \pi^\gamma, \lambda^\gamma) = (G^\gamma, \pi^\gamma)^{\lambda^\gamma}
\]

\[
= (G^{\gamma\lambda^\gamma}, \pi^{\gamma\lambda^\gamma})
\]

\[
= (G^{\gamma\gamma^{-1}\lambda}, \pi^{\gamma\gamma^{-1}\lambda}) \text{ by (3.1)}
\]

\[
= (G^\lambda, \pi^\lambda)
\]

\[
= \text{weakest-invar}(G, \pi, \lambda).
\]

Property (ii) holds because

\[
\text{weakest-invar}(G_1, \pi_1, \lambda_1) = \text{weakest-invar}(G_2, \pi_2, \lambda_2)\]

means that

\[
(G_1, \pi_1)^{\lambda_1} = (G_2, \pi_2)^{\lambda_2}, \text{ which implies}
\]

\[
(G_1, \pi_1)^{\lambda_1\lambda_2^{-1}} = (G_2, \pi_2), \text{ and since by (3.1),}
\]

\[
\lambda_1^{\lambda_1\lambda_2^{-1}} = (\lambda_1\lambda_2^{-1})^{-1}\lambda_1 = \lambda_2\lambda_1^{-1}\lambda_1 = \lambda_2, \text{ it follows that}
\]

\[
(G_1, \pi_1, \lambda_1) \cong (G_2, \pi_2, \lambda_2) \text{ via the isomorphism } \lambda_1\lambda_2^{-1}.
\]
Now that the existence of a leaf certificate has been shown, a canonical labeling based on the search tree can be developed.

**Definition** (incremental leaf certificate). Let $\nu_0$ be the root node of $T(G, \pi)$ and $\nu_\ell$ be a node at level $\ell$ whose path from the root node is $\nu_0 \xrightarrow{u_1} \nu_1 \xrightarrow{u_2} \cdots \xrightarrow{u_\ell} \nu_\ell$. Let $I(G, \pi, \nu_\ell)$ be a node invariant such that the ordered sequence

$$\vec{I}(G, \pi, \nu_\ell) = (I(G, \pi, \nu_0), I(G, \pi, \nu_1), \ldots, I(G, \pi, \nu_\ell))$$

is a leaf certificate, where the output of $\vec{I}$ is ordered lexicographically. Then $I$ is called an incremental leaf certificate

**Notation.** Let $I$ be an incremental leaf certificate. Then define

$$\mathcal{L}_I(G, \pi) = \left\{ \vec{I}(G, \pi, \lambda) : \lambda \in \mathcal{L}(G, \pi) \right\}.$$ 

That is, $\mathcal{L}_I(G, \pi)$ is the set of all incremental leaf certificate values for the leaf nodes of $T(G, \pi)$.

The canonical labeling is ready to be defined. For completeness, it is proven that it is indeed a canonical labeling. The proof uses the following two lemmas.

**Lemma 2.** Two colored graphs are isomorphic if and only if their set of leaf-invariant values are equivalent: $\mathcal{L}_I(G, \pi) = \mathcal{L}_I(G^\gamma, \pi^\gamma)$ for all $\gamma \in \text{Sym}(V)$.
Proof.

\[ \mathcal{L}_I(G^\gamma, \pi^\gamma) = \left\{ \vec{I}(G^\gamma, \pi^\gamma, \lambda^\gamma) : \lambda^\gamma \in \mathcal{L}(G^\gamma, \pi^\gamma) \right\} \]

= \{ \vec{I}(G^\gamma, \pi^\gamma, \lambda^\gamma) : \lambda \in \mathcal{L}(G, \pi) \} \text{ by Lemma 1, } \lambda \in \mathcal{L}(G, \pi)

= \{ \vec{I}(G, \pi, \lambda) : \lambda \in \mathcal{L}(G, \pi) \} \text{ because } \vec{I} \text{ is a leaf certificate}

= \mathcal{L}_I(G, \pi).

\[ \Box \]

Lemma 3. If two graphs are not isomorphic then their leaf-invariant values are disjoint:

\((G_1, \pi_1) \not\cong (G_2, \pi_2) \text{ implies } \mathcal{L}_I(G_1, \pi_1) \cap \mathcal{L}_I(G_2, \pi_2) = \emptyset.\)

Proof. Assume by contradiction that \(\mathcal{L}(G_1, \pi_1) \cap \mathcal{L}(G_2, \pi_2) \neq \emptyset\). Then there exists a \(\lambda_1 \in \mathcal{L}_I(G_1, \pi_1)\) and a \(\lambda_2 \in \mathcal{L}_I(G_2, \pi_2)\) such that \(\vec{I}(G_1, \pi_1, \lambda_1) = \vec{I}(G_2, \pi_2, \lambda_2)\). Since \(\vec{I}\) is a leaf certificate this implies that \((G_1, \pi_1, \lambda_1) \cong (G_2, \pi_2, \lambda_2)\) which implies that \((G_1, \pi_1) \cong (G_2, \pi_2)\), contradicting the assumption that \((G_1, \pi_1) \not\cong (G_2, \pi_2)\). \(\Box\)

Theorem 2. The function defined by

\[ \text{CL}(G, \pi) = \lambda \in \mathcal{L}(G, \pi) \text{ such that } \vec{I}(G, \pi, \lambda) \text{ is minimal} \]

is a canonical labeling.

Proof. By Lemma 2, isomorphic colored graphs have the same set of leaf-certificate values. By Lemma 3, any leaf-certificate value can be chosen, and a leaf which has this value is then
a canonical labeling. Therefore, any leaf with the minimal leaf-certificate value is a canonical labeling.

3.5 Automorphism discovery and pruning

If two leaves \( \lambda_1 \) and \( \lambda_2 \) have identical leaf-certificate values, \( \vec{I}(G, \pi, \lambda_1) = \vec{I}(G, \pi, \lambda_2) \), then by the property of certificates there exists a \( \gamma \) such that \( (G, \pi, \lambda_1)^\gamma = (G^\gamma, \pi^\gamma, \lambda_1^\gamma) = (G, \pi, \lambda_2) \). Since \( \lambda_1^\gamma = \gamma^{-1} \lambda_1 = \lambda_2 \) (see Equation 3.1), it follows that \( \gamma = \lambda_1 \lambda_2^{-1} \in \text{Aut}(G, \pi) \).

By Corollary 2, the search tree will have at least as many leaves as there are automorphisms of the colored graph (which can be very large; consider \( K_n \) which has \( n! \) automorphisms). Fortunately, any automorphism discovered by finding two leaf nodes with equivalent invariant values can be used to prune large sections of the search tree. Given a node \( \nu \) with two vertices \( u \) and \( v \) in its target cell, if there is a known automorphism \( \gamma \in \text{Aut}(G, \nu) \) such that \( u^\gamma = v \), then only one of the subtrees \( T(G, \nu \vdash u) \) or \( T(G, \nu \vdash v) \) needs to be generated. This is because \( T(G, \nu \vdash u)^\gamma = T(G, \nu \vdash v) \).

Which tree is generated, \( T(G, \nu \vdash u) \) or \( T(G, \nu \vdash v) \), is determined by the linear ordering of the vertices, since children are only generated for vertices which are the minimum in their orbits. The set of generators discovered so far (denoted by \( \theta \subseteq \text{Sym}(V) \)) is used to find minimum cell representatives. Let \( \langle \theta \rangle \) be the group generated by \( \theta \) and \( \langle \theta \rangle_\nu = \{ \gamma \in \langle \theta \rangle : \nu^\gamma = \nu \} \).
\textbf{Definition} (minimum cell representatives). The minimum cell representatives are defined as

\[ \text{mcrs}(\theta, \nu) = \{ \min(X) : X \text{ is an orbit of } \langle \theta \rangle_{\nu} \} . \]

Using this definition of mcrs is called \textit{exact pruning}. Using exact pruning requires the use of computational group theory to change the base of the generating set to the fixed path of the node \( \nu \) (see [40]). In practice, a weaker form of pruning is used.

\textbf{Definition} (approximate minimum cell representatives). Let

\[ \theta'_{\nu} = \{ \gamma \in \theta : \nu \text{'s fixed path is fixed by } \gamma \} . \]

A weaker form of pruning, \textit{approximate pruning}, uses the following approximation of mcrs:

\[ \text{mcrs}'(\theta, \nu) = \{ \min(X) : X \text{ is an orbit of } \langle \theta'_{\nu} \rangle \} . \]

The approximate approach to calculating the minimum cell representatives has the advantage that the orbits of \( \langle \theta'_{\nu} \rangle \) are easy to compute (via union find). The drawback to approximate pruning is that an element which is not minimal in its orbit could be branched on, resulting in the traversal of a subtree already generated by the current generating set. Furthermore, the traversal of this subtree with result in the discovery of a redundant generator.
3.6 Traversing the search tree

In traversing the tree, the smallest invariant value at each level must be stored to facilitate comparing invariants, as well as a “smallest labeling” which is the $\lambda$ such that $\vec{I}(G, \pi, \lambda)$ is the smallest leaf-certificate seen so far.

Nodes in the search tree are traversed in lexicographic order based on their fixed paths. As each node is generated, its invariant value is compared against the previous smallest invariant value seen at the current level. If the invariant value of the node is larger, the search is resumed at the parent of the node. If it is smaller, then the next leaf node traversed will be the new smallest labeling.

The first leaf node traversed is stored as the smallest labeling. As each other leaf is traversed, the smallest labeling is updated if the invariant value of the leaf is smaller than the current smallest. If the invariant value of the leaf is equivalent to the invariant value of the smallest labeling, then a generator of the automorphism group is discovered and stored. In this case, the search is resumed at the greatest common ancestor of the two equivalent leaf nodes.

Before generating a child of a node, it is checked whether the vertex about to be added to the fixed path is the smallest in its orbit (according to previously discovered automorphisms). If the vertex is the smallest in its orbit, the child is generated, otherwise it is not. This ensures that no redundant subtrees will be examined.
At the end of the search, the smallest labeling seen will be the smallest in the tree, and thus correspond to a canonical representative. Also, the discovered automorphisms will generate the full automorphism group. The phrase “size of the search tree” will henceforth refer to the number of nodes traversed in a search, not the number of nodes in $T(G, \pi)$.

### 3.7 An example traversal

To illustrate the algorithm, consider the graph in Figure 7. It has four automorphisms and its vertex set has three orbits. The vertices $a$ and $b$ are in different orbits because the neighbors of $a$ are not adjacent while the neighbors of $b$ are.

![Graph G](image)

Figure 7: A graph $G$ on eight vertices with twelve edges and four automorphisms; the identity permutation, a horizontal flip $(b\ c)\(e\ f)(g\ h)$, a vertical flip $(a\ d)\(e\ g)(f\ h)$ and a $180^\circ$ rotation $(a\ d)(b\ c)(e\ h)(f\ g)$. The orbits of its vertices are $\{a, d\}$, $\{b, c\}$, and $\{e, f, g, h\}$.

The portion of the search tree that is traversed during the execution of the canonical-labeling algorithm is shown in Figure 8. The target-cell function is **first-smallest** and the invariant is **weakest-invar**.
Figure 8: The search tree for the graph $G$ defined in Figure 7. All nodes are equitable partitions, the label on an edge indicates which vertex is fixed, and a permutation under a node indicates the generator that was discovered. Thick edges indicate that the new child has a smaller invariant value than the current smallest at that level; thin edges indicate that the invariant value is equal; dashed edges indicate that the invariant value is larger.

Consider the leftmost leaf node, with fixed path $(a, b)$, which corresponds to the first leaf and overall smallest labeling, and the leaf node with fixed path $(b, a)$, which corresponds to the first node with a larger invariant value than the smallest up to that point. Figure 9 shows the adjacency matrices corresponding to the invariant values of these nodes, illustrating why the final canonical labeling returned is $\lambda_1 = (e \ h)(f \ g)$. The set of generators are the horizontal and vertical flips, which when composed equal a $180^\circ$ rotation.

Generating the three nodes with $b$ as the first element on their fixed path is essentially wasted effort, because they do not lead to the discovery of automorphisms or smaller invariant values. The cell chosen by the target-cell function, $\{a, b, c, d\}$, contains two orbits, $\{a, d\}$ and $\{b, c\}$. This means that the leaf-invariant values of the leaf nodes starting with $a$ on their
(x) The adjacency matrix for $G^{\lambda_1}$ where $\lambda_1 = (e\ h)(f\ g)$

(y) The adjacency matrix for $G^{\lambda_3}$ where $\lambda_3 = (a\ b)(c\ d)(e\ h)(f\ g)$

Figure 9: The invariant values for the first and third leaves. The matrix of the third leaf (y) is greater than the matrix of the first leaf (x) because the first entry at which they differ, row 5, column 6, is larger in the third leaf.

fixed paths will be different from the leaf-invariant values of the leaf nodes starting with $b$ on their fixed paths. As a result, since the ordering of invariant values favors having $a$ first on the fixed path rather than $b$, no generators will be discovered when branching on $b$.

### 3.8 Analysis of search tree size

By Corollary 2, and observing that each equivalence class of leaves corresponds to a leaf-invariant value, the number of leaves in a search tree $T(G, \pi)$ is

$$|\mathcal{L}(G, \pi)| = |\mathcal{L}_I(G, \pi)| \cdot |\text{Aut}(G, \pi)|.$$

In a traversal, the tree is pruned using discovered generators. Ignoring the nodes used to discover the generators (which can be at most $n^2$ [23]), the number of traversed leaf nodes is $|\mathcal{L}_I(G, \pi)|$. Because the tree can have height at most $n = |V|$, an upper bound
on the number of nodes traversed during a search is $n \cdot |L_I(G, \pi)|$. The exponential bound on individualization-refinement algorithms established by Miyazaki is developed by ensuring that $|L_I(G, \pi)|$ is exponential in $n$.

3.9 Pseudocode for a canonical labeling algorithm

This section provides pseudocode for a canonical-labeling algorithm. The algorithm is presented in concise pseudocode which can be implemented in a high level language (like python). The focus is on expressing the essential ideas, not on efficiency. For implementation details and on the data structures used to process large and sparse graphs efficiently, consult [44] and [23].

A linked-list data-structure called a `path_node` stores bookkeeping information for each level in the search. It contains the following fields and method:

- $\nu$ - the current partition
- $I$ - the best invariant value seen at this level
- $W$ - the set of unused branches in the target cell, initially $\emptyset$
- `parent` - the parent of this node
- `next()` - a method returning the `path_node` for the next level, or creating a new `path_node` and returning it if one does not already exist.
The following variables are used throughout the pseudocode:

- **curr** - the current path_node, initially the head of an empty linked-list
- **λ_{min}** - the leaf with the smallest \(\vec{I}(G, \pi, \lambda)\) seen so far
- **cmp** - used for comparing invariants, initially -1
- **ℓ** - the current level in the search (number of elements on the fixed path of \(\text{curr.}\nu\)), initially 0 and always equal to the depth of \(\text{curr}\) in the linked list
- **θ** - the set of generators discovered so far, initially \(\emptyset\)

The following functions are defined:

- **canonical_labeling\((G, \pi, \text{tc}, I)\)** - initializes and steps through the search tree. The arguments \(G, \pi, \text{tc}, I\) are considered global and fixed to all other functions.
- **traverse_search_tree()** - calls **process_node()** or **process_leaf()** as appropriate.
- **process_node()** - generates the next child of \(\text{curr.}\nu\), or backs up to \(\text{curr}\)'s parent if all of \(\text{curr.}\nu\)'s children have been generated.
- **process_leaf()** - updates the new best labeling or discovers a generator of the automorphism group.
- **backup(ℓ_{new})** - backs up to level \(ℓ_{new}\) in the linked list.

The following functions are used, but not defined:
• \text{compare
 invariant}(I) - if \text{cmp} = -1, returns -1, otherwise returns -1 if \( I < \text{curr}.I \), 0 if they are equal, and 1 if \( I > \text{curr}.I \).

• \text{gcalevel}(\nu_1, \nu_2) - returns the level of the greatest common ancestor of the two nodes \( \nu_1 \) and \( \nu_2 \). This is the length of the greatest common prefix of their two fixed paths.

\begin{verbatim}
Subroutine 2 canonical_labeling(G, \pi, tc, I)
Input: colored graph (G, \pi), target cell function tc(G, \pi), and invariant I
Output: labeling \( \lambda_{\text{min}} \) and generators \( \theta \) of Aut(G, \pi)
        curr.\nu = R(G, \pi)
        curr.I = I(G, \pi, curr.\nu)
          traverse_search_tree()
        return \((\lambda_{\text{min}}, \theta)\)

Subroutine 3 traverse_search_tree()

while (\ell \geq 0) do
  if (curr.\nu is not discrete) then
    process_node()
  else
    process_leaf()

return
\end{verbatim}
Subroutine 4 process_node()

Globals Modified:

- curr - curr if the current branch’s invariant value is larger, curr.parent if this node’s branches are exhausted, or curr.child otherwise
- cmp - equal to the comparison value of the invariant

\[ W = curr.W \]

if (curr.k == -1) then
  \[ k = tc(G, curr.\nu) \]
  \[ W = curr.\nu[k] \]

\[ W = W \cap mcrs(\theta, curr.\nu) \] // pruning with generators

if (W == \emptyset) then
  backup(\ell - 1)
  return

\[ b = \min(W) \]
\[ W = W \setminus \{b\} \]
\[ curr.W = W \]
\[ \nu_b = R(G, curr.\nu \vdash b) \]
\[ I_b = I(G, \pi, \nu_b) \]
\[ cmp = compare_invariant(I_b) \]

if (cmp \leq 0) then
  \[ \ell = \ell + 1 \]
  curr = curr.next()
  curr.\nu = \nu_b
  curr.I = I_b
Subroutine 5 process_leaf()

Globals Modified:
- curr - curr.parent if cmp == −1 or backs up to the greatest common ancestor
- cmp - reset to zero θ - gains an element if cmp == 0
- λ_min - updated if cmp == −1

λ = curr.ν

if (cmp == −1) then
    λ_min = λ
    cmp = 0
    backup(ℓ − 1)
else
    γ = λϕ−1
    θ = θ ∪ {γ}
    ℓ_gca = gcalevel(λ, ϕ)
    backup(ℓ_gca)

Subroutine 6 backup(ℓ_new)

while (ℓ > ℓ_new) do
    curr.b = −1
    curr = curr.parent
    ℓ = ℓ − 1
CHAPTER 4
CANONICAL LABELING OF MIYAZAKI GRAPHS

The algorithm described in Chapter 3 is very effective and embodies the essential idea behind \textit{nauty}, \textit{saucy}, and \textit{bliss}. However, it was shown by Miyazaki in [69] that \textit{nauty} has exponential runtime. A family of colored graphs with parameter $k$ was constructed such that the size of the search tree is $O(c^k)$ where $c > 1$. Another coloring which resulted in a search tree of polynomial size with respect to $k$ was also shown.

The two colorings have the same set of cells, but they differ by how the cells are ordered. The ordering of the cells affects the target cell, which determines the number of children of a search node.

This chapter presents the construction of Miyazaki graphs and the reasons why different colorings yield vastly different search tree sizes. Then, a method is presented which guarantees that colored Miyazaki graphs will be canonically labeled in polynomial time, followed by an empirical study verifying the effectiveness of the method.
4.1 Miyazaki graphs

**Definition** (the multigraph $Y_k$). Let $Y_k(V, E)$ be an undirected multigraph with self-loops where $V$ and $E$ are defined as:

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

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

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

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

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

The multigraph $Y_k$ consists of a self-loop at each end and a series of $k - 1$ cycles connected to each other via bridges (Figure 10). It has $2k$ vertices and $3k$ edges.

**Example.** The multigraph $Y_2$ is shown in Figure 10.

![Figure 10: The multigraph $Y_2$ which consists of four vertices and six edges (two bridges and three cycles).](image)

Miyazaki graphs are constructed by replacing each vertex in $Y_k$ by gadgets $Ξ_3$ called *Fürer gadgets* (see [17] by Cai, Fürer, and Immerman).
**Definition** (Fürer gadget [17]). A Fürer gadget $\Xi_n = (V, E)$ is defined by

\[
V_1 = \{ S \subseteq \{v_1, \ldots, v_n\} : |S| \text{ is even} \}, \text{ the } \textit{internal} \text{ vertices},
\]

\[
V_2 = \{ v_i, \overline{v}_i : 1 \leq i \leq n \}, \text{ the } \textit{edge} \text{ vertices},
\]

\[V = V_1 \cup V_2, \text{ and}\]

\[
E_1 = \{ \{v_i, S\} : v_i \in V_2, S \in V_1, \text{ and } v_i \in S\}, \text{the } \textit{contains} \text{ edges}
\]

\[
E_2 = \{ \{\overline{v}_i, S\} : \overline{v}_i \in V_2, S \in V_1, \text{ and } v_i \not\in S\}, \text{the } \textit{does not contain} \text{ edges}, \text{ and}
\]

\[E = E_1 \cup E_2.\]

$\Xi_n$ is designed to replace a vertex of degree $n$, but since each vertex in $Y_k$ has degree 3, only the $n = 3$ case is considered. Figure 11 shows $\Xi_3$ using the convention that $a = v_1$, $b = v_2$, $c = v_3$. Each group of vertices (denoted by the dotted lines) is placed in a different partition than the others, so that automorphisms can only move internal vertices to internal vertices and each edge-vertex can only move to its compliment.

**Definition** (Miyazaki graphs [69]). The application of Fürer gadgets to $Y_k$ generates the Miyazaki graph with parameter $k$, denoted $\mathcal{X}(Y_k)$. It has $20k$ vertices, $30k$ edges and is three-regular. Each vertex in $Y_k$ is replaced by 10 vertices in $\mathcal{X}(Y_k)$. Each vertex in $Y_k$ adds 4 colors, so an unordered partition $\vartheta_k$ with $8k$ cells is created. Edge-vertices derived from bridges are called $\textit{bridge-vertices}$ and edge-vertices derived from cycles are called $\textit{cycle-vertices}$. Figure 12 illustrates $(\mathcal{X}(Y_2), \vartheta_2)$. 

48
An effect of this construction is that the cycle subspace of $Y_k$ is isomorphic (in a group sense) to the automorphism group of $(\mathcal{X}(Y_k), \vartheta_k)$. Since $Y_k$ has $k + 1$ cycles and each cycle is either present or not, there are $2^{k+1}$ vectors in the cycle subspace of $Y_k$ and $2^{k+1}$ automorphisms in $\text{Aut}(\mathcal{X}(Y_k), \vartheta_k)$. Figure 13 illustrates three natural generators of $(\mathcal{X}(Y_3), \vartheta_3)$. This also gives insight into the orbits of the automorphism group; cells containing bridge-vertices and cells containing internal-vertices contain two orbits, while cells containing cycle-vertices only contain one orbit.
The coloring $\vartheta_{a,k}$ forces the target-cell function \texttt{first-smallest} to choose the cells containing cycle-vertices first. Each of these cells contains only one orbit thus only one leaf-certificate value is seen throughout the whole search, and all leaf nodes other than the first yield generators. The coloring $\vartheta_{b,k}$ forces \texttt{first-smallest} to choose the cells containing bridge-vertices first. Each of these cells contains two orbits; thus each node is guaranteed to have at least two children. Similar to $\vartheta_{b,k}$, the coloring $\vartheta_{c,k}$ forces \texttt{max-joins} to choose the cells with internal-vertices first,
(w) The graph \((\mathcal{X}(Y_k), \vartheta_k)\) for \(k = 2\). Each vertex in \(Y_k\) adds four colors which contain the internal vertices, two bridge-vertices and two pairs of cycle-vertices.

(x) the coloring \(\vartheta_{a,k}\) which orders the colors (cycle)\(4^k\) (internal)\(2^k\) (bridge)\(2^k\); gives polynomial-size search trees with \(tc = \text{first-smallest}\)

(y) the coloring \(\vartheta_{b,k}\) which orders the colors (bridge)\(2^k\) (cycle, cycle, internal)\(2^k\); gives exponential-size search trees with \(tc = \text{first-smallest}\)

(z) the coloring \(\vartheta_{c,k}\) which orders the colors (internal)\(2^k\) (bridge)\(2^k\) (cycle)\(4^k\); gives exponential-size search trees with \(tc = \text{max-joins}\)

Figure 14: The Miyazaki graph for \(k = 2\) (w); the corresponding colorings \(\vartheta_{a,k}\) (x) and \(\vartheta_{b,k}\) (y), introduced in [69]; the coloring \(\vartheta_{c,k}\) (z), introduced here.

which have two orbits as well. For both of these colorings and the corresponding target-cell functions, the search tree has a complete binary-tree of height \(k\) as a subgraph starting at the root.

The following propositions about \textit{nauty} (at version 1.5), which demonstrate that the coloring can be the difference between polynomial and exponential runtime, were proven by Miyazaki.

**Proposition 2** (due to Miyazaki). \textit{Given} \((\mathcal{X}(Y_k), \vartheta_{a,k})\), \textit{where} \(n = 20k\), \textit{the algorithm nauty generates a search tree having} \(O(n^2)\) \textit{nodes to compute canonical form and generators of the automorphism group.}
Proposition 3 (due to Miyazaki). Given $(\mathcal{X}(Y_k), \vartheta_{b,k})$, where $n = 20k$, the algorithm nauty generates $\Omega(c^n)$ nodes in the search tree to compute the canonical form and generators of the automorphism group for some constant $c > 1$.

4.2 Canonical labeling of Miyazaki graphs

The colorings $\vartheta_{a,k}$, $\vartheta_{b,k}$ and $\vartheta_{c,k}$ demonstrate the importance of choosing a cell which contains few orbits. Cells with multiple orbits are guaranteed to have at least two children. Therefore, if before the algorithm starts, $\text{Aut}(G, \pi)$ is known, then the following target-cell function would be desirable:

$$\text{min-orbits}(G, \pi) = k \text{ such that } \pi[k] \text{ contains the fewest orbits in } \text{Aut}(G, \pi).$$

To help show that this satisfies the requirements of a target-cell function, it is first shown that given a colored graph $(G, \pi)$ and a cell $W$ of $\pi$, the sorted list of orbit sizes of the cell $W$ is permutation independent and thus can be used by the target-cell function. The proof makes use of the following lemmas.

Lemma 4. Acting on a colored graph $(G, \pi)$ by $\delta \in \text{Sym}(V)$ maps the automorphism $\gamma \in \text{Sym}(V)$ to $\delta^{-1}\gamma\delta \in \text{Aut}(G^\delta, \pi^\delta)$ and vice versa. That is, $\text{Aut}(G^\delta, \pi^\delta) = \delta^{-1} \text{Aut}(G, \pi)\delta$. 

52
Proof. An automorphism $\gamma \in \text{Aut}(G, \pi)$ has the property that

$$(G, \pi)^\gamma = (G, \pi) = (G, \pi)^{-1}$$

if and only if

$$(G, \pi)^{\delta \gamma \delta^{-1}} = (G, \pi)^{\gamma^{-1} \delta \gamma \delta^{-1}} \quad \text{(act on with } \delta \gamma \delta^{-1})$$

$$= (G, \pi)^{\gamma^{-1} \gamma \delta}$$

$$(G, \pi)^{\delta \gamma \delta^{-1}} = (G, \pi)^{\delta} \text{ if and only if } \delta^{-1} \gamma \delta \in \text{Aut}(G^\delta, \pi^\delta).$$

Therefore, $\gamma \in \text{Aut}(G, \pi)$ if and only if $\delta^{-1} \gamma \delta \in \text{Aut}(G^\delta, \pi^\delta)$ for all $\delta \in \text{Sym}(V)$. \hfill \Box

Lemma 5. Let $(G, \pi)$ be a colored graph. Then $X$ is an orbit of $\text{Aut}(G, \pi)$ if and only if $X^\delta$ is an orbit of $\text{Aut}(G^\delta, \pi^\delta)$.

Proof. Let $X$ be an orbit of $\text{Aut}(G, \pi)$ and let $u \in X$. Then

$$X = \{u^\gamma : \gamma \in \text{Aut}(G, \pi)\} \text{ if and only if}$$

$$X^\delta = \{u^{\gamma \delta} : \gamma \in \text{Aut}(G, \pi)\}$$

$$= \{u^{(\delta \gamma \delta^{-1})} : \gamma \in \text{Aut}(G, \pi)\} \quad \text{(since } \delta \gamma \delta^{-1} \text{ is the identity)}$$

$$= \{u^{\delta \gamma \delta^{-1}} : \delta^{-1} \gamma \delta \in \text{Aut}(G^\delta, \pi^\delta)\} \quad \text{(by Lemma 4)}$$

$$= \{u^{(\delta^{-1} \gamma \delta)} : \delta^{-1} \gamma \delta \in \text{Aut}(G^\delta, \pi^\delta)\}.$$  

Since each orbit $X$ of $\text{Aut}(G, \pi)$ contained in $W$ corresponds to an orbit $X^\delta$ of $\text{Aut}(G^\delta, \pi^\delta)$ for all $\delta \in \text{Sym}(V)$, the lemma holds. \hfill \Box
Corollary 3. The colored graph \((G, \pi)\) has orbits \(X_1, X_2, \ldots, X_m\) if and only if \((G^\delta, \pi^\delta)\) has orbits \(X_1^\delta, X_2^\delta, \ldots, X_m^\delta\) for all \(\delta \in \text{Sym}(V)\). In particular, the sorted list of orbit sizes are the same and thus can be used in calculating target-cell functions.

Unfortunately, the generators of \(\text{Aut}(G, \pi)\) are not known before starting the algorithm and determining them is as hard as solving graph isomorphism [61]. However, the set of generators discovered at each point in a search provides partial knowledge of \(\text{Aut}(G, \pi)\) and can be used to approximate the full group until all of the generators are discovered. Hence, given a partial set of generators \(\theta \subseteq \text{Sym}(V)\), the following target-cell function can be used:

\[
\text{min-orbits}(G, \pi, \theta) = \text{the first nontrivial cell which contains the fewest orbits in the group generated by } \theta.
\]

4.2.1 A better target cell choice through generator discovery: an example

Figure 14 illustrates the Miyazaki graph with parameter \(k = 2\). Its vertex set is \(\{0, \ldots, 39\}\). Consider its search tree when using the coloring \(\vartheta_b, 2\) and \(\text{tc} = \text{first-smallest}\). The root node is \(\vartheta_b, 2\) and the first generator discovered, which moves the vertices in the rightmost cycle, is

\[
\gamma_1 = (32 33)(34 35)(36 37)(38 39).
\]
After discovering this generator, a different choice for the target cell is revealed (see Figure 15).

Figure 15: The root node of $(\mathcal{X}(Y_2), \vartheta_{b,2})$ and the root node filtered via orbits after discovering the first generator. The initial target-cell choice is \{0, 1\}, but after discovering the first generator it becomes \{32, 33\}. Dashed lines in the bottom partition separate vertices not known to be in the same orbit.

A new target-cell choice has been found for the root node in the search tree. What action should the canonical-labeling algorithm take upon receiving this information (if any)? One possibility is to restart the search at the root node, but using the new target-cell choice instead of the old one. This leads to a general strategy.

After two leaves $\lambda_1$ and $\lambda_2$ are found to have equivalent invariant values and a generator $\gamma$ is discovered, start at the root node and move down to the greatest common ancestor of $\lambda_1$ and $\lambda_2$. At each node, reevaluate the target-cell choice with the additional information provided by the new generator. If a target-cell choice has changed, backup and resume the search (using the new target-cell choice) where the change occurred. Otherwise, just return to the greatest common ancestor as usual.
4.2.2 An example traversal using \textit{min-orbits}

Figure 16 illustrates this strategy using the graph in Figure 7 (see Section 3.7) but with a target-cell function of \textit{min-orbits} instead of \textit{first-smallest}.

Figure 16: The search tree for the graph defined in Figure 7, using \textit{min-orbits}. After discovering the generator \((b\ c)(e\ f)(g\ h)\) the cell \(\{a, b, c, d\}\) in the root node contains three orbits: \(\{a\}, \{b, c\},\) and \(\{d\}\); while the cell \(\{e, f, g, h\}\) contains only two: \(\{e, f\}\) and \(\{g, h\}\). Thus the search is restarted at the root using \(\{e, f, g, h\}\) as the target cell. The labeling \((a\ d)(b\ c)(g\ h)\) is returned.

After discovering the first generator, the search is resumed at the root and the second, rather than the first cell, is used as the target cell. After discovering the second generator, the target cell of the root node is reevaluated again, and determined to be unchanged. Note that this search tree has six nodes while the search tree in Figure 8 has nine.
4.3 Guide trees

When resuming the search using a new target cell, the partial search completed using the previous target-cell choice should not be abandoned. With the new target-cell choice, it is possible to discover a generator for which the target cell function returns the old target cell. In this instance, the search should be resumed exactly where it left off before switching target cells. To allow this resumption of the search, a structure called a guide tree keeps track of the state of the search for each sequence of target-cell choices encountered in the search.

A guide tree consists of several guide nodes. Each guide node holds the following data and methods:

- $\nu$ - the current search node
- $k$ - the current target-cell choice
- $I$ - the smallest invariant seen
- $W$ - the set of unused branches in the target cell of the parent’s node
- $\lambda$ - the smallest labeling if a leaf is found at this guide node, $\emptyset$ if no leaf is found at this guide node
- $fp_\lambda$ - the fixed path of the smallest labeling (if it exists for this node)
- $\text{child}(k)$ - a method returning (or creating and returning) the guide node using $k$ as the target cell
- **clear_descendants()** - a method which clears the $b$ and $k$ values of all descendants

- **clear_for_better_invar()** - a method which clears the $b$, $k$, and $W$ values and deletes all children (the new invariant value makes them obsolete)

- **remove_branch()** - removes $b$ from $W$

At the conclusion of traversing the search tree, the smallest labeling must be recovered from the guide tree. Each node in the guide tree which has a smallest labeling stores this labeling’s fixed path. A labeling is valid if, using the final set of generators, the elements of its fixed paths are in the target cells returned by the target-cell function. Finally, of all valid labelings, the one with the smallest leaf certificate is returned by the algorithm. This is made explicit by Subroutine 7 and Subroutine 8.

**Subroutine 7 is_valid_path($G, \pi, fp = (v_1, v_2, \ldots, v_\ell), \theta$)**

**Input:** colored graph $(G, \pi)$, a fixed path $fp = (v_1, v_2, \ldots, v_\ell)$ and the set of generators $\theta \subseteq \text{Sym}(V)$

**Output:** True if values of $fp$ are in the target cells when using $\theta$, False otherwise

$\nu = R(G, \pi)$

**for** ($v$ in $fp$) **do**

$k = \text{tc}(G, \nu, \theta)$

**if** (the index of $\nu$ containing $v \neq k$) **then**

return false

$\nu = R(G, \nu \vdash v)$

**return** true
Subroutine 8 recover_labeling\((G, \pi, \text{guide\_tree})\)

**Input:** colored graph \((G, \pi)\) and the guide tree

**Output:** the leaf with the smallest certificate whose fixed path is valid

\[
I_{\min} = \infty \\
\lambda_{\min} = \emptyset
\]

for (guide\_leaf in guide\_tree) do

\[
\lambda = \text{guide\_leaf.}\lambda
\]

if (is\_valid\_path(\lambda's fixed path) ) then

if (\(\vec{I}(G, \pi, \lambda) < I_{\min}\) then

\[
I_{\min} = \vec{I}(G, \pi, \lambda) \\
\lambda_{\min} = \lambda
\]

return \(\lambda_{\min}\)

4.3.1 An example using Miyazaki graphs

Figure 17 illustrates the difference between using the target cell functions \textit{first-smallest} and \textit{min-orbits} on the graph \((\mathcal{X}(Y_3), \vartheta_{b,3})\). Using \(tc = \textit{first-smallest}\) results in a search tree of size 57 containing a complete binary-tree of height three, while using \(tc = \textit{min-orbits}\) results in a search tree of size 31 which reevaluates the target cell choice at each discovery of a generator, eventually choosing target cells with only one orbit each.

The guide tree when using \(tc = \textit{first-smallest}\) is just a path, whereas the guide tree when using \(tc = \textit{min-orbits}\) has more than one leaf. However, only one of these leaves is valid as Figure 18 illustrates.

In general, the number of nodes traversed for \((\mathcal{X}(Y_k), \vartheta_{b,k})\) using \(tc = \textit{first-smallest}\) is

\[
2^k(k + 3) + \frac{(k + 1)(k + 2)}{2} - 1,
\]
Figure 17: Two different search trees (displayed compactly) for \((X(Y_3), \vartheta_{b,3})\).

Figure 18: The guide tree for the search tree in Figure 17(y). Each guide node represents a unique series of target-cell choices. Thick lines lead to the guide node which stores the smallest valid labeling.

while the number of nodes traversed when using \(tc = min-orbits\) is

\[
(k + 1)(k + 3) + \frac{k(k + 1)}{2} + 1.
\]
4.4 Pseudocode for a canonical labeling algorithm using a guide tree

This section provides pseudocode for a canonical-labeling algorithm which allows for the target-cell function to take in a third parameter, $\theta$, the set of generators discovered so far. This allows for making target-cell decisions based on the automorphism group.

The name of the modified algorithm will be called $nishe$ and all modified methods have $nishe$ in their suffix. The following functions need to be modified from those in Section 3.9:

- **canonical_labeling_nishe**($G, \pi, tc, I$) - traverses the search tree and then calls $recover$ labeling (the variable $\lambda_{\text{min}}$ is no longer global)

- **process_node_nishe**() - similar to $\text{process\_node}()$ but modifies the $k$-child of $curr$ rather than $curr$.

- **process_leaf_nishe**() - updates the new best labeling (which is now local to the guide node) or discovers a generator of the automorphism group.

- **backup_nishe**() - backs up $\ell - \ell_{\text{new}}$ levels in the guide tree and clears the guide tree below the new $curr$
Subroutine 9 canonical_labeling_nishe($G$, $\pi$, tc, $I$)

**Input:** colored graph ($G$, $\pi$), target cell function tc($G$, $\pi$), and invariant $I$

**Output:** labeling $\lambda_{\min}$ and generators $\theta$ of Aut($G$, $\pi$)

root = curr
curr.$\nu$ = R($G$, $\pi$)
curr.$I$ = I($G$, $\pi$, curr.$\nu$)

traverse_search_tree()
$\lambda_{\min}$ = recover_labeling($G$, $\pi$, root)

return ($\lambda_{\min}$, $\theta$)
Subroutine 10 process_node_nishe()

Globals Modified:
curr - curr if the current branch’s invariant value is larger, curr.parent if this node’s branches are exhausted, or curr.child(curr.k) otherwise
cmp - equal to the comparison value of the invariant

\[ k = curr.k \]
\[ \nu = curr.\nu \]

if \((k == -1)\) then
\[ k = tc(G, \nu, \theta) \]
\[ curr.k = k \]

child = curr.child(k)

if \((\text{child.b} == -1)\) then
\[ \text{child.W} = \nu[k] \]

\[ W = \text{child.W} \]
\[ W = W \cap \text{mcrs}(\theta, \nu) \]

if \((W == \emptyset)\) then
\[ \text{curr.remove_branch()} \]
\[ curr.k = -1 \]
\[ \text{backup}(\ell - 1) \]
\[ \text{return} \]

\[ b = \min(W) \]
\[ \text{child.b} = b \]
\[ \text{child.W} = W \]
\[ \nu_b = R(G, \nu \vdash b) \]
\[ I_b = I(G, \pi, \nu_b) \]
\[ \text{cmp} = \text{compare_invariant}(I_b) \]

if \((\text{cmp} \leq 0)\) then
  if \((\text{cmp} = -1)\) then
    \[ \text{child.clear_for_better_invar()} \]
    \[ \ell = \ell + 1 \]
    \[ \text{child.I} = I_b \]
    \[ \text{child.\nu} = \nu_b \]
    \[ \text{curr} = \text{child} \]
  else
    \[ \text{curr.remove_branch()} \]
Subroutine 11 process_leaf_nishe()

Globals Modified:
curr - curr.parent if cmp == -1 or backs up to the first guide node whose target cell choice changes (not going past the greatest common ancestor)
cmp - reset to zero θ - gains an element if cmp == 0

λ = curr.ν

if (cmp == -1) then
    curr.φ = λ
    cmp = 0
    backup(ℓ - 1)
else
    γ = λ(curr.φ)^-1
    θ = θ ∪ {γ}
    ℓ_gca = gcalevel(λ, curr.φ)
    i = 0
    temp = root

while (i ≤ ℓ_gca) do
    k = tc(G, temp.ν, θ)

    if (temp.k ≠ k) then
        temp.k = k
        ℓ = i
        return

    temp = temp.child(k)
    i = i + 1

backup(gcalevel(λ, φ))

Subroutine 12 backup_nishe(ℓ_new)

while (ℓ > ℓ_new) do
    curr = curr.parent
    ℓ = ℓ - 1

if (ℓ ≥ 0) then
    child = curr.child(k)
    child.b = -1
    child.clear_descendants()
4.5 Correctness of target-cell reevaluation

The method introduced in this chapter allows for the target-cell function to utilize information regarding the automorphism group. However, this depends on knowledge of the full group. Since this is not known initially, it must be approximated using a partial generating set. It must be shown that eventually, the full group will be discovered.

In addition, Corollary 3 demonstrates why the orbits of the full automorphism group can be used, but to what extent is knowledge of the automorphism group allowed to be used by the enhanced target-cell function?

4.5.1 Group target-cell functions

The conventional target-cell function must be augmented with a third parameter, as well as additional constraints placed on that parameter.

Definition (group target-cell function). A group target-cell function $\text{tc}(G, \pi, A)$ takes as input a colored graph $(G, \pi)$ and a subgroup $A$ of $\text{Aut}(G, \pi)$. The function must satisfy the following properties:

(i) $k \in \text{indices}(\pi)$

(ii) $|\pi[k]| \geq 2$

(iii) $\text{tc}(G^\gamma, \pi^\gamma, \gamma^{-1}A\gamma) = k = \text{tc}(G, \pi, A)$ for all $\gamma \in \text{Sym}(V)$. 

65
The restriction that $A$ is a subgroup of $\text{Aut}(G, \pi)$ is not strictly necessary, but it simplifies the definition of the corresponding search tree. Properties (i) and (ii) still ensure that $\text{tc}(G, \pi, A)$ returns the index of a nontrivial cell. Property (iii) ensures that $\text{tc}(G, \pi, A)$ is independent of the vertex labeling, with respect to the group as well. In fact, a well-known group action (conjugation) can be defined on groups to allow a more concise and intuitive notation.

**Proposition 4.** Let $A \subseteq \text{Sym}(V)$ and $\gamma \in \text{Sym}(V)$. Then define the action of $\gamma \in \text{Sym}(V)$ on $a \in A$ by $a^\gamma = \gamma^{-1}a\gamma$.

**Proof.** A group action must satisfy the following for all $a \in A$:

(i) $(a^\gamma)^\delta = a^{(\gamma^\delta)}$ for all $\gamma, \delta \in \text{Sym}(V)$, and

(ii) $a^e = a$ where $e$ is the identity element of $\text{Sym}(V)$.

The second criterion is satisfied because $a^e = e^{-1}ae = a$ by the definition of identity element.

The first is satisfied because

$$(a^\gamma)^\delta = \delta^{-1}a^\gamma \delta$$

$$= \delta^{-1}(\gamma^{-1}a\gamma)\delta$$

$$= (\gamma\delta)^{-1}a(\gamma\delta)$$

$$= a^{(\gamma\delta)}.$$
Therefore, by Proposition 4, the third requirement for target-cell functions can be restated as

(iii) $\text{tc}(G^\gamma, \pi^\gamma, A^\gamma) = k = \text{tc}(G, \pi, A)$ for all $\gamma \in \text{Sym}(V)$.

4.5.2 Group search-trees

Using a group target-cell function changes the definition of the search tree. It makes sense to speak of the tree $T(G, \pi, A)$ which uses a group target-cell function. A partition stabilizer must first be defined in order to define the group search tree.

**Definition** (partition stabilizer). Given a subgroup $A$ of $\text{Sym}(V)$ and a partition $\pi \in \Pi(V)$, the partition stabilizer of $\pi$ in $A$ is the set of all permutations in $A$ which fix $\pi$, denoted by $A_\pi$. That is,

$$A_\pi = \{ \gamma \in A : \pi^\gamma = \pi \}.$$

**Definition** (group search tree). Given a refiner $R$, a subgroup $A$ of $\text{Aut}(G, \pi)$, and a group target-cell function $\text{tc}$, the group search tree is defined inductively as follows:

1. $r = R(G, \pi) \in \mathcal{N}$
2. If $\nu \in \mathcal{N}$ and $\nu$ is discrete, then $\nu$ is a leaf. Otherwise, let $k = \text{tc}(G, \nu, A_\nu)$. If $u \in \nu[k]$ and $\nu_u = R(G, \nu \uparrow u)$ then $\nu_u \in \mathcal{N}$ and $(\nu, \nu_u, u) \in \mathcal{E}$.

Note that the group target-cell function is restricted to using the stabilizer of $\nu$ in $A$.

To what extent do the results regarding search trees apply to group search trees? A variant of Lemma 1 still applies, but the proof requires the use of another lemma.

**Lemma 6.** Let $A$ be a subgroup of $\text{Sym}(V)$, $\nu \in \Pi(V)$, and $\gamma \in \text{Sym}(V)$. Then $(A_\nu)^\gamma = (A_\nu^\gamma)_{\nu^\gamma}$.

**Proof.** It is first shown that $(A_\nu)^\gamma \subseteq (A_\nu^\gamma)_{\nu^\gamma}$. If $\delta \in A$ and $\nu^\delta = \nu$, then observe that $\nu^{\gamma \gamma^{-1} \delta \gamma} = \nu^{\gamma \gamma^{-1} \delta \gamma} = \nu^\gamma$. Thus, $\nu^\gamma$ is fixed by $\gamma^{-1} \delta \gamma$ and hence $\gamma^{-1} \delta \gamma \in (A_\nu)^\gamma$. To show the other direction, that $(A_\nu^\gamma)_{\nu^\gamma} \subseteq (A_\nu)^\gamma$, let $\delta \in A$ such that $(\nu^\gamma)^{-1} \delta \gamma = \nu^\gamma$. Then multiplying on the right by $\gamma^{-1}$ gives $\nu^\delta = \nu$ and hence $\delta \in A_\nu$ and therefore $\gamma^{-1} \delta \gamma \in (A_\nu)^\gamma$. 

Now, the extended version of Lemma 1 can be proven. The proof is very similar, but the group target-cell function is used instead of the normal target-cell function.

**Lemma 7.** A node $\nu$ with fixed path $u_1, u_2, \ldots, u_m$ where $m \geq 0$ is in $T(G, \pi, A)$ if and only if $\nu^\gamma$ is a node in $T(G^\gamma, \pi^\gamma, A^\gamma)$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma$ for all $\gamma \in \text{Sym}(V)$.

**Proof.** This is shown by induction on $m$, the fixed path length of $\nu$. Observe that $R(G, \pi)^\gamma = R(G^\gamma, \pi^\gamma) = r^\gamma$ by Property (ii) of refiners, so the root nodes of $T(G, \pi, A)^\gamma$ and $T(G^\gamma, \pi^\gamma, A^\gamma)$ are equal and the base case of $m = 0$ holds.
Assume that $m \geq 1$ and let $\nu_{m-1}$ be the parent of $\nu$. By hypothesis, $\nu_{m-1}$ is a node in $T(G, \pi, A)$ with fixed path $u_1, u_2, \ldots, u_{m-1}$ if and only if $\nu_{m-1}^\gamma$ is a node in $T(G^\gamma, \pi^\gamma, A^\gamma)$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_{m-1}^\gamma$. The definition of the group search tree will be used to show that $\nu_{m-1} \xrightarrow{u_m} \nu$ is in $T(G, \pi, A)$ if and only if $\nu_{m-1}^\gamma \xrightarrow{u_m^\gamma} \nu^\gamma$ is in $T(G^\gamma, \pi^\gamma, A^\gamma)$.

Let $k = \text{tc}(G, \nu_{m-1}, A)$, then by Property (iii) of group target-cell functions,

$$
k = \text{tc}(G, \nu_{m-1}, A) = \text{tc}(G^\gamma, \nu_{m-1}^\gamma, (A_{\nu_{m-1}})^\gamma) = \text{tc}(G^\gamma, \nu_{m-1}^\gamma, (A^\gamma)_{\nu_{m-1}^\gamma}) \quad \text{(by Lemma 6)}.
$$

Therefore, $u_m \in \nu_{m-1}[k]$ if and only if $u_m^\gamma \in \nu_{m-1}^\gamma[k]$ by the defined group action on partitions. This means $\nu_{m-1}$ has a child $R(G, \nu_{m-1} \vdash u_m) = \nu$ if and only if $\nu_{m-1}^\gamma$ has a child $R(G^\gamma, \nu_{m-1}^\gamma \vdash u_m^\gamma)$. Since $\nu_{m-1}^\gamma \vdash u_m^\gamma = (\nu_{m-1} \vdash u_m)^\gamma$ by the definition of individualization, $R(G^\gamma, \nu_{m-1}^\gamma \vdash u_m^\gamma) = R(G^\gamma, (\nu_{m-1} \vdash u_m)^\gamma)$. Therefore, by Property (ii) of refiners, $R(G^\gamma, (\nu_{m-1} \vdash u_m)^\gamma) = R(G, \nu_{m-1} \vdash u_m)^\gamma = \nu^\gamma$. Thus, $\nu_{m-1} \xrightarrow{u_m} \nu$ is in $T(G, \pi, A)$ if and only if $\nu_{m-1}^\gamma \xrightarrow{u_m^\gamma} \nu^\gamma$ is in $T(G^\gamma, \pi^\gamma, A^\gamma)$, proving the lemma. 

A slight variant of Theorem 1 also applies to group search trees. However, the proof is nearly identical and thus omitted.

**Theorem 3.** Given a colored graph $(G, \pi)$, a refiner $R$, a subgroup $A$ of $\text{Aut}(G, \pi)$ and a group target-cell function $\text{tc}$, $T(G, \pi, A)^\gamma = T(G^\gamma, \pi^\gamma, A^\gamma)$. That is, two colored graphs are isomorphic if and only if their group search trees are isomorphic.
Corollary 1 does not in general apply to its group search tree analog. This poses a problem when trying to determine generators of the automorphism group. Corollary 1 is restated here:

**Corollary 1.** A node $\nu$ with fixed path $u_1, u_2, \ldots, u_m$ is in $T(G, \pi)$ if and only if the node $\nu^\gamma$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma$ is in $T(G, \pi)$ for all $\gamma \in \text{Aut}(G, \pi)$.

**Proposition 5.** The analog of Corollary 1 does not in general apply to group search trees. It is not true that for any subgroup $A$ of $\text{Aut}(G, \pi)$, a node $\nu$ with fixed path $u_1, u_2, \ldots, u_m$ is in $T(G, \pi, A)$ if and only if the node $\nu^\gamma$ with fixed path $u_1^\gamma, u_2^\gamma, \ldots, u_m^\gamma$ is in $T(G, \pi, A)$ for all $\gamma \in \text{Aut}(G, \pi)$.

**Proof.** Let $(G, \pi)$ be any colored graph such that $\text{Aut}(G, \pi)$ has a subgroup $A$ which is not normal in $\text{Aut}(G, \pi)$. Such a colored graph exists because given any finite group, a finite graph exists whose automorphism group is isomorphic (in a group sense) to the finite group (a theorem of Frucht [33]). Then since $A$ is not normal in $\text{Aut}(G, \pi)$, there exists a $\gamma \in \text{Aut}(G, \pi)$ such that $\gamma^{-1} A \gamma \neq A$, or $A^\gamma \neq A$ (using the group action notation). Therefore, $T(G, \pi, A)^\gamma = T(G, \pi, A^\gamma)$ but $T(G, \pi, A^\gamma) \neq T(G, \pi, A)$ and thus the corollary is not true in general. On the other hand, the corollary is true whenever $A$ is a normal subgroup of $\text{Aut}(G, \pi)$ since then $T(G, \pi, A)^\gamma = T(G, \pi, A)$ for all $\gamma \in \text{Aut}(G, \pi)$.

In particular, since $\text{Aut}(G, \pi)$ is always a normal subgroup of itself, a valid choice for $A$ is $A = \text{Aut}(G, \pi)$. This means that using a group target-cell function where the group is $\text{Aut}(G, \pi)$ will result in a group search tree for which both the generators of the automorphism group, as well as a canonical labeling can be found.
4.5.3 Approximating the automorphism group

It has been shown that the automorphism group can be used as the input to a group target-cell function. However, \( \text{Aut}(G, \pi) \) is initially unknown. In a search tree \( T(G, \pi) \) that uses a normal target cell function, generators are discovered throughout the traversal. This results in a sequence of generating sets \( \theta_0, \theta_1, \ldots, \theta_m \) such that \( \theta_0 = \emptyset \) generates the trivial group and \( \theta_m \) generates \( \text{Aut}(G, \pi) \).

The idea behind target-cell reevaluation is that each time a new generator is discovered, the search is restarted where a new target-cell choice occurs. This can be viewed abstractly as traversing an entirely new group search tree. Let \( A_0 \) be the trivial group. Then initially, \( T(G, \pi, A_0) \) is traversed until a generator \( \gamma_1 \) is discovered. This leads to the traversal of \( T(G, \pi, A_1) \), where \( A_1 = \langle \{ \gamma_1 \} \rangle \). Then, after the discovery of the next generator \( \gamma_2 \), \( T(G, \pi, A_2) \) is traversed, where \( A_2 = \langle \{ \gamma_1, \gamma_2 \} \rangle \). This restarting process continues until for some \( m \), \( A_m = \text{Aut}(G, \pi) \).

To see that the automorphism group can be approximated in such a way, it needs to be shown that if \( A \) is a proper subgroup of \( \text{Aut}(G, \pi) \), then traversing \( T(G, \pi, A) \) will yield a generator \( \gamma \in \text{Aut}(G, \pi) \) not in \( A \).

**Theorem 4.** Given a colored graph \( (G, \pi) \), a group target-cell function \( \text{tc} \), and a proper subgroup \( A \) of \( \text{Aut}(G, \pi) \), a generator of \( \text{Aut}(G, \pi) \) not in \( A \) will be discovered in the traversal of the group search tree \( T(G, \pi, A) \).
Proof. The proof is by induction on the height of $T(G, \pi, A)$. The base case, that the height is 0 is vacuously true since $\text{Aut}(G, \pi)$ is trivial and hence no $A$ can be a proper subgroup. Assume then that the height of $T(G, \pi, A)$ is one or larger. Let $\lambda$ be the first leaf traversed whose leaf-certificate value is minimal, and let $\gamma \in \text{Aut}(G, \pi) \setminus A$ be the automorphism such that $\lambda^\gamma$’s fixed path is minimal. Then there are three cases, all of which lead to the discovery of an automorphism:

Case 1. A generator is discovered before reaching $\lambda$ in the traversal.

Case 2. If no generator is discovered before reaching $\lambda$, and if $\lambda^\gamma$ is in $T(G, \pi, A)$, then because its leaf-certificate value is minimal, it will be found equivalent to $\lambda$ and the generator $\gamma$ will be discovered.

Case 3. If $\lambda^\gamma$ is not in $T(G, \pi, A)$, then let $\nu$ be the first child on the path to $\lambda$ of the greatest common ancestor of $\lambda$ and $\lambda^\gamma$. Observe that $A_\nu = \text{Aut}(G, \nu)$ since if it did not, then there would exist a $\gamma' \in \text{Aut}(G, \pi) \setminus A$ such that $\lambda^{\gamma'}$’s fixed path is less than $\lambda^\gamma$’s fixed path, violating the minimality of $\lambda^\gamma$’s fixed path.
Furthermore, $A_{\nu\gamma} \neq \text{Aut}(G, \nu^\gamma)$, which is shown by contradiction. If $A_{\nu\gamma} = \text{Aut}(G, \nu^\gamma)$ then

$$
T(G, \nu, A_{\nu\gamma}) = T(G, \nu^\gamma, (A_{\nu\gamma})^\gamma) \quad (\text{by Theorem } 3)
$$

$$
= T(G, \nu^\gamma, \text{Aut}(G, \nu^\gamma))
$$

$$
= T(G, \nu^\gamma, \text{Aut}(G, \nu^\gamma)) \quad (\text{by Lemma } 4)
$$

$$
= T(G, \nu^\gamma, \text{Aut}(G, \nu^\gamma)) \quad (\text{because } \gamma \text{ is an automorphism})
$$

$$
= T(G, \nu^\gamma, A_{\nu\gamma}).
$$

Therefore, if $\lambda$ is a leaf in $T(G, \nu, A_{\nu\gamma})$ then $\lambda^\gamma$ is a leaf in $T(G, \nu^\gamma, A_{\nu\gamma})$. However, $T(G, \nu, A_{\nu\gamma})$ and $T(G, \nu^\gamma, A_{\nu\gamma})$ are both subtrees of $T(G, \pi, A)$. Therefore, if $\lambda$ is in $T(G, \pi, A)$ then $\lambda^\gamma$ is in $T(G, \pi, A)$, contradicting the assumption that $\lambda^\gamma$ is not a leaf in $T(G, \pi, A)$. Hence, the assumption that $A_{\nu\gamma} = \text{Aut}(G, \nu^\gamma)$ must be false and therefore $A_{\nu\gamma}$ is a proper subgroup of $\text{Aut}(G, \nu^\gamma)$ (it cannot be a supergroup).

By the induction hypothesis, since $A_{\nu\gamma}$ is a proper subgroup of $\text{Aut}(G, \nu^\gamma)$ and the height of $T(G, \nu^\gamma, A_{\nu\gamma})$ is at least one less than the height of $T(G, \pi, A)$, a generator is discovered in the traversal of $T(G, \nu^\gamma, A_{\nu\gamma})$, completing the proof.

\[\square\]

Theorem 4 shows that, if not all of the generators of the automorphism group are known, then the traversal of the group search tree will yield a generator. Eventually, generators of the whole group will be discovered. The third case of the proof also implies that a generator
will be discovered even when using approximate pruning via minimum cell representatives (see Section 3.5).

Let \((G, \pi)\) be a colored graph, and the sequence of group search trees examined be

\[ T(G, \pi, A_0), T(G, \pi, A_1), \ldots, T(G, \pi, A_m). \]

Here \(A_0\) is the trivial group and \(A_m = \text{Aut}(G, \pi)\). Note that it is entirely possible that \(T(G, \pi, A_i) = T(G, \pi, A_j)\) for some \(0 \leq i \neq j \leq m\). The algorithm does not restart searching at a previously seen search tree, because the guide tree allows for a graceful resumption of the search.

Ideally, the group target-cell function is chosen such that \(T(G, \pi, \text{Aut}(G, \pi))\) is much smaller than \(T(G, \pi, A_0)\) by preferring cells with few orbits. However, it is possible to use any group target-cell function, such as one that maximizes the number of orbits.

The number of group search trees examined is equal to the number of generators discovered. It could happen that many of the search trees are equivalent. In particular, it could be the case that \(T(G, \pi, A_0) = T(G, \pi, \text{Aut}(G, \pi))\) in which case all of the intermediate target-cell reevaluations are unnecessary. The next section addresses this issue by using a less active target-cell reevaluation scheme.
4.6 Lazy target-cell reevaluation

Observe from Figure 18 that only one leaf node in the guide tree contains a valid labeling. Furthermore, every other leaf node’s only use is to find a single generator; it is not used afterwards. It seems that the target-cell choices are being reevaluated too often, and that this greedy approach could be detrimental to overall performance.

The search will only be disrupted once per generator, and at most $|V|^2$ generators can be discovered [23]. In practice, there are many fewer than $|V|^2$ generators discovered. However, it is still desirable to keep disruptions to a minimum, so a lazy disruption strategy is developed in this section.

The main goal of introducing the guide tree data structure is to allow the target-cell function to make use of the automorphism group (in a permutation independent manner). The ideal choices for short runtime would be to prefer cells which have few orbits, and of those to prefer cells which contain the largest orbit. In a sense, choosing cells with small orbits can be viewed as “avoiding” cells which contain many orbits.

So, why not only check for different target-cell choices whenever it can be determined that a target cell contains more than one orbit? This is a good strategy because for most graphs, the refiner splits vertices into the automorphism partition, and restarting with each generator discovery would be wasteful. This lazy approach will only look for a different target cell choice whenever an invariant value is found not to be equivalent to the current stored invariant value.
4.6.1 An example of lazy target-cell revaluation

Consider the search tree for \((\mathcal{X}(Y_3), \vartheta_{b,3})\) using \(tc = \text{first-smallest}\) in Figure 17(x). Using the lazy strategy and \textbf{min-orbits}, target cell reevaluation will not occur until encountering the first node other than the first leaf which does not yield a generator (its invariant value is different). Figure 19 illustrates this.

![Search and guide trees](image)

Figure 19: The search and guide trees when performing lazy target cell reevaluation on \((\mathcal{X}(Y_3), \vartheta_{b,3})\) using \(tc = \text{min-orbits}\).

Observe that target cell reevaluation is triggered by encountering an invariant which is different (in this case larger) than what was previously seen. This causes the search to be restarted at the root using the full automorphism group. Up until this point, all generators have been “hidden” to the target cell chooser; they are only revealed during and after performing reevaluation.
4.6.2 Implementing lazy reevaluation

There are several difficulties associated with performing lazy reevaluation of the target cells. Two objectives are paramount: first the full automorphism group must be computed, and secondly the labeling returned must be canonical. The first requires no modification to the algorithm; whereas the second requires modification of the guide tree, as well as some additional processing after the automorphism group is found.

To see the problem with recovering the canonical labeling, consider the following scenario: throughout the search, the refiner splits all nodes into their automorphism partitions; thus only generators are discovered and all invariant values are equivalent. Assume as well that the root node contains one cell of size three and another of size two (all other cells are discrete). Using $\text{tc} = \text{min-orbits}$, the cell of size two will be chosen as the root’s target cell because no generators have been discovered and therefore the cell of size two appears to contain the fewest orbits. However, at the completion of the search it is known that the cell of size three is the first cell with the fewest orbits (illustrated in Figure 20). However, it was never examined and thus no leaf in the guide tree is valid.

![Diagram](image)

Figure 20: A complication with recovering the canonical labeling using lazy target cell reevaluation. The partition on the left shows the initial target-cell choice given no generators while the partition on the right shows the target-cell choice given all generators.

Somehow the cell of size three needs to be selected and a leaf node derived from it. This suggests that after traversing the search tree, a target cell reevaluation phase must take
place, which looks for changes in the target cell choices for leaf nodes in the guide tree. Each leaf node in the guide tree needs to be reevaluated only once and only if the number of generators visible upon its creation is less than the number of generators discovered. When each leaf node has been reevaluated with the full set of generators, the search is complete.

### 4.7 A good group target-cell function

The target-cell function used by our method is a variant of \textit{min-orbits}. Given a colored graph $(G, \pi)$ and a set of generators $\theta$, let the group $A$ be the subgroup of $\langle \theta \rangle$ which fixes $\pi$. Then the following decision process is used to select a cell:

- If $A$ is trivial then return \textit{max-joins}(\(G, \pi\)).
- Otherwise, find all cells containing an orbit of size greater than one.
- Select those which have the fewest number of orbits.
- Choose those which contain the largest orbit.
- Finally, choose the cell which is nontrivially joined to the most others.
- If at any point in this decision process, only one cell is remaining, stop and return it.

If no generators of a graph are known, this target-cell function behaves exactly like \textit{max-joins}. If a generator is known, then a cell with a nontrivial orbit should be chosen to ensure that at least some progress is made in determining the automorphism group. Of the
cells with nontrivial orbits, those with the fewest orbits should be chosen to minimize the branching factor. Of the remaining cells, those which contain the largest orbit should be chosen so that a leaf node is reached as soon as possible. Finally, if more than one cell is left, the cell which is nontrivially joined to the most others is selected.

4.8 Empirical study

It has been shown how to use partial generators of the automorphism group for target-cell computation. In the case of colored Miyazaki graphs, this allows the target-cell function to avoid cells with multiple orbits, resulting in polynomial-time processing of these graphs. This section provides empirical results demonstrating the benefits of using our method.

In the following experiments, the target-cell function used by bliss and nauty is max-joins (the default). Lazy target-cell reevaluation is used for our method nishe. The two metrics used to benchmark performance are runtime and the number of refinements performed. The number of refinements are used in place of search tree size since nishe needs to perform a few more refinements to recover the canonical labeling. For nauty and bliss, the number of refinements is equal to the search tree size. There are several optimizations used in the refiners for bliss and nauty that have not been incorporated into nishe yet (in particular, taking advantage of active indices of singleton cells), so nishe will typically be slower than bliss and nauty given the same number of refinements.
4.8.1 Experimental setup

The experiments are run on a cluster of 64 Dell PowerEdge PCs running Linux, each with a 3.0-GHz Intel Xeon processors that has 2048 KB cache and 8 GB of main memory. Each tool was compiled with the GNU C Compiler (version 4.12) using the default compilation flags for the tool. The version of nauty used is 2.47b, using the new sparse data structures. The version of bliss used is 0.50. All tools are run with their canonical labeling option turned on (as opposed to just finding generators).

4.8.2 Benchmarking on Miyazaki graphs

The most dramatic benefits of using our method are seen on colored Miyazaki graphs or graphs derived from them. The first benchmark family is the family of colored graphs $(\mathcal{X}(Y_k), \vartheta_{c,k})$ introduced in Section 4.1.

Figure 21 illustrates the exponential behavior of bliss and nauty, and the polynomial behavior of nishe on the colored graph $(\mathcal{X}(Y_k), \vartheta_{c,k})$ for several values of $k$. The data is averaged over 60 runs when the runtime is less than 1000 seconds, otherwise just one run is performed. The $c$ coloring is designed to thwart the target-cell function max-joins. Hence, nauty and bliss both exhibit exponential behavior. For nishe, each time a generator is discovered a new target-cell choice is revealed at the root level which has only one orbit. This means that, eventually, each target cell on the path from the root to the canonical leaf
node has only one orbit. Since there are $k + 1$ generators in a generating set for colored Miyazaki graphs, $nishe$ will only reevaluate the target-cells $k + 1$ times and thus process them in polynomial time. Table 2 displays the results for the $c$ colorings. Note that $bliss$ performs better than $nauty$ on this benchmark. This is most likely because $bliss$ uses a stronger node-invariant than does $nauty$ and is able to determine sooner that the leaves of a subtree must all have smaller node-invariant values than the best so far.
Figure 21: The runtimes and refines for *nauty*, *bliss*, and *nishe* on \((X(Y_k), \vartheta_{c,k})\).
Table 2: Results for the $c$ coloring of Miyazaki graphs.

The next experiment demonstrates the polynomial behavior of $nishe$ on random orderings of the the unordered partition $\vartheta_k$. This shows that our method, $nishe$, is stable on permutations of $\vartheta_k$ and will process them in polynomial time. Figure 22 displays the runtimes and refinements collected from 1024 random orderings of $\vartheta_k$ for each value of $k$. For both the runtimes and refinements, the gap between the minimum and maximum appears
to remain constant. In particular, the maximum values show clear polynomial behavior. Table 3 displays the runtime results and Table 5 displays the refinement results.

![Runtimes for Miyazaki graphs with random orderings of $\vartheta_k$](image1)

![Refinements for Miyazaki graphs with random orderings of $\vartheta_k$](image2)

Figure 22: The min and max runtimes and refines for nishe on 1024 random orderings of $\vartheta_k$. 

84
\[(\mathcal{X}(Y_k), \vartheta_k)\] runtimes for nishe

<table>
<thead>
<tr>
<th>(k)</th>
<th>vertices</th>
<th>min</th>
<th>max</th>
<th>avg</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>200</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>0.02</td>
<td>0.04</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>30</td>
<td>600</td>
<td>0.04</td>
<td>0.10</td>
<td>0.06</td>
<td>0.01</td>
</tr>
<tr>
<td>40</td>
<td>800</td>
<td>0.07</td>
<td>0.17</td>
<td>0.10</td>
<td>0.02</td>
</tr>
<tr>
<td>50</td>
<td>1000</td>
<td>0.12</td>
<td>0.29</td>
<td>0.17</td>
<td>0.03</td>
</tr>
<tr>
<td>60</td>
<td>1200</td>
<td>0.18</td>
<td>0.44</td>
<td>0.27</td>
<td>0.04</td>
</tr>
<tr>
<td>70</td>
<td>1400</td>
<td>0.24</td>
<td>0.60</td>
<td>0.39</td>
<td>0.05</td>
</tr>
<tr>
<td>80</td>
<td>1600</td>
<td>0.36</td>
<td>0.87</td>
<td>0.54</td>
<td>0.08</td>
</tr>
<tr>
<td>90</td>
<td>1800</td>
<td>0.49</td>
<td>1.13</td>
<td>0.73</td>
<td>0.10</td>
</tr>
<tr>
<td>100</td>
<td>2000</td>
<td>0.61</td>
<td>1.46</td>
<td>0.96</td>
<td>0.13</td>
</tr>
<tr>
<td>110</td>
<td>2200</td>
<td>0.83</td>
<td>1.80</td>
<td>1.24</td>
<td>0.17</td>
</tr>
<tr>
<td>120</td>
<td>2400</td>
<td>1.05</td>
<td>2.32</td>
<td>1.56</td>
<td>0.20</td>
</tr>
<tr>
<td>130</td>
<td>2600</td>
<td>1.26</td>
<td>2.72</td>
<td>1.94</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Table 3: Runtimes for nishe on random orderings of \(\vartheta_k\).

\[(\mathcal{X}(Y_k), \vartheta_k)\] refinements for nishe

<table>
<thead>
<tr>
<th>(k)</th>
<th>vertices</th>
<th>min</th>
<th>max</th>
<th>avg</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>200</td>
<td>83</td>
<td>148</td>
<td>99.28</td>
<td>10.20</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>195</td>
<td>368</td>
<td>244.28</td>
<td>27.11</td>
</tr>
<tr>
<td>30</td>
<td>600</td>
<td>336</td>
<td>669</td>
<td>427.80</td>
<td>45.47</td>
</tr>
<tr>
<td>40</td>
<td>800</td>
<td>499</td>
<td>947</td>
<td>645.46</td>
<td>70.51</td>
</tr>
<tr>
<td>50</td>
<td>1000</td>
<td>677</td>
<td>1314</td>
<td>893.09</td>
<td>93.57</td>
</tr>
<tr>
<td>60</td>
<td>1200</td>
<td>911</td>
<td>1685</td>
<td>1169.55</td>
<td>127.77</td>
</tr>
<tr>
<td>70</td>
<td>1400</td>
<td>1116</td>
<td>2038</td>
<td>1461.81</td>
<td>147.95</td>
</tr>
<tr>
<td>80</td>
<td>1600</td>
<td>1403</td>
<td>2596</td>
<td>1801.14</td>
<td>185.72</td>
</tr>
<tr>
<td>90</td>
<td>1800</td>
<td>1658</td>
<td>3063</td>
<td>2150.49</td>
<td>205.14</td>
</tr>
<tr>
<td>100</td>
<td>2000</td>
<td>1943</td>
<td>3485</td>
<td>2523.51</td>
<td>247.02</td>
</tr>
<tr>
<td>110</td>
<td>2200</td>
<td>2220</td>
<td>3887</td>
<td>2945.34</td>
<td>289.03</td>
</tr>
<tr>
<td>120</td>
<td>2400</td>
<td>2632</td>
<td>4462</td>
<td>3361.15</td>
<td>314.59</td>
</tr>
<tr>
<td>130</td>
<td>2600</td>
<td>2844</td>
<td>4943</td>
<td>3812.00</td>
<td>348.28</td>
</tr>
</tbody>
</table>

Table 4: Refinements for nishe on random orderings of \(\vartheta_k\).

Figure 23 demonstrates that the polynomial/exponential behavior is not just an artifact specific to colored graphs. It applies to uncolored graphs as well. In benchmarking bliss [44]
Juntila and Kaski applied gadgets to colored Miyazaki graphs to yield uncolored graphs which still exhibit exponential behavior. The timeout for this series is two hours and averaged over 60 random isomorphs for each run.

Figure 23: The runtimes and refines (averaged over 60 isomorphs) using \textit{nauty}, \textit{bliss}, and \textit{nishe} for \textit{mz-aug2-k}.
<table>
<thead>
<tr>
<th>$k$</th>
<th>mz-aug2-$k$ vertices</th>
<th>nishe (new) refines</th>
<th>nishe time (s)</th>
<th>bliss refines</th>
<th>bliss time (s)</th>
<th>nauty refines</th>
<th>nauty time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>96</td>
<td>55</td>
<td>0.00</td>
<td>106</td>
<td>0.00</td>
<td>109</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>144</td>
<td>91</td>
<td>0.00</td>
<td>369</td>
<td>0.00</td>
<td>400</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>135</td>
<td>0.01</td>
<td>1341</td>
<td>0.01</td>
<td>1576</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>240</td>
<td>187</td>
<td>0.01</td>
<td>5176</td>
<td>0.06</td>
<td>6124</td>
<td>0.04</td>
</tr>
<tr>
<td>12</td>
<td>288</td>
<td>247</td>
<td>0.01</td>
<td>20586</td>
<td>0.25</td>
<td>24001</td>
<td>0.19</td>
</tr>
<tr>
<td>14</td>
<td>336</td>
<td>315</td>
<td>0.02</td>
<td>82306</td>
<td>1.08</td>
<td>98247</td>
<td>0.91</td>
</tr>
<tr>
<td>16</td>
<td>384</td>
<td>391</td>
<td>0.02</td>
<td>327353</td>
<td>4.75</td>
<td>396481</td>
<td>4.19</td>
</tr>
<tr>
<td>18</td>
<td>432</td>
<td>475</td>
<td>0.03</td>
<td>1332715</td>
<td>21.06</td>
<td>1574341</td>
<td>18.97</td>
</tr>
<tr>
<td>20</td>
<td>480</td>
<td>567</td>
<td>0.04</td>
<td>5351974</td>
<td>91.17</td>
<td>6312416</td>
<td>84.74</td>
</tr>
<tr>
<td>22</td>
<td>528</td>
<td>667</td>
<td>0.05</td>
<td>21211457</td>
<td>390.24</td>
<td>26360355</td>
<td>398.01</td>
</tr>
<tr>
<td>24</td>
<td>576</td>
<td>775</td>
<td>0.06</td>
<td>85653210</td>
<td>1683.71</td>
<td>104968416</td>
<td>1721.96</td>
</tr>
<tr>
<td>26</td>
<td>624</td>
<td>891</td>
<td>0.07</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>28</td>
<td>672</td>
<td>1015</td>
<td>0.08</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>30</td>
<td>720</td>
<td>1147</td>
<td>0.10</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>32</td>
<td>768</td>
<td>1287</td>
<td>0.12</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>34</td>
<td>816</td>
<td>1435</td>
<td>0.13</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>36</td>
<td>864</td>
<td>1591</td>
<td>0.15</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>38</td>
<td>912</td>
<td>1755</td>
<td>0.18</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>40</td>
<td>960</td>
<td>1927</td>
<td>0.20</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>42</td>
<td>1008</td>
<td>2107</td>
<td>0.22</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>44</td>
<td>1056</td>
<td>2295</td>
<td>0.25</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>46</td>
<td>1104</td>
<td>2491</td>
<td>0.29</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>48</td>
<td>1152</td>
<td>2695</td>
<td>0.32</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
<tr>
<td>50</td>
<td>1200</td>
<td>2907</td>
<td>0.36</td>
<td>?</td>
<td>t.o</td>
<td>?</td>
<td>t.o</td>
</tr>
</tbody>
</table>

Table 5: Results for the mz-aug2-$k$ series.

It is notable that nishe performs the same number of refinements on each of the 60 isomorphs while the number of refinements varies for nauty and bliss. Furthermore, bliss and nauty perform very similarly in terms of runtime, while bliss performs slightly fewer refines, most likely due to its stronger node-invariant. The data suggests that, for these graphs, the time-cost of applying the stronger node-invariant used by bliss almost exactly compensates for the reduction in search tree size that it yields.
4.8.3 Benchmarking on non-Miyazaki graphs

The previous section illustrates that our method performs extremely well on colored Miyazaki graphs and uncolored graphs derived from them. However, this does not address the efficiency of our method to graphs not related to the Miyazaki construction. Our method is first compared against bliss and nauty. Note that even though our method occasionally performs many fewer refines than both bliss and nauty, its runtime is not proportionally smaller. This is most likely due to the aforementioned lack of optimization of our refinement procedure.

The most interesting benchmark timings are displayed in Table 6. The “gens” column records the number of generators discovered by each tool.

<table>
<thead>
<tr>
<th>name</th>
<th>vertices</th>
<th>edges</th>
<th>gens</th>
<th>nishe (new) refines</th>
<th>time gens</th>
<th>bliss refines</th>
<th>time gens</th>
<th>nauty refines</th>
<th>time gens</th>
</tr>
</thead>
<tbody>
<tr>
<td>pp-16-12</td>
<td>546</td>
<td>4641</td>
<td>?</td>
<td>?</td>
<td>t.o</td>
<td>6</td>
<td>5025112</td>
<td>149.12</td>
<td>7</td>
</tr>
<tr>
<td>pp-16-18</td>
<td>546</td>
<td>2320</td>
<td>18</td>
<td>36999538</td>
<td>34180.40</td>
<td>4</td>
<td>46005059</td>
<td>1427.88</td>
<td>?</td>
</tr>
<tr>
<td>pp-16-22</td>
<td>546</td>
<td>4641</td>
<td>20</td>
<td>62680538</td>
<td>6592.96</td>
<td>7</td>
<td>539781990</td>
<td>7518.75</td>
<td>7</td>
</tr>
<tr>
<td>pp-16-2</td>
<td>546</td>
<td>4641</td>
<td>17</td>
<td>34814478</td>
<td>10980.50</td>
<td>9</td>
<td>81992440</td>
<td>1391.21</td>
<td>?</td>
</tr>
<tr>
<td>pp-16-3</td>
<td>546</td>
<td>4641</td>
<td>18</td>
<td>806200</td>
<td>310.36</td>
<td>9</td>
<td>119505</td>
<td>3.34</td>
<td>?</td>
</tr>
<tr>
<td>pp-16-4</td>
<td>546</td>
<td>4641</td>
<td>28</td>
<td>989853</td>
<td>701.18</td>
<td>7</td>
<td>18774117</td>
<td>453.07</td>
<td>7</td>
</tr>
<tr>
<td>pp-16-5</td>
<td>546</td>
<td>4641</td>
<td>27</td>
<td>997712</td>
<td>704.29</td>
<td>9</td>
<td>591642</td>
<td>14.94</td>
<td>?</td>
</tr>
<tr>
<td>pp-16-8</td>
<td>546</td>
<td>4641</td>
<td>23</td>
<td>33302830</td>
<td>23218.70</td>
<td>8</td>
<td>193522537</td>
<td>3586.06</td>
<td>?</td>
</tr>
<tr>
<td>pp-9-3</td>
<td>182</td>
<td>910</td>
<td>26</td>
<td>12101</td>
<td>3.97</td>
<td>7</td>
<td>163463</td>
<td>1.58</td>
<td>7</td>
</tr>
<tr>
<td>had-184</td>
<td>736</td>
<td>34040</td>
<td>9</td>
<td>9951218</td>
<td>8406.77</td>
<td>7</td>
<td>119937</td>
<td>4.19</td>
<td>7</td>
</tr>
<tr>
<td>had-232</td>
<td>928</td>
<td>108112</td>
<td>9</td>
<td>16699083</td>
<td>20884.40</td>
<td>7</td>
<td>181668</td>
<td>97.25</td>
<td>7</td>
</tr>
<tr>
<td>had-sw-96</td>
<td>384</td>
<td>18024</td>
<td>2</td>
<td>40814013</td>
<td>12512.80</td>
<td>2</td>
<td>4643774</td>
<td>537.58</td>
<td>2</td>
</tr>
<tr>
<td>kef13</td>
<td>338</td>
<td>2197</td>
<td>35</td>
<td>325282</td>
<td>35.87</td>
<td>34</td>
<td>1626908</td>
<td>21.99</td>
<td>34</td>
</tr>
<tr>
<td>kef14</td>
<td>392</td>
<td>2744</td>
<td>16</td>
<td>85037</td>
<td>48.39</td>
<td>9</td>
<td>362858</td>
<td>11.60</td>
<td>10</td>
</tr>
<tr>
<td>kef15</td>
<td>450</td>
<td>3375</td>
<td>45</td>
<td>116620</td>
<td>502.38</td>
<td>38</td>
<td>15588024</td>
<td>230.21</td>
<td>40</td>
</tr>
<tr>
<td>kef16</td>
<td>512</td>
<td>4096</td>
<td>24</td>
<td>25789608</td>
<td>23283.00</td>
<td>10</td>
<td>44819369</td>
<td>1524.15</td>
<td>11</td>
</tr>
<tr>
<td>kef17</td>
<td>578</td>
<td>4913</td>
<td>52</td>
<td>11690874</td>
<td>3416.55</td>
<td>44</td>
<td>26508927</td>
<td>455.33</td>
<td>46</td>
</tr>
<tr>
<td>cfi-180</td>
<td>1800</td>
<td>2700</td>
<td>239</td>
<td>10006</td>
<td>8.70</td>
<td>91</td>
<td>19984</td>
<td>1.60</td>
<td>91</td>
</tr>
</tbody>
</table>

Table 6: Results comparing nishe to bliss and nauty on non-Miyazaki graphs.

While nishe occasionally performs fewer refinements than the other tools, it is never enough to offer a spectacular difference in timings. The graph which benefits the most
from target-cell reevaluation is pp-16-22, performing about eight times fewer refinements. It is clear from the generator counts that *nishe* is discovering redundant generators. This is because normally, the fixed path of the best leaf node doesn’t change much, but when reevaluating target cells, it can change a lot. This means that, unless a change of base is applied to the partial generators discovered (using computational group theory techniques), the approximate pruning heuristic will return too many false minimum cell representatives. This makes it clear that exact pruning should be used with our method to provide the most benefit.
CHAPTER 5
ADAPTIVE REFINEMENT

The preceding chapter shows how to find good target cells if there are any, but what if there are none? This means the refiner was not very effective. The purpose of a refiner is to split vertices which are in different orbits, while keeping those which are in the same orbit together. The equitable refiner \( R(G, \pi) \) does a good job in most instances. However, for any regular graph \( G \) and the unit partition \((\text{V})\) (all vertices colored the same), equitable refinement does nothing. Since most regular graphs are rigid [50], the equitable refiner will fail in its job of splitting vertices on most regular graphs.

One solution is to use a stronger vertex-invariant to split more cells. This was the path taken by McKay with nauty. In nauty, there are fifteen different vertex-invariants to choose from. When using a vertex-invariant, the canonical labeling could be affected, so when comparing labelings with another graph, the same vertex-invariant must be applied. This poses no problem when detecting symmetries.

There are some pitfalls associated with using vertex-invariants. The vertex-invariant could be too weak and split no cells. Or, the equitable refiner could already split the cells into the automorphism partition, in which case applying the vertex-invariant just incurs extra work. On the positive side, if a split is found by a vertex-invariant, then even more splits can be found with an equitable refiner. The more splits found, the shorter the search. From
experience with *nauty*, the use of a vertex-invariant can mean the difference between days of computation and mere seconds (if the vertex-invariant finds a split) on difficult graphs.

One question that arises with using vertex-invariants is when to apply them. The time required to run the vertex-invariant could be much higher than the time required by *nauty* without using the vertex-invariant. As McKay expressed it in *nauty*’s user manual [64]:

> A great number of vertex-invariants have been proposed in the literature, but very few of them are suitable for use with nauty. Most of them are either insufficiently powerful or require excessive amounts of time or space to compute. Even amongst the vertex-invariants which are known to be useful, their usefulness varies so much with the type of graph they are applied to, or the levels of the search tree at which they are applied, that intelligent automatic selection of a vertex-invariant by nauty would seem to be a task beyond our current capabilities. Consequently, the choice of vertex-invariant (or the choice not to use one) has been left up to the user.

In this chapter, a method is developed which allows for using a stronger refiner, but only when needed. This refiner is based on node-invariants. It is similar to a technique mentioned in [80] as well as the multirefinement technique introduced by Piperno in [72].
5.1 Taking advantage of node-invariants

Recall that a node-invariant is a function $I(G,\pi,\nu)$ which takes in a colored graph $(G,\pi)$ and a search node $\nu \in T(G,\pi)$ and outputs an element of a totally ordered set. The only node-invariant seen thus far is \textit{weakest-invar}, which only has an effect when $\nu$ is a leaf node.

A stronger invariant, which returns the list of cell sizes of $\nu = (W_1, W_2, \ldots, W_k)$ is defined as

$$\text{shape-invar}(G,\pi,\nu) = (|W_1|, |W_2|, \ldots, |W_k|).$$

The node-invariant \textit{shape-invar} can be turned into a leaf-certificate by defining for each leaf node $\lambda \in T(G,\pi)$, $\text{shape-invar}(G,\pi,\lambda) = \text{weakest-invar}(G,\pi,\lambda)$. The ordering of the list of sizes is lexicographic, and graphs are defined to be smaller than lists of sizes.

The graph which has the fewest edges and is connected for which \textit{shape-invar} has an effect is shown in Figure 24. The orbits of the graph are $\{a, c, d, f\}$, $\{b, e\}$, and $\{g, h\}$ and it has eight automorphisms.

![Figure 24: The smallest (in terms of edges) connected graph for which \textit{shape-invar} has an effect.](image)
The search tree for the graph in Figure 24 is shown in Figure 25. It uses $I = \text{shape-invar}$ and $tc = \text{first}$, where

$$\text{first}(G, \pi) = \text{the index of the first nontrivial cell of } \pi.$$
a vertex-invariant (applicable only to search nodes) $I_I(G, \pi, \nu, u)$ can be defined by

$$I_I(G, \pi, \nu, u) = I(G, \pi, R(G, \nu \vdash u))$$

This provides a method of distinguishing between vertices in the same cell of the node $\nu$. Furthermore, since the output of the node-invariant is ordered, the vertices in a cell can be sorted and then split where the vertex-invariant values differ.

For example, consider the tree in Figure 25 with $\nu$ equal to the root node. The node-invariants for the elements of $\nu$’s target cell are

$$I_{\text{shape-invar}}(G, \pi, \nu, a) = (1, 1, 1, 2, 1, 1, 1)$$
$$I_{\text{shape-invar}}(G, \pi, \nu, b) = (1, 2, 2, 1, 1, 1)$$
$$I_{\text{shape-invar}}(G, \pi, \nu, c) = (1, 1, 1, 2, 1, 1, 1)$$
$$I_{\text{shape-invar}}(G, \pi, \nu, d) = (1, 1, 1, 2, 1, 1, 1)$$
$$I_{\text{shape-invar}}(G, \pi, \nu, e) = (1, 2, 2, 1, 1, 1)$$
$$I_{\text{shape-invar}}(G, \pi, \nu, f) = (1, 1, 1, 2, 1, 1, 1)$$

Thus, $b$ and $e$ are not in the same orbit as $a, c, d,$ or $f$. This demonstrates how node-invariants can be used to reveal splits that were invisible to the equitable refiner. After splitting the
target cell, the resulting partition might not be equitable, and refining it could reveal even more splits (it often does).

Take, for example, the graph in Figure 24 and its search tree in Figure 25. After determining that the 2nd child of the root node has a different shape than the first child, the target cell is split via the node-invariant and a new node at the same level is created. Figure 26 illustrates the resultant search tree while Figure 27 illustrates the guide tree (whose construction will be explained in the next section).

![Diagram of search tree](image)

Figure 26: The modified search tree for the graph in Figure 24. The root node has a node-invariant refined child at the same level.
Figure 27: The guide tree for the search tree in Figure 26. The zig-zag line represents a refinement of the target cell via node-invariants. The bold line represents the path for the canonical labeling. Observe that the zig-zag line must be used, since it represents the path chosen by the stronger refiner.

5.2 Implementation

Using node-invariants to split the elements of the target cell effectively uses a stronger refiner than an equitable one, called a node-invariant refiner. The refiner is defined in terms of $I(G, \pi, \nu, v)$ and the function $\text{split}(\pi, f)$, which sorts the cells of $\pi$ based on their output in $f$ and splits on the boundaries. Let $k$ be the target-cell choice, then $f$ is defined as

$$f(u) = \begin{cases} I(G, \pi, \nu, u) & \text{if } u \in \nu[k] \\ 0 & \text{otherwise} \end{cases}$$

The node-invariant refiner $R_I(G, \nu)$ (where $\nu$ is a search node in $T(G, \pi)$) is then

$$R_I(G, \nu) = R(G, \text{split}(\nu, f)).$$
The challenge is how to implement node-invariant refinement in practice. It could be performed on every target cell of every search node, as is done in [72]. This has the drawback of being very expensive, in that not just the target cell, but every nontrivial cell is examined for splits. This strategy yields good results for instances that need node-invariant refinement, but for graphs that would not benefit from node-invariant refinement, the strategy needlessly looks for splits where the node-invariant is too weak to reveal any.

**Definition** (directly impure). A cell $W$ of a search node $\nu$ in the search tree $T(G, \pi)$ is called *directly impure* with respect to the node-invariant $I$ if there exist two vertices $u, v \in W$ such that $I(G, \pi, \nu, u) \neq I(G, \pi, \nu, v)$.

**Definition** (impure, pure). A cell $W$ of a search node $\nu$ in the search tree $T(G, \pi)$ is called *impure* if it contains more than one orbit. Cells containing only one orbit are called *pure*.

The previous strategy implicitly hopes that each cell of every search node is directly impure. An alternative strategy keeps track of node-invariant values for elements in the target cell of each search node. Let $\nu$ be a search node with target cell $k$ and $u, v \in \nu[k]$ with $u < v$. Assume that $T(G, \pi, \nu \vdash u)$ is traversed and that $I_I(G, \pi, \nu, v)$ has just been calculated. If $I_I(G, \pi, \nu, u) \neq I_I(G, \pi, \nu, v)$, then $\nu$’s target cell is directly impure with respect to $I$ and should be node-invariant refined (and it is only node-invariant refined if it is found to be directly impure). This has the advantage that node-invariant refinement is only performed when it is guaranteed to yield results. A possible drawback is that all of $T(G, \pi, \nu \vdash u)$ is traversed. This could be a lengthy computation without first node-invariant refining $\nu$. 

97
A good compromise between the two extremes makes use of the guide tree. Whenever it is found that a search node \( \nu \) has a target cell \( k \) which is directly impure, this fact is recorded in the \( k \)-child of the current guide node. The target cell is then node-invariant refined and the current guide node is set to a special child; the invariant-refined child. When the next search node reaches the guide node that has an invariant-refined child, a node-invariant refinement is attempted. This strategy is called *adaptive refinement*.

The following data field is added to the guide tree:

- **invar_refine_child** - empty if the parent of this node never encountered two different node-invariants in its target cell. Otherwise, **invar_refine_child** is the guide node which contains the relevant data for the invariant-refined child.

So, whenever a guide node which has a nonempty **invar_refine_child** field is reached, the target cell of its parent’s partition is node-invariant refined.

Using node-invariant refinement means that a node’s fixed path is no longer unique. The information of which target cell has been used for node-invariant refinement should somehow be stored in the fixed path as well. If \( k \) is the target cell, then store \(- (1 + k)\) when moving to a guide node’s **invar_refine_child**. For example, the uniquely identifying fixed path of the rightmost leaf node in Figure 26 is \((-1, 0, 1)\).
5.3 Runtime considerations

To perform full node-invariant refinement, the node-invariant of each element in the target cell is calculated and then used as a key to sort its elements. To avoid sorting, and also any space overhead that might be required by storing node-invariant values, only keep track of a single split. For instance, assume that a search node $\nu$'s target cell $W$ is directly impure with respect to $I$ and that node-invariant refinement would reveal several splits. Then let $S = \{u \in W : I_I(G, \pi, \nu, u) \text{ is minimal}\}$. Splitting the cell $W$ into $S, W \setminus S$ and then refining could reveal all of the splits that would result from full sorting (and usually many more in other cells).

**Definition.** Using only the split between the smallest set of node-invariant values and the larger ones is called *min node-invariant refinement*. This requires no sorting and little space overhead.

**Definition.** Using every split revealed by the node-invariant values is called *full node-invariant refinement*. This incurs additional space overhead as well as a sorting round.

It is also possible to reduce the amount of work needed by using previously discovered automorphisms. If two elements of the directly impure cell are in the same orbit, then only one of their invariant values should be calculated. That is, if there are elements $u, v \in W$ where $W$ is a cell in a search node $\nu$, then if there exists a $\gamma$ generated by the set of generators $\theta$ such that $u^\gamma = v$, then $I_I(G, \pi, \nu, u) = I_I(G, \pi, \nu, v)$. Only the elements which are the minimum in their orbits need to be considered when performing node-invariant refinement.
5.4 A refinement-trace invariant

The strength of node-invariant refinement is proportional to the strength of the node-invariant. The node-invariant used by early versions of \textit{nauty} is essentially \textit{shape-invar}. A stronger invariant called \textit{singleton-invar} uses the adjacency information of singleton cells. Let \( \nu' \) be the parent of \( \nu \) in the search tree (\( \nu' = \pi \) if \( \nu \) has no parent). Then define

\[
\text{singleton-invar}(G, \pi, \nu) = \begin{cases} 
(k, j) : & \{u\} \text{ is a trivial cell in } \nu \text{ but not in } \nu' \text{ and} \\
 & \{v\} \text{ is a trivial cell in } \nu \text{ and} \\
 & \{u, v\} \in E(G) \text{ and} \\
 & (k, j) \text{ are the indices of cells } \{u\} \text{ and } \{v\} 
\end{cases}
\]

This node-invariant fills in the rows of the canonical representative’s adjacency matrix as they are found. The algorithm \textit{bliss} uses a variant of \textit{singleton-invar} along with a hash of refinement information.

A slightly stronger invariant uses not only the singleton cells, but also the nontrivial cells. It is defined as

\[
\text{multigraph-invar}(G, \pi, \nu) = \begin{cases} 
(k, j, m) : & W_1 \text{ is a cell in } \nu \text{ but not in } \nu', \\
 & W_2 \text{ is a cell in } \nu, \\
 & m (\text{greater than 0}) \text{ is the number of edges} \\
 & \text{from } W_1 \text{ to } W_2, \text{ and} \\
 & (k, j) \text{ are the indices of cells } W_1 \text{ and } W_2 
\end{cases}
\]
This dissertation introduces a stronger invariant than \texttt{multigraph-invar} which also uses refinement information explicitly to ensure that the invariant is a leaf certificate. Using information from the refiner in the form of a hash has been used in \textit{nauty} and \textit{bliss}, and explicitly in [72]. However, this information is always augmented with enough adjacency information to form a leaf certificate.

The refinement-trace invariant is defined via pseudocode in Subroutine 13. It returns a function which maps active indices to affected cells and their degree within the affected cells. The adjacency information is interleaved with the refinement information and is a full trace of what occurred during the refinement operation (in a vertex-labeling independent manner). Not every invariant calculation must be carried out fully. Only invariant-values which are smaller than or equal to the current best need to be calculated fully. The invariant-values which are larger are only partially calculated. This is an idea introduced with \textit{bliss}.

\section{Empirical study}

As in the preceding chapter, the experiments are run on a cluster of 64 Dell PowerEdge PCs running Linux, each with a 3.0-GHz Intel Xeon processors that has 2048 KB cache and 8 GB of main memory. Our tool \texttt{nishe} is run in three different configurations. The first does not use node-invariant refinement. The second uses min node-invariant refinement, and the third uses full node-invariant refinement. Table 7 lists the results.
**Subroutine 13 A refinement-trace invariant**

**Input:** colored graph \((G, \pi)\) and a search node \(\nu\)

**Output:** a mapping, \(\text{trace}\), between indices and lists of (index, degree) pairs

\[ f: V \rightarrow \mathbb{N} \]

\(\text{active} = \emptyset\)

\(\text{affected} = \emptyset\)

\(\text{trace} = \) mapping between indices and lists of (index, degree) pairs

\[
\text{if} \ (\nu \text{ has a parent } \nu') \ \text{then} \\
\quad \text{active} = \{\text{the index of } b \text{ in } \nu'\} \\
\quad \pi = \mu \vdash b \\
\text{else} \\
\quad \text{active} = \text{indices}(\pi)
\]

**while** \((\text{active} \neq \emptyset)\) **do**

\(k = \min(\text{active})\)

\(\text{active} = \text{active} \setminus \{k\}\)

\(\text{trace}(k) = \) an empty list

\[
f(u) = \deg_G(u, \pi[k]) \text{ for all } u \in V \\
\pi' = \text{split}(\pi, f)
\]

\(\text{active} = \text{active} \cup (\text{indices}(\pi') \setminus \text{indices}(\pi))\)

let \(u \in \pi'[k]\)

\(\text{affected} = \{j : \deg(u, \pi'[j]) > 0 \text{ for all } j \in \text{indices}(\pi')\}\)

**for** \((0 \leq j < n \text{ such that } j \in \text{affected})\) **do**

\(m = \deg(u, \pi[j])\)

\(\text{trace}(k).\text{append}( (j, m) )\)

\(\pi = \pi'\)

**return** \(\text{trace}\)

Observe that for some graphs, the reduction in runtime is enormous. This most likely indicates that a split was detected at an early level. However, in some cases the runtime increases, indicating that splits were only determined in the deepest levels of the search tree. The series \texttt{kef} benefits the most from adaptive refinement, while some of the graphs derived
| name    | |V| | |E| | nishe | |E| | nishe | |E| | nishe | |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| pp-9-4  | 182 | 910 | 1156212 | 362.74 | 100803 | 5.18 | 229183 | 38.70 |
| pp-16-4 | 546 | 4641 | 989853 | 701.18 | 533959 | 54.53 | 1124499 | 713.87 |
| pp-16-5 | 546 | 4641 | 997712 | 704.29 | 612801 | 61.99 | 1529928 | 958.66 |
| pp-16-7 | 546 | 4641 | 40756484 | 27993.00 | 38600546 | 3563.70 | ? | t.o |
| pp-16-14 | 546 | 4641 | 775120381 | 63487.80 | 394118766 | 34942.40 | ? | t.o |
| pp-16-18 | 546 | 4641 | 369993538 | 34180.40 | 101238593 | 9059.82 | ? | t.o |
| pp-16-22 | 546 | 4641 | 62668038 | 2654.14 | 30222661 | 2654.14 | ? | t.o |
| kef12   | 288 | 1728 | 42650 | 16.46 | 1597 | 0.14 | 1546 | 0.25 |
| kef13   | 338 | 2197 | 325282 | 35.87 | 2809 | 0.22 | 3645 | 0.53 |
| kef14   | 392 | 2744 | 85037 | 48.39 | 2452 | 0.26 | 2099 | 0.44 |
| kef15   | 450 | 3375 | 110520 | 502.38 | 4600 | 0.66 | 3869 | 0.79 |
| kef16   | 512 | 4096 | 25789608 | 23283.00 | 2938 | 0.39 | 2748 | 0.75 |
| kef17   | 578 | 4913 | 11690874 | 3416.55 | 5621 | 0.72 | 6182 | 1.50 |
| kef18   | 648 | 5832 | 23231642 | 25616.90 | 2716 | 0.41 | 3473 | 1.24 |
| had-sw-56 | 224 | 6384 | 5022336 | 681.06 | 14239474 | 3937.35 | 15020726 | 7663.22 |
| had-sw-64 | 256 | 8320 | 10234586 | 2475.55 | 23912474 | 9588.59 | 11787099 | 9037.45 |
| had-sw-88 | 352 | 15664 | 25675337 | 6762.53 | 5726959 | 38341.40 | ? | t.o |
| had-sw-96 | 384 | 18624 | 40814013 | 12512.80 | 25750060 | 36887.00 | ? | t.o |
| had-sw-112 | 448 | 25312 | 32528261 | 9145.42 | 11076166 | 2990.56 | ? | t.o |
| had-92 | 368 | 17112 | 1302875 | 353.02 | 448234 | 91.21 | 1717716 | 1205.53 |
| had-100 | 400 | 20200 | 2164994 | 622.08 | 605656 | 136.32 | 2188089 | 1722.82 |
| had-116 | 464 | 27144 | 3174500 | 1122.81 | 1725315 | 490.73 | 3334673 | 3212.62 |
| had-184 | 736 | 68080 | 9951218 | 8406.77 | 12649715 | 11452.30 | ? | t.o |
| had-232 | 928 | 108112 | 16690803 | 20884.40 | 23720288 | 30982.90 | ? | t.o |

Table 7: Results for adaptive refinement.

from Hadamard matrices (with the had prefix) incur a penalty by using adaptive refinement. Furthermore, the use of full node-invariant refinement is rarely worth the additional effort, signifying that minimum node-invariant refinement is a good strategy.
CHAPTER 6
SHORT-CIRCUITING GENERATOR DISCOVERY

It was observed in a recent paper by the authors of *saucy* that many of the generators discovered by symmetry-detection tools only move a few of the vertices in a graph (in relation to the total number of vertices) [24]. That is, the generators are sparse. They showed how to take advantage of this sparsity and effectively short-circuit generator discovery. This chapter explains this method and how to implement it within the context of guide trees.

6.1 The motivation

Consider the graph in Figure 28. It has 7 vertices, 9 edges, and 36 automorphisms.

![Graph Diagram]

Figure 28: A graph for illustrating generator discovery short circuiting.

The graph’s search tree is displayed in Figure 29 (using tc = first). The fixed path for the first leaf is \((e, f, a, c)\). The node with fixed path \((e, f)\), \(\nu_1\), has the same nontrivial cell (at the same index) as the node with fixed path \((e, g)\), \(\nu_2\). If some descendant of \(\nu_2\) yields a
generator $\gamma$, then $\{a, c, d\}^\gamma = \{a, c, d\}$. Assuming generators are sparse implies that $\gamma$ fixes each element ($a^\gamma = a$, $c^\gamma = c$, and $d^\gamma = d$). Based on this assumption, the permutation which takes $\nu_1$ to $\nu_2$ is $(f \; g)$. It can be checked that $(f \; g)$ is indeed an automorphism, and the search resumed at the greatest common ancestor of $\nu_1$ and $\nu_2$. In this case, no descendant of $\nu_2$ is traversed. This technique will be called *short-circuiting generator discovery* in this dissertation.

Figure 29: The search tree for the graph in Figure 28
6.2 Singleton permutations

**Definition** (singleton permutation). Given two partitions \( \pi_1, \pi_2 \in \Pi(V) \), the *singleton permutation* from \( \pi_1 \) to \( \pi_2 \) maps trivial cells (singletons) from \( \pi_1 \) to singletons of \( \pi_2 \) (if they have the same index) and vice versa. Specifically, the cycle \((u_1 \ u_2 \ \cdots \ u_m)\) is in the singleton permutation from \( \pi_1 \) to \( \pi_2 \) if and only if

(i) \( \{u_i\}_{1 \leq i \leq m} \) is a singleton cell in both \( \pi_1 \) and \( \pi_2 \) and

(ii) the index of \( \{u_i\} \) in \( \pi_1 \) is equal to the index of \( \{u_{i+1}\} \) (modulo \( m \)) in \( \pi_2 \) for \( 1 \leq i \leq m \).

Observe that the singleton permutation from \( \pi_1 \) to \( \pi_2 \) is the inverse of the singleton permutation from \( \pi_2 \) to \( \pi_1 \). In the case that no cycle can be completed, the singleton permutation is trivial (the identity). As an example of singleton permutations, let

\[
\pi_1 = \begin{array}{cccccc}
a & g & d & e & b & f \\
\end{array}
\]

\[
\pi_2 = \begin{array}{cccccc}
a & g & e & c & b & f \\
\end{array}
\]

\[
\pi_3 = \begin{array}{cccccc}
a & e & g & d & b & f \\
\end{array}
\]

\[
\pi_4 = \begin{array}{cccccc}
g & a & c & e & b & f \\
\end{array}
\]

The the singleton permutation from \( \pi_1 \) to \( \pi_2 \) is \((d \ e \ c)\), from \( \pi_2 \) to \( \pi_1 \) is \((e \ d \ c)\), from \( \pi_3 \) to \( \pi_i \) for \( 1 \leq i \leq 4 \) is the identity permutation because no cycle can be completed, from \( \pi_1 \) to \( \pi_4 \) is \((d \ c)\), and from \( \pi_2 \) to \( \pi_4 \) is \((e \ c)\).
If a search node \( \nu \) is generated and passes the node-invariant test (it is equivalent to what is expected in the guide tree), then it can be tested for a short-circuiting generator. Let \( \lambda \) be a leaf node. If the singleton permutation from \( \nu \) to \( \lambda \) is nontrivial, then it should be tested to see if it is an automorphism. If it is, then the search can be resumed at the greatest common ancestor of \( \nu \) and \( \lambda \). If not, the search continues as normal. Figure 30 shows the tree for the graph in Figure 28 when performing generator short-circuiting. Two nodes whose singleton permutations were nontrivial yielded generators, resulting in 4 fewer nodes traversed.

The savings from short-circuiting generator discovery can be great. For instance, the complete graph on \( n \) vertices, \( K_n \), normally has a search tree of size \( n \cdot (n + 1)/2 \). However, when short-circuiting generator discovery, the search tree only has \( 3n - 3 \) nodes (for \( n > 1 \)) resulting in an order of magnitude reduction in the search tree size.

### 6.3 Implementation using a guide tree

In the context of a guide tree, any smallest search-leaf on a guide-leaf descendant of the current guide node could be used to discover a generator. A simple and effective solution for generating these singleton permutations is to loop over each leaf guide-node descended from the current guide node and check the singleton permutation from the current to each leaf. That is, given a node \( \nu \) and a leaf \( \lambda \), if the singleton permutation \( \gamma \) from \( \nu \) to \( \lambda \) is an automorphism then no children of \( \nu \) need to be generated.
Figure 30: The search tree for the graph in Figure 28 when short-circuiting generator discovery.

The benefits of using this technique are that vast portions of the search tree can be pruned, since most generators are sparse. The main drawback is that too many automorphism tests could occur between the current node and leaf nodes which are not equivalent. Another drawback is that most generators could be dense, in which case the singleton permutation creation is wasted effort.
CHAPTER 7
A PARALLEL GRAPH ISOMORPHISM ALGORITHM

It is sometimes the case that none of the aforementioned techniques helps to process a graph faster. For instance, the graph could have no automorphisms, obviating the main benefits garnered by using the guide tree. Node-invariant refinement could only detect splits at the lowest level in the search, but be unable to detect impurities at higher levels. Short-circuiting generator discovery when there are not many automorphisms in the graph provides no benefit, and could even be a detriment by checking if a permutation is an automorphism too frequently.

Graphs exhibiting this kind of behavior usually come from the realm of combinatorics, and are derived from objects such as Hadamard matrices or projective planes. They typically have small automorphism groups, and are highly regular in the sense that equitable refinement and node-invariant refinement are too weak to split vertices not in the same orbit. Currently, there are no methods which can be applied to process these graphs faster. Node-invariant refinement offers the best hope, but it is not always effective. On the opposite extreme, graphs with large automorphism groups and large numbers of vertices and edges can also be time consuming even when short-circuiting generator discovery.

In this chapter we develop a parallel algorithm for canonically labeling a graph and detecting its symmetries. This fills a currently empty niche in the parallel realm, particularly in the realm of symmetry detection.
For instance, SAT solving requires the generators of the automorphism group of a graph derived from CNF formula to create symmetry-breaking predicates which vastly speed up the search [6]. Symmetry-detection can account for more than half of the amount of time required to solve SAT instances using this technique [23]. Any parallel SAT solver desiring to make use of symmetry-breaking predicates could use a parallel symmetry detector to achieve additional speedup. Furthermore, with the prevalence of multi-core machines and the cheap cost of clusters, a parallel graph isomorphism algorithm is needed to make use of the wealth of computing power that is available.

7.1 Design

The goal of the parallel algorithm is to return a canonical labeling and generators of the automorphism group faster than the sequential algorithm can. The search tree can be decomposed amongst each process, which then traverses a subtree of the whole tree.

There are a few parallel backtracking frameworks available, notably ZRAM [16] and BkFr [53]. While not used directly, they have influenced the design of how to divide work and manage worker processes. The code is developed in C++ using the message passing framework MPI. See [73] for an introduction to MPI. The algorithm only uses point-to-point communication.

Given $p$ processes with ranks $0, 1, \ldots, p - 1$, the process with rank 0 is the master and processes $1, \ldots, p - 1$ are the workers. Each worker will traverse a subtree specified by an
initial path (the term path will abbreviate fixed path). When a worker is finished, it reports back to the master to request more work. The master refers idle workers to busy workers so that busy workers may share a portion of their work. When no worker is busy, the algorithm is finished and a canonical labeling as well as generators of the graph’s automorphism group are returned.

Each worker stores a guide tree and a list of generators as in the sequential algorithm. The guide tree serves the same purpose in the parallel algorithm and aids in directing the search in the event of changing target-cell values and node-invariant values. The set of generators is used to prune the search and assist with target-cell choosing. It is also used to determine if a worker’s subtree is generated already, in which case its work is finished and it becomes idle again.

Other than messages coordinating work, the only information that needs to be communicated is the discovery of a generator and the discovery of a smaller canonical labeling at a leaf node in the guide tree. If a generator is discovered, it is broadcast (asynchronously) to each process.

If a leaf is found with a smaller certificate than what is stored in the guide tree, then the path of this leaf is broadcast to all other workers. Each worker then traverses to this leaf, comparing node-invariant values along the way. If this leaf certificate is also found to be smaller than the leaf certificate stored in the local guide tree, the guide tree is updated. If the leaf certificate is found to be equivalent to what is stored in the guide tree, a generator is discovered and broadcast. If it is found to be worse, no action is taken and the broadcast
is not propagated. The broadcast trees are created explicitly to ensure communication takes place in an asynchronous manner.

### 7.2 Communication

When a worker process is traversing its subtree, there are two main events that trigger communication. First, a generator could be discovered. This generator is potentially useful to other processes because it allows them to prune more of their search tree, as well as make better target-cell decisions if a group target-cell function is used. Secondly, a better candidate leaf for the canonical labeling could be found. Each process should be informed of this leaf node’s fixed path so that it can update its guide tree with the better labeling.

In both of these instances, every process needs the information; ideally as soon as possible so that it can be put to use. To accomplish this goal, the data is sent using an explicitly created broadcast tree as opposed to using the MPI function `MPI_Bcast()`. Creating the broadcast tree explicitly has several advantages.

First, messages can be sent asynchronously so that no process has to wait on another. Secondly, not all broadcasts need to be completed in their entirety. There are a few cases when a broadcast should be cut short.

The next two subsections detail when each communication event occurs and how it is handled. Then, the mechanism for load balancing is explained.
7.2.1 Generator discovery

A generator of a colored graph \((G, \pi)\) with \(n\) vertices can be represented by an array of length \(n\). However, generators usually do not move many of the \(n\) elements; most are fixed. So, when broadcasting a generator, it is first factored into its disjoint cycle representation and then this is broadcast.

Let \(\theta\) be the set of generators already discovered. When receiving a generator \(\gamma\), there are two cases a process must check:

1. The incoming generator \(\gamma\) is already generated by the elements of \(\theta\) – This could happen if it has received \(\gamma\) (or generators for \(\gamma\)) from another process, or it could have discovered generators for \(\gamma\) on its own. In this case, the broadcast is not continued.

2. The incoming generator \(\gamma\) is not already generated by the elements of \(\theta\) – In this case, \(\gamma\) is added to \(\theta\) and the broadcast continues.

A generator can be discovered in one of two ways. The first way is through a normal traversal of the subtree a worker process has examined. The second way is through receiving a canonical path representing a canonical labeling from another process which is equivalent to a canonical labeling already present in the guide tree. In this case, another generator is discovered and broadcast.
7.2.2 Canonical path updates

Whenever a process updates a section of its guide tree with a better canonical path, this new canonical path should be broadcast to every other process. This ensures that at the end of the search that every process’s local guide tree will have the best overall canonical labeling in it, and that all of the generators of the automorphism group are discovered.

When receiving a better canonical path \((u_1, u_2, \ldots, u_\ell)\) for a node \(\lambda\), the receiving process begins traversing the \(m\) search nodes towards \(\lambda\). The traversal is stopped if an invariant value disagrees with what is currently stored in the guide tree. There are three possibilities:

1. The traversal stopped at a leaf in the guide tree – In this case, \(\lambda\) is equivalent to a leaf node that has already been traversed and placed in the guide tree, say \(\lambda^\gamma\). If \(\gamma\) is not already generated, then it is broadcast to the other processes.

2. The traversal stopped because a node on the path to \(\lambda\) has a smaller invariant value than is present in the guide tree – The traversal to \(\lambda\) should be continued and the guide tree updated. The broadcast of the canonical path should be continued. Furthermore, if the process is working on a subtree, then its search should be backed up to the level of the invariant difference.

3. The traversal stopped because a node on the path to \(\lambda\) has a larger invariant value – In this case the broadcast is not continued because this process has already sent out, or received a smaller invariant value. The guide tree remains unchanged.
7.2.3 Load balancing

At the start of the parallel algorithm, the master process calculates the target-cell of the root node of the search tree while the rest of the processes remain idle. Each vertex in the target-cell is sent as the initial path to as many idle processes as possible.

A process can become idle by completing its portion of the search tree or by determining that its portion is redundant by receiving a generator. In the ideal case, all processes will be active and working on a portion of the search tree. Load balancing is accomplished as follows:

1. When a worker becomes idle, it informs the master.

2. If there are no working workers, stay idle.

3. Otherwise, the master sends the id of the newly idle worker to a currently working worker.

4. The working worker receives the id and looks for an unused branch in the most shallow target cell and sends it to the newly idle worker.

5. If no unused branches are found, no more spilt requests are made of this working worker, and the idle worker’s id is referred to a different working worker.
7.3 Empirical study

The experiments are run on a cluster of 64 Dell PowerEdge PCs running Linux, each with a 3.0-GHz Intel Xeon processors that has 2048 KB cache and 8 GB of main memory (the same machines as in Chapter 4). Because the parallel algorithm does not yet work with group target-cell functions, the target-cell function \texttt{max-joins} is used for all experiments. Table 8 lists the graphs and associated runtime statistics for the graphs chosen for benchmarking the parallel algorithm.

| name     | $|V|$ | $|E|$ | $|\text{Aut}|$ | gens | refines       | time   |
|----------|-----|-----|-------|------|---------------|--------|
| had-sw-96| 384 | 18624| 4     | 2    | 21089877      | 6639.2 |
| had-sw-112| 448 | 25312| 2     | 1    | 32528647      | 4391.8 |
| had-188  | 752 | 71064| 2     | 1    | 52515187      | 19472.9|
| had-236  | 944 | 111864| 2     | 1    | 104246566     | 58243.2|
| pp-16-17 | 944 | 4641 | 3456  | 5    | 949164808     | 49929.9|
| pp-16-18 | 944 | 4641 | 3840  | 12   | 613870168     | 32755.0|
| pp-16-22 | 944 | 4641 | 18432 | 12   | 291492334     | 15676.5|

Table 8: Sequential results for the benchmark graphs using the target-cell function \texttt{max-joins}.

Tables 9 and 10 show the refinements, runtime (in seconds), and speedup for each benchmark graph with several values of $p$ (the number of processes). The data for each value of $p$ is averaged over 30 runs.
| 20 | avg 19474016.9 | 334.3 | 19.9 |
|    | std 617637.8  | 12.0  | 0.7  |
| 40 | avg 18909994.1 | 159.0 | 42.3 |
|    | std 2147710.0 | 19.8  | 4.8  |
| 60 | avg 19986377.1 | 112.4 | 59.1 |
|    | std 521015.9  | 3.5   | 1.8  |
| 80 | avg 23420046.2 | 99.1  | 67.0 |
|    | std 413765.2  | 2.4   | 1.6  |
| 100| avg 22958960.0 | 76.9  | 86.6 |
|    | std 1117887.2 | 4.3   | 4.5  |

(a) had-sw-96

| 20 | avg 32470041.5 | 252.7 | 17.4 |
|    | std 2189.1     | 0.8   | 0.1  |
| 40 | avg 33329264.9 | 128.6 | 34.2 |
|    | std 461117.2   | 2.0   | 0.5  |
| 60 | avg 33623670.9 | 87.1  | 50.5 |
|    | std 618948.9   | 1.6   | 0.9  |
| 80 | avg 32960728.2 | 64.9  | 67.7 |
|    | std 665250.6   | 1.6   | 1.6  |
| 100| avg 33863044.2 | 54.7  | 80.4 |
|    | std 1153132.6 | 2.3   | 3.3  |

(b) had-sw-112

| 20 | avg 104429605.9 | 3122.1 | 18.7 |
|    | std 119157.5    | 4.2    | 0.0  |
| 40 | avg 105858432.5 | 1549.4 | 37.6 |
|    | std 320113.6    | 4.4    | 0.1  |
| 60 | avg 105401497.3 | 1024.5 | 56.9 |
|    | std 394361.8    | 4.0    | 0.2  |
| 80 | avg 106666390.3 | 777.4  | 74.9 |
|    | std 553757.3    | 4.0    | 0.4  |
| 100| avg 106187253.2 | 618.5  | 94.2 |
|    | std 715398.5    | 3.8    | 0.6  |

(c) had-188

| 20 | avg 52539064.8 | 1063.5 | 18.3 |
|    | std 94487.0    | 2.4    | 0.0  |
| 40 | avg 52917134.9 | 526.0  | 37.0 |
|    | std 193410.5   | 2.5    | 0.2  |
| 60 | avg 53425977.9 | 353.5  | 55.1 |
|    | std 274112.3   | 2.0    | 0.3  |
| 80 | avg 53694032.8 | 267.5  | 72.8 |
|    | std 356179.1   | 2.1    | 0.6  |
| 100| avg 53206542.3 | 212.8  | 91.5 |
|    | std 213804.7   | 1.0    | 0.4  |

(d) had-236

| 20 | avg 32470041.5 | 252.7 | 17.4 |
|    | std 2189.1     | 0.8   | 0.1  |
| 40 | avg 33329264.9 | 128.6 | 34.2 |
|    | std 461117.2   | 2.0   | 0.5  |
| 60 | avg 33623670.9 | 87.1  | 50.5 |
|    | std 618948.9   | 1.6   | 0.9  |
| 80 | avg 32960728.2 | 64.9  | 67.7 |
|    | std 665250.6   | 1.6   | 1.6  |
| 100| avg 33863044.2 | 54.7  | 80.4 |
|    | std 1153132.6 | 2.3   | 3.3  |

Table 9: Parallel results for the selected Hadamard matrices.
<table>
<thead>
<tr>
<th>$p$</th>
<th>refines</th>
<th>time</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>avg 1381332314.2</td>
<td>4729.9</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>std 54499930.8</td>
<td>181.2</td>
<td>0.4</td>
</tr>
<tr>
<td>40</td>
<td>avg 1242510280.3</td>
<td>2413.8</td>
<td>20.9</td>
</tr>
<tr>
<td></td>
<td>std 635865065.7</td>
<td>269.6</td>
<td>1.6</td>
</tr>
<tr>
<td>60</td>
<td>avg 925784479.2</td>
<td>1090.3</td>
<td>46.1</td>
</tr>
<tr>
<td></td>
<td>std 90481609.2</td>
<td>107.7</td>
<td>3.5</td>
</tr>
<tr>
<td>80</td>
<td>avg 991879816.8</td>
<td>889.3</td>
<td>56.9</td>
</tr>
<tr>
<td></td>
<td>std 118519409.7</td>
<td>107.1</td>
<td>6.3</td>
</tr>
<tr>
<td>100</td>
<td>avg 937083362.8</td>
<td>674.7</td>
<td>74.0</td>
</tr>
<tr>
<td></td>
<td>std 11196220.3</td>
<td>6.9</td>
<td>0.8</td>
</tr>
</tbody>
</table>

(a) pp-16-17

<table>
<thead>
<tr>
<th>$p$</th>
<th>refines</th>
<th>time</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>avg 1031650624.7</td>
<td>3551.2</td>
<td>9.3</td>
</tr>
<tr>
<td></td>
<td>std 88861831.1</td>
<td>303.2</td>
<td>0.8</td>
</tr>
<tr>
<td>40</td>
<td>avg 780015397.6</td>
<td>1362.0</td>
<td>25.7</td>
</tr>
<tr>
<td></td>
<td>std 209852840.9</td>
<td>366.6</td>
<td>6.1</td>
</tr>
<tr>
<td>60</td>
<td>avg 661773026.1</td>
<td>780.8</td>
<td>42.8</td>
</tr>
<tr>
<td></td>
<td>std 106703884.7</td>
<td>126.8</td>
<td>5.4</td>
</tr>
<tr>
<td>80</td>
<td>avg 654169101.2</td>
<td>587.0</td>
<td>57.1</td>
</tr>
<tr>
<td></td>
<td>std 108169269.8</td>
<td>99.6</td>
<td>7.9</td>
</tr>
<tr>
<td>100</td>
<td>avg 638366624.4</td>
<td>463.1</td>
<td>74.1</td>
</tr>
<tr>
<td></td>
<td>std 141946480.4</td>
<td>106.7</td>
<td>14.8</td>
</tr>
</tbody>
</table>

(b) pp-16-18

<table>
<thead>
<tr>
<th>$p$</th>
<th>refines</th>
<th>time</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>avg 294180748.7</td>
<td>1014.6</td>
<td>18.2</td>
</tr>
<tr>
<td></td>
<td>std 92551820.5</td>
<td>320.0</td>
<td>9.4</td>
</tr>
<tr>
<td>40</td>
<td>avg 181850860.7</td>
<td>317.6</td>
<td>77.5</td>
</tr>
<tr>
<td></td>
<td>std 135824927.6</td>
<td>240.0</td>
<td>41.1</td>
</tr>
<tr>
<td>60</td>
<td>avg 177084192.7</td>
<td>208.6</td>
<td>94.6</td>
</tr>
<tr>
<td></td>
<td>std 99220777.8</td>
<td>118.4</td>
<td>39.2</td>
</tr>
<tr>
<td>80</td>
<td>avg 183069076.6</td>
<td>164.6</td>
<td>121.2</td>
</tr>
<tr>
<td></td>
<td>std 138058484.8</td>
<td>126.0</td>
<td>38.5</td>
</tr>
<tr>
<td>100</td>
<td>avg 314527596.8</td>
<td>230.1</td>
<td>94.7</td>
</tr>
<tr>
<td></td>
<td>std 214547288.5</td>
<td>159.2</td>
<td>41.4</td>
</tr>
</tbody>
</table>

(c) pp-16-22

Table 10: Parallel results for the selected projective planes.
7.3.1 Observations

The average speedup for the graphs derived from Hadamard matrices is nearly linear. Since each graph has a small automorphism group and for each run the generator(s) of the group are discovered early, each run performs about the same number of refinements. Thus, the speedup is approximately linear and the standard deviation of the runtime statistics is low. Since `had-sw-112` has two more automorphisms than the other Hadamard matrices, its average runtime varies more.

For the graphs derived from projective planes however, the automorphism group is larger, so the runtime of the parallel algorithm varies depending on when generators are discovered. If some generators are discovered later in the search, then each process traverses larger subtrees than necessary for a longer period of time. The order of generator discovery is unpredictable and it is possible that the subdivision of the search tree can result in large differences between when generators are discovered. This phenomenon is apparent with `pp-16-22`, whose automorphism group is larger than the other planes. For instance, one run took 92.6 seconds while another took 547.7 seconds. The main difference between the runs that the 92.6-second run discovered most of the automorphism group much earlier than the 547.7-second run.

For some instances, we observe superlinear speedup. We hypothesize that this is due to generators being discovered earlier than in the sequential algorithm, allowing all processes to prune more of their search tree.

119
CHAPTER 8
CONCLUSION

The graph isomorphism problem is a difficult, interesting, and intriguing problem. This dissertation develops several techniques for solving graph isomorphism faster. The first technique we propose modifies individualization-refinement algorithms so that Miyazaki graphs (the only family of graphs with proven exponential runtime) are processed in polynomial time. This is accomplished by exploiting information about the orbits of nodes in the search tree. The set of discovered generators always provided partial information about the graph’s automorphism group, but this information was not exploitable without first introducing the concept of a guide tree.

With the previous exponential bound out of the way, the most prominent remaining difficult instances are graphs with small automorphism groups and seemingly regular structure. These graphs are difficult because typically every cell in each search node is impure. Hence, a stronger refiner is needed to split orbits into different cells. The problem with using a stronger refiner is that in the general case, the extra effort is wasted. We propose an adaptive refinement method which only uses a stronger refiner when needed, or when it will most likely prove worthwhile based on past experience recorded in the guide tree.

These two techniques are complementary. Target cells with the fewest orbits can be identified using the first technique, and these cells can be split further via the second (adaptive refinement). If the cell with the fewest orbits is split via adaptive refinement, then the
resultant partition is guaranteed to have a nontrivial cell with even fewer orbits (if it is not already discrete). This cell, or another with fewer orbits, will be chosen by the target cell chooser.

The third technique explained, short-circuiting generator discovery, has already been shown by the authors of *saucy* to vastly speed up instances which are highly symmetric. This technique assumes that most generators are sparse and attempts to short-circuit generator discovery based on this assumption.

Finally, we develop a parallel algorithm for canonically labeling a graph and discovering generators of its automorphism group. The parallel algorithm is benchmarked, and shown to achieve impressive superlinear speedup in certain cases.

### 8.1 Future work

#### 8.1.1 Group target-cell functions

Chapter 4 introduces a new method of target-cell choosing. We have only benchmarked one group target-cell function; there are certainly others worth of being examined. For instance, using the heuristic of fewest orbits might not be a good idea. The ideal goal of choosing the cell with the fewest orbits is to find a cell which only contains a single orbit. Why not just use a group target-cell function which acts like *max-joins* unless a cell with only one orbit is
detected? We believe that this will be a more effective target-cell chooser for non-Miyazaki graphs.

Another way to improve the effectiveness of using a group target-cell function is to use exact orbit pruning. As the results in Subsection 4.8.3 show, approximate orbit pruning leads to too many generators being discovered. We plan on using J. S. Leon’s implementation of the Schreier-Sims algorithm to calculate exact orbits [58] as a replacement for approximate orbit calculation.

8.1.2 Adaptive refinement

One drawback of adaptive refinement is that it only looks one level deep. For some of the graphs benchmarked in Chapter 5, this is insufficient. The splits in cells which are impure, but not directly impure are not detectable to the current method of adaptive refinement. To reveal these splits, the node-invariant values two or more levels deep must be used to distinguish between vertices in different orbits. We propose performing multi-level adaptive refinement up to an arbitrary depth whenever a target cell is identified to be impure.

8.1.3 Parallel algorithm improvement

The parallel algorithm is young in its development. We intend to benchmark the algorithm on a wider suite of graphs to better gauge its scalability and performance.
8.2 Final remarks

The author shares the hope expressed in [52] that one day a polynomial-time graph isomorphism algorithm will be discovered. In the presence of symmetry, the graph isomorphism problem seems nearly solved. However, solving graph isomorphism necessitates solving graph automorphism. Determining if a graph is rigid or not is a difficult problem in and of itself. If the graph automorphism problem can be solved in polynomial time, the result might be extended to general graph isomorphism.

One avenue of attacking the isomorphism of rigid graphs is to determine the number of vertices which are required to be individualized before the equitable refiner (or even the node-invariant refiner) can determine that no automorphisms exist. If the number is a small enough function of the number of vertices in the graph, then the height of the search tree could be appropriately bounded such that a polynomial-time algorithm is possible.
LIST OF REFERENCES


GLOSSARY

action: see group action. 15

adaptive refinement: our strategy of performing node-invariant refinement only when it is determined to be useful. 98

approximate minimum cell representatives: a heuristic for approximating the minimum cell representatives. 36

approximate pruning: pruning of the search tree which uses approximate minimum cell representatives; can result in more generators than necessary. 36

automorphism: an isomorphism from a graph to itself. 5

automorphism group: the group of all automorphisms of a graph. 17

bliss: canonical labeling algorithm by Junttila and Kaski; based on nauty. 3

bridge-vertices: vertices in a Miyazaki graph derived from the bridges of \( Y_k \). 48

canonical form: a certificate whose range and domain are the same. 19

canonical labeling: a permutation \( \gamma \) of a graph \( G \) such that \( G^\gamma \) is a canonical representative; alternatively, a function which when input a graph \( G \) returns the \( \gamma \). 19
**individualization (of a vertex):** splits the cell $W$ containing the vertex $u$ into the two cells $\{u\}$ and $\{W \setminus u\}$. 21

**canonical representative:** a canonical representative of an isomorph class. 2

**cell:** a set which is an element of a partition, usually denoted by $W$. 12

**certificate:** see isomorphism certificate. 19

**color preserving automorphism group:** the group of permutations which fix a colored graph. 17

**color class:** the set of all elements colored the same by a coloring. 13

**colored graph:** a pair $(G, \pi)$ where $G$ is a graph and $\pi$ is a coloring. 14

**coloring (of a set $V$):** a function $\pi$ from $V$ to a totally ordered set $\{v_1, v_2, \ldots, v_k\}$. 13

**cycle-vertices:** vertices in a Miyazaki graph derived from cycles in $Y_k$. 48

**degree (of a vertex in a set $W$):** the number of neighbors of a vertex in the set $W$. 22

**directly impure cell (with respect to a node invariant):** a cell in a colored graph is directly impure with respect to an invariant if there exist two vertices in the cell with different invariant values. 97

**discrete partition:** a partition where every cell has size one. 16

**disjoint cycle notation:** a compact notation for representing permutations. 16
equitable partition: a partition \( \pi \) is equitable (with respect to a graph \( G \)) if and only if for every pair of vertices \( u \) and \( v \) in the same cell and for every cell \( W \) of \( \pi \),
\[
\deg_G(u, W) = \deg_G(v, W).
\]
equitable refiner: a refiner is equitable if its output is an equitable partition.

generators (of a group): a subset \( \theta \) of a group \( A \) such that every element of \( A \) can be expressed as a product of elements of \( \theta \).

group action: the action of a permutation group element on an object.

group search tree: a search tree \( T(G, \pi, A) \) which uses a group target-cell function.

group target-cell function: a target cell function \( \text{tc}(G, \pi, A) \) taking in a third parameter \( A \), a subgroup of \( \text{Aut}(G, \pi) \); the third property for group target-cell functions is (iii)
\[
\text{tc}(G^\gamma, \pi^\gamma, A^\gamma) = k = \text{tc}(G, \pi, A) \text{ for all } \gamma \in \text{Sym}(V).
\]
guide tree: a data structure we present to allow for changing target cell choices.
impure cell: a cell in a colored graph is impure if it contains more than one orbit. 97

incremental leaf certificate: a node invariant $I(G, \pi, \nu)$ such that $\vec{I}(G, \pi, \nu)$ is a leaf certificate. 33

index (of a cell): the sum of all previous cell sizes. 14

indices (of a partition): the set of indices of each cell in the partition. 14

invariant: see isomorphism invariant. 18

isomorphism (w.r.t. $\text{Sym}(V)$): a permutation $\gamma \in \text{Sym}(V)$ such that $X^\gamma = Y$ for any two objects; applies to graphs, colored graphs, search nodes, and search trees. 17

isomorphism certificate: an isomorphism invariant with the additional property that $I(X) = I(Y)$ implies $X \cong Y$. 19

isomorphism invariant: a function $I$ which is invariant on isomorphic objects: $X \cong Y$ implies $I(X) = I(Y)$. 19

leaf certificate: an node invariant $I(G, \pi, \nu)$ which is a leaf when restricted to leaf nodes. 31

minimum cell representatives: the set of elements in a permutation group which are minimal in their orbits. 36

Miyazaki graphs: a family of graphs with parameter $k$ constructed by Miyazaki to show the exponential runtime of $\text{nauty}$, denoted by $\mathcal{X}(Y_k)$. 48
MPI: acronym for the Message Passing Interface. 110

nauty: canonical labeling algorithm by McKay; currently the basis for saucy and bliss. 3

neighborhood (of a vertex): the set of vertices adjacent to a vertex. 22

nishe: the name of our new canonical labeling algorithm. 61

node-invariant refiner: a refiner which splits cells based on a node-invariant. 96

node invariant: an invariant \( I(G, \pi, \nu) \) defined on nodes in the search tree \( T(G, \pi) \). 31

nontrivial cell: a cell of size greater than one. 14

nontrivially joined cells: two cells \( U \) and \( W \) in a colored graph \( (G, \pi) \) are nontrivially
joined if for all \( w \in W, 0 < \deg_G(w, U) < |U| \). 27

orbit of a vertex: the set of vertices a vertex can move to via automorphisms. 18

orbits of a graph: the set of orbits of the vertices of a graph. 18

ordered partition (of a set \( V \)): an ordered sequence of disjoint sets \( (W_1, W_2, \ldots, W_k) \)
which union to \( V \). 12

partition: abbreviation for ordered partition. 12

partition stabilizer: the partition stabilizer of \( \pi \) in a permutation group \( A \) is \( A_\pi = \{ \gamma \in A : \pi^\gamma = \pi \} \). 67

pure cell: a cell in a colored graph is pure if it contains exactly one orbit. 97
**refiner:** a function \( R : \mathcal{G}(V) \times \Pi(V) \to \Pi(V) \) designed to split vertices not in the same orbit; satisfies (i) \( R(G, \pi) \leq \pi \) and (ii) \( R(G, \pi)\gamma = R(G\gamma, \pi\gamma) \) for all \( \gamma \in \text{Sym}(V) \). 25

**rigid graph:** a graph whose only automorphism is the identity. 7

**saucy:** symmetry detection algorithm by Darga, Skallah, and Markov; based on *nauty*. 3

**search tree:** the tree \( T(G, \pi) \) traversed by the base canonical labeling algorithm. 28

**singleton permutation:** the singleton permutation between two partitions \( \pi_1 \) and \( \pi_2 \) maps the element in each singleton cell of \( \pi_1 \) to the element at the same index in \( \pi_2 \) (only if it is a singleton in \( \pi_2 \) as well). 106

**symmetry:** an automorphism of a graph. 5

**target-cell function:** a function \( \text{tc} : \mathcal{G}(V) \times \Pi(V) \to \{1, \ldots, n\} \) which returns a nontrivial cell of its input; if \( k = \text{tc}(G, \pi) \) then \( k \) must satisfy (i) \( k \in \text{indices}(\pi) \), (ii) \( |\pi[k]| \geq 2 \), and (iii) \( \text{tc}(G\gamma, \pi\gamma) = \text{tc}(G, \pi) \) for all \( \gamma \in \text{Sym}(V) \). 27

**trivial cell:** a cell of size one. 14

**vertex-invariant:** any invariant that could potentially distinguish between two vertices not in the same orbit. 90

**weakest-invar:** a weak leaf certificate. 31