Graph Distinguishability and the Generation of Non-Isomorphic Labellings

by

William Herbert Bird

B.Sc., University of Victoria, 2011

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Graph Distinguishability and the Generation of Non-Isomorphic Labellings

by

William Herbert Bird

B.Sc., University of Victoria, 2011

Supervisory Committee

Dr. Wendy Myrvold, Supervisor

(Department of Computer Science)

Dr. Frank Ruskey, Departmental Member

(Department of Computer Science)

Supervisory Committee

Dr. Wendy Myrvold, Supervisor

(Department of Computer Science)

Dr. Frank Ruskey, Departmental Member

(Department of Computer Science)

ABSTRACT

A *distinguishing colouring* of a graph G is a labelling of the vertices of G with colours such that no non-trivial automorphism of G preserves all colours. The *distinguishing number* of G is the minimum number of colours in a distinguishing colouring. This thesis presents a survey of the history of distinguishing colouring problems and proves new bounds and computational results about distinguishability. An algorithm to generate all labellings of a graph up to isomorphism is presented and compared to a previously published algorithm. The new algorithm is shown to have performance competitive with the existing algorithm, as well as being able to process automorphism groups far larger than the previous limit. A specialization of the algorithm is used to generate all minimal distinguishing colourings of a set of graphs with large automorphism groups and compute their distinguishing numbers.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Consider the following problem of a graduate student and his ring of keys[1]: Each night, after staying late at the university to work on his thesis, he returns home and fumbles in the dark trying to determine which key fits his front door. Not wanting to disturb his housemates by trying every key on his key ring in the lock, he decides to attach distinctive handles to his keys to easily tell them apart in the dark. Due to the frugal circumstances of graduate school, he also wants to use as few different shapes of handle as possible but still be able to exactly identify every key on his ring regardless of the ring's orientation.

The student has stumbled upon the problem of *distinguishability*, which is the focus of this thesis. The minimum number of handle shapes needed is the *distinguishing number*, and an assignment of handles to keys such that every key can be uniquely identified regardless of the orientation of the ring is a *distinguishing colouring*. This thesis studies the mathematical foundations of distinguishability, proves new bounds on distinguishing numbers and formulates a new algorithm to enumerate all distinguishing colourings of arbitrary graphs, which is then tested against an existing algorithm. The algorithm is also applied to the exhaustive generation of other types of labellings up to isomorphism.

Chapter 2 is a survey of the computational group theory results which form the foundation

[1] Embellished from Tymoczko [35].

of the backtracking generation algorithm. Chapter 3 introduces the topic of distinguishability, along with a survey of its history and related problems, as well as new bounds and complexity results. The backtracking generation algorithm is presented in Chapter 4, along with optimizations and applications to various generation problems. Finally, Chapter 5 describes the datasets which were generated during this research, including benchmarks of the algorithm and the distinguishing numbers of a catalogue of tested graphs.

1.1 Definitions

In general, the notational style used for group theoretic aspects of this thesis will follow Gallian [14], while graph theoretic aspects will follow West [37]. The notation used for computational complexity in this thesis will follow Arora and Barak [2].

The algorithms and data structures studied in this thesis are intended for use with a random access machine with a word size sufficient to store all of the numerical values computed in each algorithm. Asymptotic bounds on time and space complexity are stated in terms of this (fixed) word size.

1.1.1 Graphs

A *graph* G is an ordered pair (V, E) containing a set V of *vertices* and a collection E of *edges* which are pairs (u, v) with $u, v \in V$. When the pairs in E are ordered, G is called a *directed graph*, and the pairs in E are usually called *arcs*. Otherwise, the graph G is *undirected*. If E is a multiset (which may contain a pair u, v more than once), G is called a *multigraph*. A *loop* in a graph is an edge from a vertex v to itself. A graph with no loops and no duplicate edges is called *simple*, and in this document, all graphs will be simple and undirected unless otherwise stated. For a given graph G, the associated vertex and edge sets may be denoted by $V(G)$ and $E(G)$ respectively.

A *path of length k* in a graph $G = (V, E)$ is a sequence $v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k, e_k, v_{k+1}$ with each $v_i \in V$, $e_i \in E$ and the property that e_i is incident with both v_i and v_{i+1}. A vertex v is said to be *reachable* from a vertex u if there exists a path from u to v. The reachability relation partitions the graph G into one or more *connected components*, and G is called *connected* if it has only one connected component. In a directed graph, a *directed path* has the property that the edge e_i equals the ordered pair (v_i, v_{i+1}). Unless otherwise stated, a vertex v is reachable from a vertex u in a directed graph only if there is a directed path from u to v. The *strongly connected components* of a directed graph are a partition of the vertices such that for two vertices v_1 and v_2 in the same strongly connected component, there is a directed path from v_1 to v_2 and v_2 to v_1. A directed graph is said to be *strongly connected* if it has only one strongly connected component.

A *subgraph* of $G = (V, E)$ is a graph $H = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.A graph H is a *spanning subgraph* of G if $V' = V$. Two graphs G_1 and G_2 are said to be *isomorphic*, written $G_1 \cong G_2$, if there exists some bijection $\pi : V(G_1) \to V(G_2)$ which puts edges of G_1 in one-to-one correspondence with edges of G_2. The *automorphisms* of a graph G are graphs G' which are obtained by relabeling the vertices of G and are identical to G. The *automorphism group* of a graph G is the group of permutations (defined in Section 1.1.3) corresponding to the automorphisms of G, and will be denoted by $\mathrm{Aut}(G)$ in this thesis.

The *complement* of G, denoted $\overline{G}$, has

$$V(\overline{G}) = V(G)$$
$$E(\overline{G}) = \{uv : uv \notin E(G)\}.$$

The *complete graph* on n vertices, denoted K_n, is the n-vertex graph with all possible edges. A *clique* of size k (or a k-clique) in a graph G is a subset $S \subseteq V(G)$ of k vertices for which there is an edge between every pair of vertices. An *independent set* of size k (or a

k-independent set) in a graph G is a subset $S \subseteq V(G)$ of k vertices for which there are no edges between any pair of vertices.

1.1.2 Complexity Classes

The definitions given in this section are sufficient for the high-level treatment of complexity problems given in Chapter 3. More rigorous and thorough definitions of fundamental aspects of complexity (such as Turing machines) are not particularly relevant here and therefore are omitted.

Let ρ be a computational problem and consider an instance of ρ with input size n. A problem ρ is in the complexity class P if there exists a polynomial-time (in n) algorithm to solve ρ. A problem ρ is in the complexity class NP if there exists an algorithm which, given a 'certificate' of size $O(n^k)$ (for some fixed k) which contains a purported solution, can verify that solution in polynomial time. When ρ is a *decision problem* (a problem with a 'yes' or 'no' answer), it is only necessary that a 'yes' answer be verifiable in polynomial time for ρ to belong to NP. If a 'no' answer can be verified in polynomial time, ρ lies in the complementary class coNP.

A *polynomial-time reduction* of a problem ρ_1 to a problem ρ_2 is a polynomial-time algorithm which, given an instance of ρ_1, produces an instance of ρ_2 whose solution gives the solution to the original instance of ρ_1. A problem ρ is said to be 'NP-hard' if every problem in NP can be reduced in polynomial-time to an instance of ρ. If ρ is in NP and NP-hard, then ρ is said to be 'NP-complete'.

The relationship between P and NP can be generalized to arbitrary quantified formulas to produce the 'polynomial hierarchy'. Define

$$\Sigma_0^p = \Pi_0^p = \mathrm{P}$$

and, for integers $i \geq 1$, define the class Σ_i^p to contain all decision problems for which a 'yes' answer can be verified in polynomial time by an algorithm with access to an 'oracle' which can solve any problem in Π_{i-1}^p in constant time. Similarly, define Π_i^p to contain decision problems for which a 'no' answer can be verified in polynomial time by an algorithm with access to a Σ_{i-1}^p oracle.

Under this model, NP is the class Σ_1^p and its complement coNP is the class Π_1^p. Both classes at level i of the hierarchy are contained in all classes at higher levels (so Σ_{i+1}^p contains both Σ_i^p and Π_i^p).

A classic NP-complete problem is SAT, which determines whether a boolean formula ϕ on n variables $x_1, \ldots, x_n$ has a solution. The decision problem can be represented by the quantified sentence:

$$\exists x_1, \ldots, x_n \quad \phi(x_1, \ldots, x_n)$$

The complementary problem $\overline{\text{SAT}}$ is coNP-complete and determines whether ϕ has no solutions. It can be represented by the quantified sentence

$$\forall x_1, \ldots, x_n \quad \overline{\phi(x_1, \ldots, x_n)}$$

SAT can generalized to decision problems involving an arbitrary number of quantifiers. Evaluating sentences with i alternating quantifiers is a complete problem for level i of the polynomial hierarchy, with the first quantifier determining whether the problem lies in Σ_i^p (for existential quantifiers) and Π_i^p (for universal quantifiers). For example, evaluating sentences of the form

$$\exists x_1, \ldots, x_k \forall x_{k+1}, \ldots, x_l \exists x_{l+1}, \ldots, x_n \quad \phi(x_1, \ldots, x_n) \qquad (1 \leq k < l < n)$$

is a Σ_3^p-complete problem, while the complement

$$\forall x_1, \ldots, x_k \exists x_{k+1}, \ldots, x_l \forall x_{l+1}, \ldots, x_n \quad \overline{\phi(x_1, \ldots, x_n)} \qquad (1 \leq k < l < n)$$

is Π_3^p-complete.

The polynomial hierarchy is said to *collapse* to a level $i \geq 1$ if $\Sigma_i^p = \Pi_i^p$. If the hierarchy collapses to level i, then all levels above level i are contained in level i. Currently it is not known whether the polynomial hierarchy collapses to any level.

1.1.3 Groups and Permutations

A *group* is a set G with a corresponding binary relation $\bullet$ with the following properties:

1. G is closed under $\bullet$. That is, for $g_1, g_2 \in G$, the product $g_1 \bullet g_2$ is also in G.
2. The operation $\bullet$ is associative with respect to elements in G.
3. There exists an element $e \in G$ such that for all $g \in G$, $e \bullet g = g \bullet e = g$. This element is called the *identity* of G.
4. For every $g \in G$, there exists a unique element $g^{-1} \in G$ such that $g \bullet g^{-1} = e$. The element g^{-1} is called the *inverse* of g.

Unless the operation $\bullet$ is explicitly specified, it is assumed to be the 'natural' operation for the type of object contained in G. For example, if G is a set of integers, the natural operation is addition, and if G is a set of permutations, the natural operation is function composition (discussed below). Additionally, groups with multiplication-like operations (including permutation groups) normally use multiplication-style notation, so the product of two elements $a, b \in G$ is written ab instead of $a \bullet b$.

Let A and B be groups. An *isomorphism* from A to B is a bijective mapping $f : A \to B$ such that for all pairs $a_1, a_2 \in A$, $f(a_1 a_2) = f(a_1) f(a_2)$. If an isomorphism from A to B exists, then A and B are said to be *isomorphic*. The group B is a *subgroup* of A, written

$B \leq A$, if $B \subseteq A$. If A is a group and $S \subseteq A$, the *group generated by* S, denoted $\langle S \rangle$, is the closure of S under the group operation of A. A set S is a group if and only if $\langle S \rangle = S$.

A *permutation* of a set Ω is a bijection $\pi : \Omega \to \Omega$. In this thesis, permutations will be considered to be specialized functions, not a separate type of formal object. Given a permutation α and an element $x \in \Omega$, the image of x under α will be written with functional notation as $\alpha(x)$. Similarly, the group operation for permutation groups will be assumed to be function composition, so for two permutations α and β,

$$\alpha\beta(x) = \alpha \circ \beta(x) = \alpha(\beta(x))$$

It is important to note that there is no settled consensus on whether permutation multiplication should proceed from right-to-left (as in function composition) or left-to-right. The right-to-left ordering is used here to follow the style of Gallian [14]. Some of the sources referenced by this thesis, particularly older articles and books, use left-to-right order for permutation multiplication, and in most cases there is no information in source material about which order is used (leaving it to the reader to determine the order being used). The differences between notational conventions can be jarring, since they also affect naming of other group theoretical concepts (a 'left coset' with right-to-left multiplication becomes a 'right coset' in left-to-right order). Since the sources of some foundational results (including the seminal papers by Sims [34, 33]) use the alternative notation, the restatement of those results in this document appears reversed.

The *Symmetric group* on a set Ω, denoted $\mathrm{Sym}(\Omega)$ is the group of all permutations of the elements of Ω under the function composition operation described above. The symmetric group on the symbols $\{1, \ldots, n\}$ is denoted by S_n. Any permutation group on n symbols is isomorphic to a subgroup of S_n, and therefore the symbols used in this document will always be $\{1, \ldots, n\}$ unless otherwise stated.

A permutation $\tau \in S_n$ is a *cycle* if there exists a sequence $s_1, s_2, \ldots, s_k$ of distinct values

in $\{1, 2, \ldots, n\}$ such that τ fixes all elements of $\{1, 2, \ldots, n\}$ which are not equal to some s_j, and,

$$\tau(s_j) = s_{j+1} \text{ for } j = 1, 2, \ldots, k-1$$
$$\tau(s_k) = s_1.$$

The cycle τ on the sequence $s_1, s_2, \ldots, s_k$ can be represented with the notation

$$\tau = (s_1 s_2 \ldots s_k).$$

Two cycles τ_1 and τ_2 are *disjoint* if for all $i \in \{1, 2, \ldots, n\}$, either $\tau_1(i) = i$ or $\tau_2(i) = i$. Every permutation $\pi \in S_n$ can be written uniquely (up to the ordering of terms and cycle elements) as the product of disjoint cycles.

If $B \leq A$ and $a \in A$, the set

$$aB = \{ab : b \in B\}$$

is called a *left coset* of B in A, and the element a is a *coset representative* of aB. The set Ba is similarly called a *right coset*. In general, it is not the case that $aB = Ba$. If B has the property that $aB = Ba$ for all $a \in A$, B is called a *normal* subgroup of A.

The coset eB containing the identity element e is equal to B, and the remaining cosets of B in A partition the elements of $A - B$.

Later counting results will hinge on the following famous theorem of Lagrange.

Theorem 1.1 (Lagrange [14])**.** *If $B \leq A$, then $|B|$ divides $|A|$ and the number of cosets (left or right) of B in A is $|A|/|B|$.*

The number of cosets of B in A is called the *index* of B in A and written $|A : B|$. A left (or right) *transversal* of B in A is a set of coset representatives for each coset of B in A. In this document, the terms 'coset' and 'transversal' with no qualification will refer to left

cosets and left transversals, respectively.

Let $G \leq \mathrm{Sym}(\Omega)$. The *pointwise stabilizer* of a set $\Delta \subseteq \Omega$ is the set

$$\mathrm{St}(\Delta) = \{g \in G : g(i) = i \quad \forall i \in \Delta\}$$

containing all permutations in G which fix all of the elements of Δ. The shorthand $\mathrm{St}(i)$ is used to denote the pointwise stabilizer of a single point. The *setwise stabilizer* of Δ is the set

$$\mathrm{SetSt}(\Delta) = \{g \in G : g(i) \in \Delta \quad \forall i \in \Delta\}$$

of permutations which map elements of Δ to other elements of Δ. In this document, the unqualified term 'stabilizer' will always refer to a pointwise stabilizer.

1.1.4 Colourings of Graphs and Groups

Let G be a graph on n vertices, arbitrarily numbered $v_1, v_2, \ldots, v_n$. A *colouring* of the vertices of G with k symbols is a sequence $C = c_1 c_2 \ldots c_n$, where $c_i \in \{0, \ldots, k-1\}$. The action of $\mathrm{Aut}(G)$ partitions the set of k^n such sequences into equivalence classes. It should be noted that the colourings studied here do not place any restrictions on which vertices can take which colours. The graph theoretic problem of finding an assignment of colours in which two neighbouring vertices receive different colours will be called 'proper colouring' in this thesis.

If $C = c_1 c_2 \ldots c_n$ and $C' = c'_1 c'_2 \ldots c'_n$ are colourings of G, then C is said to be *lexicographically less than* C', denoted by $C < C'$, if for some $i \leq n$, $c_i < c'_i$ and

$$c_j = c'_j \text{ for all } 1 \leq j < i.$$

If $C = c_1 c_2 \dots c_n$ is a colouring of G, then for a permutation $\pi \in \mathrm{Aut}(G)$ the *permuted colouring* C_π resulting from applying π to the indices of C is the colouring

$$C_\pi = c_{\pi(1)} c_{\pi(2)} \dots c_{\pi(n)}$$

The permutation π is said to *fix* C if for $i = 1, 2, \dots n$, $c_i = c_{\pi(i)}$.

The *colour classes* of C, written $C^1, C^2, \dots, C^k$, are sets

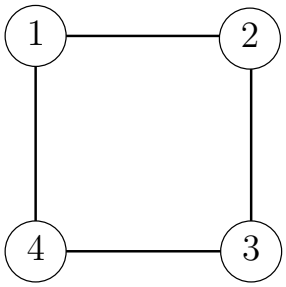
$$C^z = \{i : c_i = z\}$$

The definition used here does not preclude a colour class being empty. Indeed, the string 0^n is a lexicographically minimum k colouring of every graph G on n vertices for all values of n and $k \geq 1$.

A colouring C is said to be *lexicographically minimum* relative to the action of $\mathrm{Aut}(G)$ if for all $\pi \in \mathrm{Aut}(G)$,

$$C \leq C_\pi$$

Figure 1.1 shows the set of lexicographically minimum colourings using 2 and 3 colours for the automorphism group of the 4-cycle. The sequence $C = 1010$ is an example of a non-lexicographically-minimum 2-colouring (and 3-colouring) of the graph in Figure 1.1, since the permutation $\pi = (1234)$ (a counter-clockwise rotation) gives $C_\pi = 0101$, which is lexicographically less than C.



(a) Graph

0000
0001
0011
0101
0111
1111

(b) 2-colourings

0000	0102	0222
0001	0111	1111
0002	0112	1112
0011	0121	1122
0012	0122	1212
0022	0202	1222
0101	0212	2222

(c) 3-colourings

Figure 1.1: The 4-cycle C_4 with its lexicographically minimum 2- and 3-colourings

Colourings can also be defined for permutation groups H over the set $\Omega = \{1, 2, \ldots, n\}$ by adopting the convention that a colouring

$$C = c_1 c_2 \ldots c_n$$

colours the element $i \in \Omega$ with the colour c_i.

For notational clarity, the chapters of this thesis alternate between graph- and group-centric representations depending on the context. Chapters 2 and 4 are primarily group-focused and use the group version of colouring defined above, while Chapter 3, which is graph-focused, treats a colouring as a labelling of the vertices of a graph. In the group-focused chapters, G is used as the default group name, while in graph-focused chapters, G is used as the default graph name. In both cases, all uses of the name G clearly indicate whether it is a graph or group.

Chapter 2

Computational Group Representation

There are two fundamental approaches to representing an arbitrary permutation group $G \leq S_n$ in a computer. The simplest approach is to store a list of all permutations in G. Since this method requires $\Theta(n|G|)$ space, it is not feasible for large groups. Alternatively, a set S of permutations which generates G can be stored. In general, finite groups can be generated by relatively small generating sets. The symmetric group S_n, which has size $n!$, can be generated by the permutations

$$\tau = (123\ldots n) \qquad \text{and} \qquad \sigma_{12} = (12).$$

The problem of finding small generating sets, or determining the size of the smallest generating set of a group G (also known as the *rank* of G) are computationally difficult problems [26].

For subgroups of S_n, an algorithm described by Jerrum in [18] can find a generating set of size at most $n-1$, while the Schreier-Sims algorithm [34, 33] discussed later in this chapter finds a generating set of size at most $\binom{n}{2}$.

2.1 Sims Tables

An algorithm described briefly by Sims in [34] constructs a table of at most $\binom{n}{2}$ permutations which generate and uniquely characterize every permutation in a group $G \leq S_n$. The theoretical underpinnings of the algorithm are further described by Sims in [33]. The algorithm is usually called the 'Schreier-Sims algorithm' since a key insight is provided by a lemma of Schreier. The constructed table of permutations will be called a 'Sims table' in this thesis, following the convention of Knuth in [20]. In the broader mathematical literature, Sims' results are often discussed without any reference to tables, since a table is only necessary for implementation purposes, so the term 'Sims table' does not seem to be widely used in theoretical contexts. A Sims table for a group $G \leq S_n$ can be constructed from a set of $O(n^3)$ generators in $\Theta(n^5)$ time [5].

Let $G \leq S_n$. A sequence $X = k_1, \ldots, k_m$ of elements of $\{1, 2, \ldots, n\}$ is a *base* for G if no element of G except the identity permutation e fixes every k_i. The sequence $X = 1, 2, \ldots, n-1$ is a base for S_n, and therefore also a base for any subgroup of S_n.

Given a base $X = k_1, \ldots, k_m$ of G, define the set G_i for $i \in \{1, \ldots, m\}$ by

$$G_i = \{g \in G : g(k_j) = k_j \text{ for all } j < i\}.$$

Each element of G_i is a permutation which fixes the first $i-1$ elements of the base B. The set G_1 is equal to the entire group G, and G_m contains only the identity permutation.

Each G_i is closed under the group operation. Additionally, the inverse of every element in G_i also lies in G_i. Therefore, G_i is a group for all i, and the set of all G_i forms a chain

$$G = G_1 \geq G_2 \geq \ldots \geq G_m = \{e\}$$

of subgroups.

If U_i is a set of coset representatives for G_{i+1} in G_i, then every element α_i of G_i can be written uniquely in the form

$$\alpha_i = g_i \beta$$

where $g_i \in U_i$ and $\beta \in G_{i+1}$. Applying this relation recursively yields the following result.

Lemma 2.1 (Sims [33]). *Given a set of transversals U_i for each G_i as defined above, every element $\pi \in G$ can be written uniquely in the form*

$$\pi = g_1 g_2 \dots g_m$$

where $g_i \in U_i$.

Since G_i fixes all elements $k_1, \dots, k_{i-1}$, and G_{i+1} fixes $k_1, \dots, k_i$, the cosets of G_{i+1} in G_i correspond to the possible images of k_i under elements of G_i. Consequently, the cosets of G_{i+1} in G_i can be identified with images of k_i, the coset containing an element $\pi_i \in G_i$ can be determined by evaluating $\pi_i(k_i)$. Finally, given some element π in the outer group G, the largest subgroup G_i containing π can be easily identified by finding the smallest i such that $\pi(k_i) \neq k_i$.

Lemma 2.1 places no constraints on the transversals U_i other than requiring that they be fixed. This, combined with the ease with which each coset of G_{i+1} in G_i can be identified, allows a set of transversals U_i to be constructed from a list of the permutations in the group by a simple iterative process, detailed in Algorithm 1.

Algorithm 1 Build traversals U_i from the permutations in G

```
1: procedure AddPermutation(π)
2:     for i ← 0, ..., m − 1 do
3:         if π(k_{i+1}) ≠ k_{i+1} then
4:             Break
5:         end if
6:     end for
7:     {The permutation π is contained in G_i and not contained in G_{i+1}}
8:     {Get the index of the coset containing π}
9:     j ← π(k_{i+1})
10:    if U_i does not contain a representative of coset j then
11:        Add π to U_i
12:    end if
13: end procedure
14: procedure BuildTransversals(G)
15:    Let k_1, ..., k_m be a base for G
16:    Create empty transversals U_0, ..., U_{m−1}
17:    Add the identity permutation e to each U_i
18:    for each permutation π ∈ G do
19:        AddPermutation(π)
20:    end for
21: end procedure
```

The maximum number of cosets of G_{i+1} in G_i is the number of possible images of k_i under permutations in G_i. Since every permutation in G_i fixes the set $\{k_1, \ldots, k_{i-1}\}$, there are $n - i + 1$ possible images of k_i, so the maximum size of U_i is $n - i + 1$.

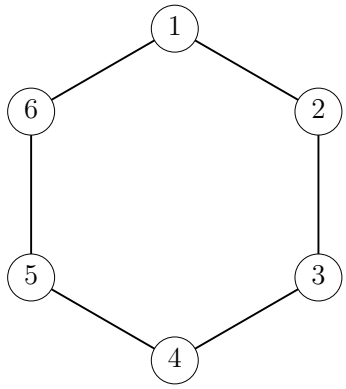
The elements of U_i can be named according to where they send k_i. Define σ_{ij} to be the representative of the coset mapping k_i to j in U_i (if such a coset exists). A *Sims table* for G is an $m \times n$ array T, where

$$T[i][j] = \begin{cases} \sigma_{ij} & \text{if } \sigma_{ij} \text{ exists} \\ \times & \text{otherwise} \end{cases}$$

The symbol $\times$ is used to indicate a missing or invalid entry of the table.

Figure 2.1 shows the Sims table and valid permutations σ_{ij} for the automorphism group of the 6-cycle (which is the dihedral group D_6). The permutations σ_{ii} are all equal to the

identity[1] and not shown in the listing.



(a) Graph

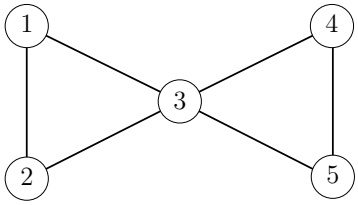
$$\begin{pmatrix} e & \sigma_{12} & \sigma_{13} & \sigma_{14} & \sigma_{15} & \sigma_{16} \\ \times & e & \times & \times & \times & \sigma_{26} \\ \times & \times & e & \times & \times & \times \\ \times & \times & \times & e & \times & \times \\ \times & \times & \times & \times & e & \times \\ \times & \times & \times & \times & \times & e \end{pmatrix}$$

(b) Sims Table

$$\begin{aligned} \sigma_{12} &= (12)(36)(45) \\ \sigma_{13} &= (13)(46) \\ \sigma_{14} &= (14)(23)(56) \\ \sigma_{15} &= (153)(264) \\ \sigma_{16} &= (165432) \\ \sigma_{26} &= (26)(35) \end{aligned}$$

(c) Sims Permutations

Figure 2.1: Sims table and permutations for C_6



(a) Graph

$$\begin{pmatrix} e & \sigma_{12} & \times & \sigma_{14} & \sigma_{15} \\ \times & e & \times & \times & \times \\ \times & \times & e & \times & \times \\ \times & \times & \times & e & \sigma_{45} \\ \times & \times & \times & \times & e \end{pmatrix}$$

(b) Sims Table

$$\begin{aligned} \sigma_{12} &= (12) \\ \sigma_{14} &= (14)(25) \\ \sigma_{15} &= (15)(24) \\ \sigma_{45} &= (45) \end{aligned}$$

(c) Sims Permutations

Figure 2.2: Sims table and permutations for the bowtie B

Applying Lemma 2.1 to a Sims table representation yields the following parallel result.

Lemma 2.2 (Sims [33])**.** *Given a Sims table for $G \leq S_n$ produced for a base $k_1, \ldots, k_m$, every element $\pi \in G$ can be written uniquely in the form*

$$\pi = \sigma_{1j_1}\sigma_{2j_2}\ldots\sigma_{mj_m}.$$

□

[1]Setting $\sigma_{ii} = e$ is a de facto convention in the literature for Sims tables, and in some sources (and most applications) it is assumed to be the case. However, the definition of a Sims table does not require any specific representative for coset i in G_i.

Since the only valid entry of row m of the table will be $\sigma_{mm} = e$, the last entry in the product in Lemma 2.2 can be ignored in practice. By considering row i of the Sims table instead of the set U_i, and placing the representative of coset j in σ_{ij}, Algorithm 1 can be used to generate a Sims table from a list of the permutations in G.

Since the length m of a base is at most n, any Sims table for G will have at most $\binom{n}{2}$ valid entries, and since each permutation in the table can be stored in $O(n)$ space, the entire table can be stored in $O(n^3)$ space. By Lemma 2.2, the group G is generated by the elements of the table, giving the following result.

Lemma 2.3 (Sims [33])**.** *Any permutation group $G \leq S_n$ has a generating set of size at most $\frac{n(n-1)}{2}$.*

□

Sims tables provide a middle-ground between storing a list of permutations in the group G (which requires $O(|G|)$ space) and storing a small generating set (which can require less space but complicates the process of accessing individual elements). A key advantage of the generating set provided by a Sims table is the uniqueness of the factorization of the elements of G, which enables easy access to each element of G. Algorithm 2 gives pseudocode to generate all permutations of G. The recursive process is started by calling GENERATE-PERMUTATIONS with T set to a Sims table for G, `level` set to 1 and π set to the identity permutation e.

Algorithm 2 Generate all permutations in G from a Sims table T

```
1: procedure GeneratePermutations(T, level, π)
2:     if level = m then
3:         Output the permutation π
4:         return
5:     end if
6:     for j ← level, level + 1, ..., n do
7:         if T[level][j] ≠ × then
8:             σ ← Permutation stored in T[level][j]
9:             GeneratePermutations(T, level + 1, πσ)
10:        end if
11:    end for
12: end procedure
```

Lemma 2.1 also yields an easy method to determine $|G|$.

Lemma 2.4 (Sims [33]). *Given a set of transversals U_i for $G \leq S_n$ with respect to some base $X = k_1, \ldots, k_m$,*

$$|G| = \prod_{i=0}^{m} |U_i|.$$

□

The size of G can therefore be found algorithmically by taking the product of the number of valid entries in each row of a Sims table representation.

If a permutation $\pi = \sigma_i \sigma_{i+1} \ldots \sigma_m$ (with each $\sigma_i \in U_i$) is an element of G_i, then the representative of the coset containing π is $\sigma_i = T[i][\pi(i)]$, and the permutation

$$\sigma_i^{-1} \pi$$

is an element of G_{i+1}. This observation implies a simple recursive test for membership in G, given in Algorithm 3. Algorithm 3 decomposes its input π into the unique factorization

$$\sigma_1 \sigma_2 \ldots \sigma_m$$

by repeatedly applying the relationship above. If at any step the entry $T[i][\pi(i)]$ is invalid, the decomposition does not exist and therefore π is not a member of G. The initial call to TestMembership in Algorithm 3 will have T set to a Sims table for G (with an associated base $X = k_1, k_2, \ldots, k_m$), π set to the permutation to test and `level` set to 1.

Algorithm 3 Test membership in G with a Sims table

```
1: procedure TestMembership(T, π, level)
2:     if level = m then
3:         return True
4:     end if
5:     if T[level][π(level)] = × then
6:         return False
7:     end if
8:     σ ← T[level][π(level)]
9:     return TestMembership(T, σ^{-1}π, level + 1)
10: end procedure
```

The TestMembership function in Algorithm 3 requires $O(n)$ time at each step, and the depth of recursion is at most $m \leq n$, so the total running time of the membership test is $O(n^2)$.

2.2 The Schreier-Sims Algorithm

The main contribution of Sims' original papers [34, 33] was an algorithm which produces a Sims table for a permutation group $G \leq S_n$ from a generating set in polynomial time in n (when the size of the generating set is also polynomial in n). This algorithm is known as the Schreier-Sims algorithm since the algorithm depends on the following lemma by Schreier, originally published in the context of free groups in [31], which was later incorporated by M. Hall into a section in [17], from which the version given here is paraphrased.

Lemma 2.5 (Schreier's Lemma [17])**.** *Let G be a group generated by a set S and $G' \leq G$. If U is a transversal for G' in G, and for each $g \in G$, $\phi(g)$ is the representative for gG' in*

U, then G' is generated by the set

$$S' = \{\phi(su)^{-1}su | u \in U, s \in S\}.$$

□

The literature contains numerous different formulations of the Schreier-Sims algorithm, ranging from broad theoretical overviews such as that given in Sims' original work [34, 33] and subsequent analyses and improvements [32, 19, 22, 13, 18] to fully specified pseudocode [5, 21]. This section presents a description and analysis of the Schreier-Sims algorithm. The theoretical underpinnings and pseudocode are adapted from the graph-theoretic approach given by Kocay in [21]. The analysis is based on that given by Butler in [5] and Seress in [32]. Butler points out in [5] that there are three common 'strains' of the Schreier-Sims algorithm, all based on Sims' initial work but differing in implementation details. The strain examined here is ascribed in [5] to Furst, Hopcroft and Luks [13]. The other major strains are due to Knuth [19] and Jerrum [18]. The version in [18] is notable in that instead of producing a Sims table, it produces a generating set of size $O(n)$.

Let $G \leq S_n$ be a permutation group and $S = \{\alpha_1, \ldots, \alpha_q\}$ be a generating set for G. Construct an edge-labelled directed multigraph H as follows:

- Add a vertex v_i for each $i \in \{1, 2, \ldots, n\}$.
- For each $\alpha \in S$, add an arc $(i, \alpha(i))$, labelled with α, for every $i \in \{1, 2, \ldots, n\}$.

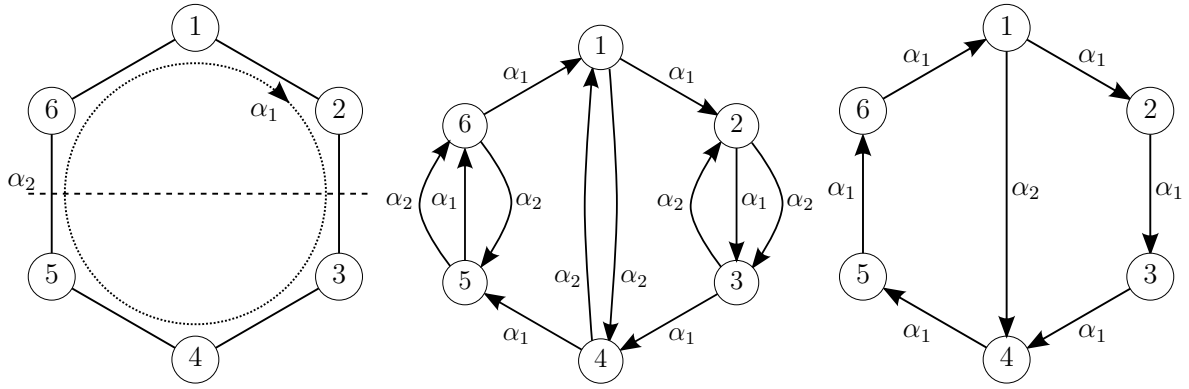
The graph H is called a *permutation diagram*[2] for G and encodes the image of each $i \in \{1, \ldots, n\}$ under products of the generating set. A *simplified permutation diagram* H' is a subgraph of the permutation diagram in which every pair (v_i, v_j) of distinct vertices adjacent in H is connected by exactly one arc (in either direction) in H'. Figure 2.3 shows

[2]The term 'permutation diagram' is used by Kocay in [21] for this construction, even though the term has a different meaning elsewhere in graph theory.

a permutation diagram for the 6-cycle for the generating set $\{\alpha_1, \alpha_2\}$, where

$$\alpha_1 = (123456) \qquad \text{(a rotation), and}$$

$$\alpha_2 = (14)(23)(56) \qquad \text{(a reflection).}$$



(a) C_6, showing generators α_1 and α_2

(b) Permutation Diagram

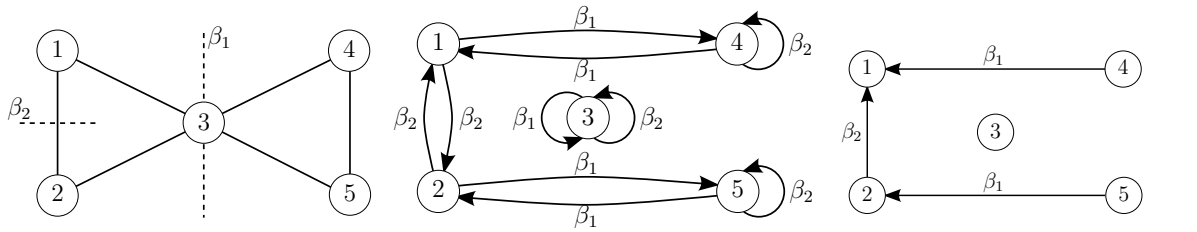
(c) A Simplified Permutation Diagram

Figure 2.3: Simplified and unsimplified permutation diagrams of C_6

Figure 2.4 shows a permutation diagram for the bowtie graph B from Figure 2.2 for the generating set $\{\beta_1, \beta_2\}$, where

$$\beta_1 = (14)(25) \qquad \text{(a horizontal reflection), and}$$

$$\beta_2 = (12) \qquad \text{(a vertical reflection of the left lobe).}$$



(a) B (b) Permutation Diagram (c) A Simplified Permutation Diagram

Figure 2.4: Simplified and unsimplified permutation diagrams of the bowtie graph B.

Every directed path in the permutation diagram H from v_i to v_j corresponds to a product of the generators such that this product maps i to j. The generators in the product can be obtained by reading off the labels of arcs in the path. Conversely, every permutation in G corresponds to at least one directed path in H, since any permutation can be written as a product of the generators of the group, and a sequence of arcs corresponding to the sequence of generators can be followed from any starting vertex in H. This yields a useful immediate result.

Lemma 2.6. *Given a generating set S of size q for a permutation group $G \leq S_n$, the orbit of $i \in \{1, \ldots, n\}$ can be determined in $O(nq)$ time.*

Proof. The orbit of i is the set of all vertices reachable by (undirected) paths from v_i in the simplified permutation diagram H'. For the purpose of finding elements j in the orbit of i, it is not significant exactly which product of generators maps i to j. Arcs of the form (k, k) are similarly irrelevant. The graph H' has a maximum of n^2 arcs. The construction still requires $O(nq)$ time since the image of each vertex under each generator must be determined. After H is constructed, a breadth-first search rooted at v_i will yield the set of all vertices v_j reachable from v_i in $O(m)$ time, where $m = |E(H')|$. The total time is therefore $O(nq + m)$, and since $m \leq nq$, the term nq dominates the sum. □

Further observations about the structure of H allow the set of all orbits to be computed

in $O(nq)$ time. The first step towards such an algorithm is the following fact.

Lemma 2.7. *If H contains a directed path from v_i to v_j, it also contains a path from v_j to v_i. That is, there are no edges between strongly connected components of H.*

Proof. Suppose a directed path from v_i to v_j corresponds to the sequence

$$\pi = \alpha_{p_1}\alpha_{p_2}\ldots\alpha_{p_l}$$

of generators in S. Then $\pi(i) = j$ and the inverse

$$\pi^{-1} = \alpha_{p_l}^{-1}\ldots\alpha_{p_1}^{-1}$$

maps j to i. The inverses of $\alpha_{p_1}, \ldots, \alpha_{p_l}$ may not be in S, but since S is a generating set, the inverse of each α_p is the composition of elements in S, so π^{-1} can be obtained by a path starting at v_j by sequentially following the edges for the sequence of generators corresponding to $\alpha_{p_1}^{-1}, \alpha_{p_2}^{-1}, \ldots, \alpha_{p_l}^{-1}$. □

The orbits of G therefore correspond to the components of H. Using a $O(|E(H)|)$ algorithm to compute components [9] yields the following.

Lemma 2.8. *Given a generating set S of size q for a permutation group $G \leq S_n$, the set of all orbits of G can be computed in $O(nq)$ time.*

Proof. Constructing the simplified permutation diagram H', as in the proof of Lemma 2.6 requires $O(nq)$ time. Identifying all components of H' requires $O(|E(H')|)$ time in total, since a series of breadth-first searches on each component is sufficient, and each traversal will only search the vertices of one component. □

Let T_i be a BFS tree of the component of H containing v_i, rooted at v_i. Define the set P_i to contain all permutations corresponding to paths through T_i starting at v_i, including the

degenerate path from v_i to itself. The set P_i contains exactly one permutation carrying i to each element in its orbit (including itself). Therefore, P_i is a transversal for the stabilizer group $\mathrm{St}(i)$ in G.

Given a base $k_1, \ldots, k_m$ for G, define H_i to be the permutation diagram for G_i. The set P_{k_1} in $G_1 = G$ is a transversal for $\mathrm{St}(k_1) = G_2$. Therefore, a traversal of H_1 rooted at v_{k_1} gives the transversal U_1. Similarly, the set P_{k_i} in G_i is a transversal for $\mathrm{St}(k_1, k_2, \ldots, k_i) = G_{i+1}$, so a traversal of H_i rooted at v_{k_i} gives the set U_i.

To obtain each each H_i, a method is needed to create a generating set for G_{i+1} given a generating set for G_i. Schreier's Lemma (Lemma 2.5) forms the core of such a method, since it gives a construction for a generating set of a subgroup $G' \leq G$ given a generating set for G, a transversal U of G' in G and some function $\phi(g)$ which maps each $g \in G$ to its coset representative in U. As the preceding discussion demonstrates, the transversal U_i of G_{i+1} in G_i can be obtained with the permutation diagram H_i. The coset representative of $g_i \in G_i$ is the permutation $u \in U$ such that $u(k_i) = g(k_i)$. This condition is easy to test, and when the permutations in U are stored as a row of the Sims table, determining whether a representative exists can be done in constant time (specifically, the representative of g is $\sigma_{ig(k_i)}$). Schreier's Lemma can therefore be employed to construct a generating set for G_{i+1}. Algorithm 4 gives a conceptual outline of a recursive algorithm to construct a Sims table from an initial generating set S for G. In Algorithm 4 and the rest of the algorithms in this section, it is assumed for notational clarity that the base $k_1, \ldots, k_m$ is $1, 2, \ldots, n$. In practice, this can always be achieved by relabeling group elements with the mapping $k_i \to i$, and if the base has size $m < n$, adding the missing elements to the end of the sequence.

Kocay [21] presents a graph-based formulation of Schreier's Lemma using the properties of the tree T_i and the (unsimplified) permutation diagram H_i. The elements of G_{i+1} are those permutations π in G_i with $\pi(k_i) = k_i$. Since each π is a product of generators of G_i, it can be represented by a walk in the permutation diagram H_i starting at vertex v_i, and

since it fixes k_i, the walk must also end at v_i. Therefore, the permutations in G_{i+1} are in correspondence with the set of closed walks from v_i to v_i in H_i. Using a previous result that the set of such closed walks are generated by *fundamental cycles* produced by adding single edges to the tree T_i, a set of generators for G_{i+1} can be obtained from walks of the form

$$p_{ik}^{-1} g p_{ij}$$

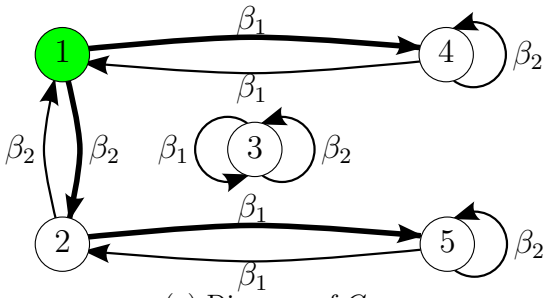
where $p_{ij}, p_{ik} \in P_i$, $p_{ij}(i) = j, p_{ik}(i) = k$ and g is a generator of G_i such that $g(j) = k$.

Algorithm 4 Produce a Sims table from a generating set S

```
1: procedure FillSimsTableRow(T, Q_i, i)
2:     H ← Simplified permutation diagram for S_i
3:     T_i ← Traversal tree for H rooted at v_i
4:     U_i ← Transversal corresponding to paths in T_i
5:     {Fill row i of the Sims table}
6:     for each permutation σ ∈ U_i do
7:         Set T[i][σ(i)] = σ
8:     end for
9:     {Create a generating set for G_{i+1}}
10:    Q_{i+1} ← empty set
11:    for each σ ∈ U_i do
12:        for each π ∈ Q_i do
13:            φ ← T[i][πσ(i)]
14:            q ← φ^{-1}πσ
15:            Add the permutation q to Q_{i+1}
16:        end for
17:    end for
18:    {Recursively populate any remaining rows}
19:    if i < n then
20:        FillSimsTableRow(T, Q_{i+1}, i + 1)
21:    end if
22: end procedure
23: procedure GenerateSimsTable(S)
24:    T ← Empty n × n array, initialized to ×.
25:    FillSimsTableRow(T, S, 1)
26: end procedure
```

Figures 2.5 to 2.9 illustrate Algorithm 4 on the graph B from Figure 2.4 with the base

$X = 1, 2, 3, 4, 5$. At each step i, vertex i is shaded in green, vertices which have already been processed are shaded in grey, edges in the tree T_i are darkened, and edges incident to already-processed vertices are dashed. The figure for step i shows the coset representatives generated at step i (which form the i^{th} row of the Sims table) and the generating set for G_{i+1} produced by applying Schreier's Lemma (Lemma 2.5) to the coset representatives and generators for G_i (in Algorithm 4, the produced generating set is called Q_{i+1}). In Figure 2.5, the generators produced for G_2 are e and the new permutation $\tau = (45)$. Figures 2.6, 2.7 and 2.9 show steps i where the only coset representative is the identity, since $G_i = G_{i+1}$.



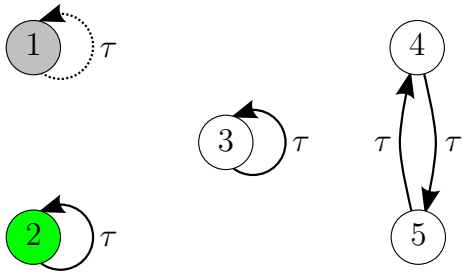
(a) Diagram of G_1

σ_{11}	$= e$	$\sigma_{11}^{-1}\beta_1\sigma_{14}$	$= e$
σ_{12}	$= \beta_2$	$\sigma_{14}^{-1}\beta_2\sigma_{14}$	$= (45) = \tau$
σ_{14}	$= \beta_1$	$\sigma_{11}^{-1}\beta_2\sigma_{12}$	$= e$
σ_{15}	$= \beta_1\beta_2$	$\sigma_{12}^{-1}\beta_1\sigma_{15}$	$= e$
		$\sigma_{15}^{-1}\beta_2\sigma_{15}$	$= \tau$

(b) Coset Representatives

(c) Generators of G_2

Figure 2.5: Schreier-Sims algorithm on the bowtie graph B - Step 1.



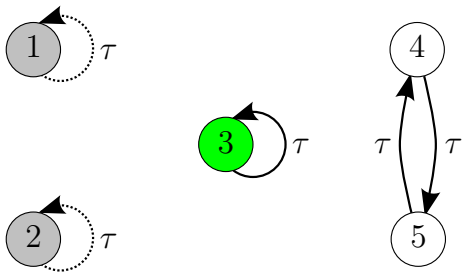
(a) Diagram of G_2

σ_{22}	$= e$	$\sigma_{22}^{-1}\tau\sigma_{22}$	$= \tau$

(b) Coset Representatives

(c) Generators of G_3

Figure 2.6: Schreier-Sims algorithm on the bowtie graph B - Step 2.



(a) Diagram of G_3

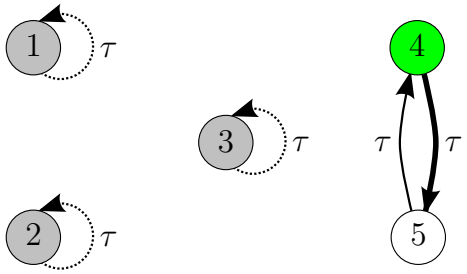
(b) Coset Representatives

$\sigma_{33} = e$

(c) Generators of G_4

$\sigma_{33}^{-1}\tau\sigma_{33} = \tau$

Figure 2.7: Schreier-Sims algorithm on the bowtie graph B - Step 3.



(a) Diagram of G_4

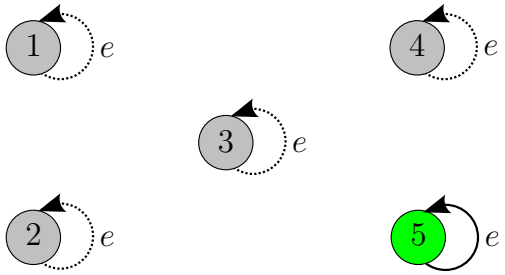
(b) Coset Representatives

$\sigma_{44} = e$
$\sigma_{45} = \tau$

(c) Generators of G_5

$\sigma_{44}^{-1}\tau\sigma_{45} = e$

Figure 2.8: Schreier-Sims algorithm on the bowtie graph B - Step 4.



(a) Diagram of G_5

(b) Coset Representatives

$\sigma_{22} = e$

(c) Generators of G_6

$\sigma_{55}^{-1}e\sigma_{55} = e$

Figure 2.9: Schreier-Sims algorithm on the bowtie graph B - Step 5.

The key difficulty in implementing Algorithm 4 lies in ensuring that the size of the generating set Q_i for each level is polynomially bounded. In the case $G = S_n$, we can take

the set $S = \{\tau, \sigma_{12}\}$ as the initial generating set Q_1, where

$$\tau = (123 \ldots n) \qquad \text{and} \qquad \sigma_{12} = (12)$$

(as discussed at the beginning of this chapter). Since $|G| = n!$, every row of the Sims table must be full (that is, row i must contain $n - i + 1$ valid entries). The size of the generating set Q_{i+1} produced by the FillSimsTableRow function in Algorithm 4, which does not attempt to eliminate redundant elements, is

$$|Q_{i+1}| = |Q_i| \cdot |U_i| = |S| \cdot \prod_{j=1}^{i} |U_j|$$

and when $G = S_n$, the above formula gives

$$|Q_n| = 2(n-1)!.$$

Although algorithm 4 is exponential in the worst case, it can be refined to produce a polynomial-time algorithm. Three improvements can be made to improve the practical performance of Algorithm 4:

- Instead of building the simplified permutation diagram and computing a traversal tree explicitly to set entries of the Sims table T, the tree can be generated implicitly by repeatedly applying generators to each entry of row i of T to compute new coset representatives.

- Instead of recursing on an explicit generating set Q_{i+1}, recurse on each generator q, progressively building the group G_{i+1} as new generators are added.

- When a new permutation $q \in Q_{i+1}$ is computed, test whether it is already a member of G_{i+1} before adding it as a generator.

To test membership in G_{i+1} the improved algorithm can use the internal TestMembershipRecursive function from Algorithm 3 directly, by providing $i + 1$ as the `level` parameter. The traversal can be performed implicitly by initially setting $T[i][i]$ to the identity and leaving the rest of row i invalid. As each generator for G_i is provided, it is applied to each element of row i of T to produce new coset representatives. New representatives are in turn recursively added as generators for G_i. Finally, Schreier's Lemma (Lemma 2.5) is applied to produce a set of potential generators of G_{i+1}, and if any of the produced permutations are not already elements of G_{i+1}, they are recursively added as generators of G_{i+1}. This modification requires a list of the known generators for each level G_i to be maintained.

The core of the improved algorithm is a function called AddToLevel which adds a new generator g to level i of the table.

The complete pseudocode for the improved approach is summarized in Algorithm 5. The algorithm is initiated by a call to GenerateSimsTable with a set S of generators for G.

Algorithm 5 Produce a Sims table from a generating set S (improved), revised from [21]

```
1: procedure AddToLevel(T, L, i, g)
2:     new_perms ← Empty list of permutations.
3:     {Apply the new generator to every entry of the table}
4:     for j = i, i + 1, ..., n do
5:         if T[i][j] ≠ × then
6:             σ ← T[i][j]
7:             π ← gσ
8:             k ← π(i)
9:             if T[i][k] = × then
10:                T[i][k] ← π
11:                Append π to new_perms
12:            else
13:                φ ← T[i][k]
14:                q ← φ^{-1}π
15:                if TestMembershipRecursive(T, q, i + 1) = false then
16:                    AddToLevel(T, L, i + 1, q)
17:                end if
18:            end if
19:        end if
20:    end for
21:    Append g to L[i]
22:    for π in new_perms do
23:        for τ in L[i] do
24:            k ← πτ(i)
25:            if T[i][k] = × then
26:                T[i][k] = ττπ
27:                Append π to new_perms
28:            else
29:                φ ← T[i][τπ(i)]
30:                q ← φ^{-1}ττπ
31:                if TestMembershipRecursive(T, q, i + 1) = false then
32:                    AddToLevel(T, L, i + 1, q)
33:                end if
34:            end if
35:        end for
36:    end for
37: end procedure
```

```
38: procedure GenerateSimsTable(S)
39:     T ← Empty n × n array, initialized to ×.
40:     Set each diagonal entry T[i][i] to the identity permutation e.
41:     L ← Array of n empty lists of permutations.
42:     for each generator g in S do
43:         AddToLevel(T, L, 1, g)
44:     end for
45: end procedure
```

Chapter 3

Distinguishing Colourings

Albertson and Collins, in [1], considered the problem of colouring the vertices of a graph G such that no non-trivial automorphism of the graph fixes the colouring. Such a colouring is called a *distinguishing colouring*, and the minimum number of colours k such that a distinguishing colouring exists is called the *distinguishing number* of G and denoted $D(G)$. If $D(G) \leq k$, G is said to be k-distinguishable.

Besides formalizing the concept of a distinguishing colouring, the initial paper by Albertson and Collins presented proofs of basic bounds on distinguishing colourings for various families of graphs. The paper's results drew interest from several different sub-areas of mathematics, and the major research results which ensued can be grouped into the following categories:

- Finding distinguishing numbers for individual graphs or infinite families with theoretical methods.
- Applied computational approaches to evaluating the distinguishing number of a graph.
- General bounds on distinguishing numbers.
- Computational problems related to distinguishability, including complexity results.

- Generalizations of distinguishability.
- Relationships between graph distinguishability and group distinguishability.

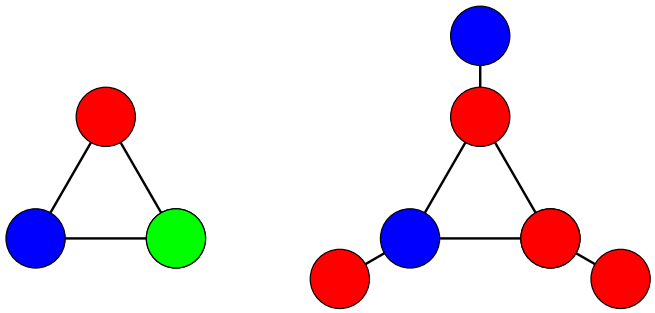
This chapter focuses mainly on bounds and the computational complexity of distinguishability and related problems. Section 3.2 describes various bounds on distinguishing numbers, including two new bounds, and Section 3.3 discusses the computation of the distinguishing number and complexity results. Section 3.1 summarizes the generalization of distinguishing numbers from graphs to groups and Section 3.4 describes the problem of list-distinguishing colouring, which is a higher-dimensional analogue of distinguishing colouring.

3.1 Definitions of the Distinguishing Number of a Group

The automorphism group of a graph G on n vertices is denoted by $\mathrm{Aut}(G)$. For any graph G, $\mathrm{Aut}(G)$ is a subgroup of S_n, and the elements of $\mathrm{Aut}(G)$ will therefore be permutations of the n vertices of G. The algorithms for distinguishing colourings will focus on colourings of the vertices of G, which correspond to labellings of elements of the permutation group. The distinguishing number of a graph G is therefore a single well-defined value. The distinguishing number of a group $H \leq S_n$ can be similarly defined as the minimum value of k for which there exists a k-colouring $C = c_1c_2 \ldots c_n$ such that no non-identity permutation in H fixes C. Under this interpretation, C is an assignment of colours to the values $\{1, 2, \ldots, n\}$ permuted by the group, rather than vertices of a graph. This definition ensures that the distinguishing number of an arbitrary permutation group H is also a single, well-defined value, and will agree with the distinguishing number of any graph G whose automorphism group is identical to H. In this thesis, with the exception of the current section, the distinguishing number of a group will always use the definition given above.

However, the above interpretation is not shared by all of the literature on distinguishability, and some important results use a less strict definition of the distinguishing number of

a group which allows a group to have more than one distinguishing number. The motivation for the alternative definition lies in the semantic differences between graph isomorphism and group isomorphism. If two graphs are isomorphic, they must have the same number of vertices and edges, as well as the same automorphism group. It is unsurprising that these two graphs will have the same distinguishing number. Two non-isomorphic graphs G_1 and G_2 may have isomorphic automorphism groups. In this case, it is possible that $D(G_1) = D(G_2)$ (for example, when $G_1 = K_n$ and $G_2 = \overline{K_n}$), but not necessary. Figure 3.1 shows two graphs whose automorphism group is isomorphic[1] to S_3, but which have different distinguishing numbers.



(a) G_1 (b) G_2

Figure 3.1: Two graphs G_1 and G_2 whose automorphism groups are isomorphic to S_3, with minimum distinguishing colourings shown. $D(G_1) = 3$ and $D(G_2) = 2$

A graph G is said to *realize* a permutation group H if $\mathrm{Aut}(G)$ is isomorphic to H.

The definition of group distinguishability given above can be relaxed in two ways. First, the distinguishing number of a group H can be broadened to a set of all distinguishing numbers of graphs realizing H. Second, the restriction to graphs can be removed altogether, and instead, the action of H on an arbitrary set X can be considered, with the resulting distinguishing number denoted $D_H(X)$.

The minimum achievable distinguishing number of a non-trivial group is 2. Albertson

[1]The definition of 'isomorphism' used here is the general definition of group isomorphism given in Section 1.1.3, not the more specific definition often applied to permutation groups.

and Collins, in [1], extended a construction originally presented by Frucht [12] to produce, given any group H, a graph G with $D(G) = 2$ which realizes H. Therefore, every group can achieve the minimum possible distinguishing number.

Upper bounds on the distinguishing number of arbitrary groups have also been studied. Tymoczko [35] proved the following theorem, generalizing an unpublished result of Albertson, Collins and Kleitman.

Theorem 3.1. *Let $H \leq S_n$ and let X be a set acted on by H. If*

$$|H| \leq k!$$

for some k, then

$$D_H(X) \leq k.$$

□

Chan [6] extended Tymoczko's results to prove upper bounds for certain classes of groups.

3.2 Bounds on Distinguishing Numbers

Albertson and Collins' original paper [1] proved several bounds on the distinguishing number of certain families of graphs, as well as group-theoretic conditions affecting the distinguishing number. For the purposes of this thesis, four of the results in [1] are of particular relevance.

Theorem 3.2 (Albertson and Collins [1])**.** *The distinguishing number of K_n is n.*

□

Theorem 3.3 (Albertson and Collins [1])**.** *Let G be a graph on n vertices. Then*

$$D(G) \leq \lfloor \log_2(|Aut(G)|) \rfloor + 1.$$

□

Theorem 3.4 (Albertson and Collins [1]). *Let G be a graph on n vertices, $U = \{u_1, \ldots, u_q\}$ be an orbit of Aut(G) and H be the subgraph of G induced by U. If no automorphism of G fixes H, then*

$$D(G) \leq D(H).$$

Proof. Suppose $D(H) = k$. A k-distinguishing colouring of G can be constructed by labelling the vertices $u_1, u_2, \ldots, u_q$ with a k-distinguishing colouring of H, and labelling all other vertices with colour 0. All non-identity permutations $\pi \in \mathrm{Aut}(G)$ move at least one vertex u_i (since no permutation fixes H besides the identity). Since the vertices $u_1, u_2, \ldots, u_q$ have been labelled with a distinguishing colouring, the permutation π must map at least one u_i to a vertex u_j with a different colour. □

Theorem 3.5 (Albertson and Collins [1]). *Let G be a graph on n vertices. If $U = \{u_1, \ldots, u_q\}$ is a collection of vertices of G, with no two in the same orbit of Aut(G), and no non-trivial automorphism of G fixes U, then*

$$D(G) = 2$$

Proof. Assign colour 0 to each vertex u_i and colour 1 to all other vertices. Let π be a non-trivial automorphism of G. The permutation π must move at least one u_i. None of the other vertices in the orbit of u_i are the same colour, so u_i must be mapped onto a vertex of a different colour. Therefore, no non-identity permutation in $\mathrm{Aut}(G)$ fixes the colour of every vertex. □

The distinguishing number of the hypercube Q_n was found for all n by Bogstad and Cowen [4]. For $n = 2, 3$, $D(Q_n) = 3$, and for all $n \geq 4$, $D(Q_n) = 2$. Bogstad and Cowen also proved results about powers of hypercubes which were later extended by Chan [7] to fully classify the distinguishing number of powers of hypercubes. The following theorem due to Tymoczko [35] establishes a key property of the distinguishing number of trees.

Theorem 3.6 (Tymoczko [35]). *If T is a tree on $n \geq 3$ vertices with maximum degree d, then $D(T) \leq d$.*

□

Potanka [28] presents a study of graphs whose automorphism group is isomorphic to the dihedral group D_n, building on previous results by Albertson and Collins [1]. Potanka also classifies the distinguishing number of all generalized Petersen graphs. Russell and Sundaram [30] provide sufficient conditions for distinguishability based on the structure of the elements of the automorphism group. For a graph G, the *motion* of a permutation $\pi \in \mathrm{Aut}(G)$ is defined to be

$$\mathrm{Motion}(\pi) = |\{v \in G : \pi(v) \neq v\}|$$

which is the number of points not fixed by π. The motion of the graph G is defined to be the minimum motion of any non-identity permutation.

Theorem 3.7 (Russell and Sundaram [30]). *If $Motion(G) > 2\log_2(Aut(G))$, then*

$$D(G) = 2.$$

□

If $\pi \in \mathrm{Aut}(G)$ is a permutation whose disjoint-cycle representation contains q cycles with sizes $s_1, \ldots, s_q$, the *cycle norm* of π is defined to be

$$\mathrm{CycleNorm}(\pi) = \sum_{i=1}^{q} s_q - q$$

and the cycle norm of G is defined to be the minimum cycle norm of a non-identity permutation in $\mathrm{Aut}(G)$.

Theorem 3.8 (Russell and Sundaram [30]). *If CycleNorm*$(G) > \log_k(Aut(G))$*, then*

$$D(G) \leq k.$$

□

Chan [6] gives a sufficient condition for distinguishability based on the divisors of the size of Aut(G) and the size of its largest orbit.

Theorem 3.9 (Chan [6]). *Let G be a graph on n vertices. If M is the size of the largest orbit of Aut(G) and p is the smallest prime dividing $|Aut(G)|$, then*

$$D(G) \leq \left\lceil \frac{M}{p-1} \right\rceil.$$

□

Using counting techniques, we will prove two new bounds on the distinguishing number of a graph G. Theorem 3.10 gives a lower bound on the distinguishing number of a graph based on the size of its automorphism group, and Theorem 3.12 gives an upper bound on the distinguishing number of a graph based on the number of equivalence classes induced by the action of the automorphism group on the set of all colourings.

Theorem 3.10. *Let G be a graph on n vertices. Let C be a k-colouring of the vertices of G with l_i counting the number of vertices receiving colour i. If C is a distinguishing colouring of G, then*

$$|Aut(G)| \leq \binom{n}{l_1, l_2, \ldots, l_k}.$$

Proof. Suppose C is a distinguishing colouring. Then each permutation in Aut(G) carries C to a distinct colouring with the same distribution of colours. By the pigeonhole principle,

the size of $\mathrm{Aut}(G)$ can be no greater than the number of such colourings, which is

$$\binom{n}{l_1, \ldots, l_k}.$$

□

Theorem 3.10 can be applied to construct a lower bound for $D(G)$ by considering the largest value of the multinomial coefficient

$$\binom{n}{l_1, l_2, \ldots, l_k}$$

for each k. The largest coefficient, which corresponds to cases where the colour classes are as even in size as possible, can be constructed explicitly to give the following corollary.

Corollary 3.11. *If $D(G) \leq k$, then*

$$|Aut(G)| \leq \binom{n}{l_1, l_2, \ldots, l_k}$$

where

$$l_i = \begin{cases} \lceil \frac{n}{k} \rceil & \text{if } i \leq n \bmod k, \text{ and} \\ \lfloor \frac{n}{k} \rfloor & \text{otherwise.} \end{cases}$$

Theorem 3.12. *Let G be a graph on n vertices and $k \geq 1$. If q is the number of equivalence classes of the k^n k-colourings of G under the action of $Aut(G)$, and*

$$q < \frac{2k^n}{|Aut(G)|}$$

then

$$D(G) \leq k.$$

Proof. Consider a k-colouring C of G. Let S contain all permutations in $\mathrm{Aut}(G)$ which fix C. Note that S is a subgroup of $\mathrm{Aut}(G)$ since it is closed under permutation composition and for every $\alpha \in S$, the inverse permutation α^{-1} is also a member of S. Therefore, by Theorem 1.1, $|S|$ divides $|\mathrm{Aut}(G)|$. Since each $\alpha \in S$ has the property $C_\alpha = C$, for any $\pi \in \mathrm{Aut}(G)$, $C_{\pi\alpha} = C_\pi$, so the equivalence class of C under the action of $\mathrm{Aut}(G)$ contains one colouring for each coset of S in $\mathrm{Aut}(G)$.

If C is distinguishing, then $S = \{e\}$. Otherwise, $|S| \geq 2$ and the equivalence class of C has size

$$\frac{|\mathrm{Aut}(G)|}{|S|} \leq \frac{|\mathrm{Aut}(G)|}{2}$$

The union of all equivalence classes must equal the complete set of k-colourings. If there are q equivalence classes of k-colourings with sizes $s_1, \ldots, s_q$, then

$$\sum_{i=1}^{q} s_i = k^n.$$

When no distinguishing colourings exist, every s_i must comply with the bound above, so

$$\sum_{i=1}^{q} s_i = k^n \leq q\frac{|\mathrm{Aut}(G)|}{2}.$$

Therefore, if

$$q\frac{|\mathrm{Aut}(G)|}{2} < k^n,$$

there must exist at least one k-distinguishing colouring of G. □

We conclude this section by proving an extension of Theorem 3.2 to classify the distinguishing number of graphs consisting of multiple copies of a complete graph K_n.

Theorem 3.13. *If G is a graph comprising q disjoint copies of K_n (for some $n \geq 1$), then*

$D(G)$ is equal to the least integer $k \geq n$ such that

$$q \leq \binom{k}{n}.$$

Proof. By Theorem 3.2, $D(K_n) = n$. Within each copy of K_n, for every pair v_1, v_2 of vertices there exists a permutation in $\mathrm{Aut}(G)$ which exchange v_1 and v_2 while fixing the rest of G. Therefore, any distinguishing colouring of G must assign distinct colours to each vertex within a copy of K_n. To prevent an automorphism from exchanging two copies of K_n, it is necessary for each copy of K_n to use a different subset of n-colours. A distinguishing colouring must therefore use at least k colours, with k defined as above, to allow each copy of K_n to receive a distinct set of colours.

A k-distinguishing colouring can be constructed by choosing q distinct n-subsets of $\{1, \ldots, k\}$ and assigning the colours in each subset to a copy of K_n. □

3.3 Computation of the Distinguishing Number

This section summarizes results on the computation of distinguishing numbers, including a theoretical treatment of the computational problems surrounding distinguishability.

3.3.1 Previous Results and Asymptotic Bounds

If G is a graph on n vertices, then $D(G) \leq n$. A brute force method to compute $D(G)$ tests all possible colourings on $1, 2, \ldots, n$ colours against all automorphisms of G until a distinguishing colouring is found. If the time required to produce a set of generators for $\mathrm{Aut}(G)$ is given by a function $p(n)$, the brute force approach requires

$$O\left(p(n) + n^5 + n|\mathrm{Aut}(G)| \sum_{k=1}^{D(G)} k^n\right)$$

where n^5 is the time complexity of Algorithm 5, which produces a Sims table for the permutation group, and the permutations in $\mathrm{Aut}(G)$ are produced by Algorithm 2 in $O(n|\mathrm{Aut}(G)|)$ time.

Further to the results of Tymoczko [35] on the distinguishing number of trees (see Theorem 3.6), Cheng [8] formulated an algorithm to compute the distinguishing number of a tree on n vertices in $O(n \log n)$ time using a recursive counting method. Arvind, Cheng and Devanur, in [3], use inclusion-exclusion techniques as the basis for an $O(n^5(\log n)^3)$ algorithm to find the distinguishing number of any planar graph. The counting methods in [3] are applicable to general graphs, but not in polynomial-time formulations. Using the inclusion-exclusion formula given in [3], it may be necessary to evaluate $2^{|\mathrm{Aut}(G)|}$ terms. Using Möbius inversion, a second formula is given in which the number of terms is proportional to the number of subgroups of $\mathrm{Aut}(G)$. Finally, a specialization of the Möbius formula is given which implies the following theorem.

Theorem 3.14 (Arvind, Cheng, and Devanur [3])**.** *Let G be a graph on n vertices and let $S \subseteq Aut(G)$ be a set of non-identity permutations such that for every subgroup $H \leq Aut(G)$, $H \cap S \neq \emptyset$. Then $D(G)$ can be computed in time proportional to $n2^{|S|}$.*

□

The minimum size of the set S in Theorem 3.14 is the maximum size of a collection of subgroups of $\mathrm{Aut}(G)$ such that the intersection of any pair contains only the identity. Since the size of any subgroup $H \leq \mathrm{Aut}(G)$ must divide $|\mathrm{Aut}(G)|$, and since the group generated by two disjoint subgroups $H_1, H_2 \leq \mathrm{Aut}(G)$ has size $|H_1||H_2|$ by [14], the size of a collection of disjoint subgroups of $\mathrm{Aut}(G)$ is bounded above by $\log_2 |\mathrm{Aut}(G)|$. Therefore, a set S of size at most $\log_2 |\mathrm{Aut}(G)|$ which satisfies the conditions of Theorem 3.14 must exist for any graph G, enabling the distinguishing number to be computed in $O(n|\mathrm{Aut}(G)|)$ time.

However, as Arvind, Cheng and Devanur [3] point out, finding a minimal set S is equivalent to the 'hitting-set' problem, which is NP-complete.

3.3.2 Complexity Results

The core computational problem associated with the distinguishing number of a graph G is the computation of $D(G)$, formalized below.

k-DISTINGUISHING
Input: A graph G and integer k.
Question: Is $D(G) \leq k$?

This section summarizes related problems and relevant results, including two theorems which classify the complexity of k-DISTINGUISHING. To discuss the computational complexity of distinguishability, we begin by defining several related computational problems.

GRAPH AUTOMORPHISM
Input: A graph G.
Question: Does Aut(G) contain a non-trivial permutation?

GRAPH AUTOMORPHISM GROUP
Input: A graph G.
Output: A polynomially sized set of generators for Aut(G).

GRAPH ISOMORPHISM
Input: Graphs G_1 and G_2.
Question: Is G_1 isomorphic to G_2?

GROUP INTERSECTION
Input: Groups $H_1, H_2 \leq S_n$, provided as polynomially sized sets of generators.
Output: A polynomially sized set of generators for $H_1 \cap H_2$.

TRIVIAL GROUP INTERSECTION
Input: Groups $H_1, H_2 \leq S_n$, provided as polynomially sized sets of generators.
Question: Does $H_1 \cap H_2 = \{e\}$?

DIST COLOURING
Input: A graph G and a colouring C of the vertices of G.
Question: Does C distinguish G?

Luks [24] proves that GRAPH AUTOMORPHISM GROUP is equivalent to GRAPH ISOMORPHISM with a reduction that uses repeated yes/no queries to the GRAPH ISOMORPHISM to find generators for the automorphism group of the input graph to GRAPH AUTOMORPHISM GROUP. Luks also proves in [24] that GRAPH ISOMORPHISM is reducible to GROUP INTERSECTION and several other group-theoretic problems. It is known that GRAPH AUTOMORPHISM is reducible to GRAPH ISOMORPHISM, but no reverse reduction has been published. The GRAPH ISOMORPHISM problem is in the class NP, but whether it is NP-complete, NP-intermediate (in NP but not in P) or in P is unknown. The problem of deciding whether a colouring is *not* distinguishing, $\overline{\text{DIST COLOURING}}$, is in NP, since a certificate consisting of an automorphism of the input graph G which fixes the input colouring C can be verified in polynomial time. This section presents a revised proof of a theorem originally by Russell and Sundaram [30] which

proves that $\overline{\text{DIST COLOURING}}$ is 'Automorphism-complete' (using the terminology of Lubiw [23]).

Since the distinguishing number of any graph G on n vertices is at most n, at most n queries to k-DISTINGUISHING are needed to determine $D(G)$. Similarly, since k-DISTINGUISHING can be reduced to querying DIST COLOURING with a k-distinguishing colouring, k-DISTINGUISHING is at most one level above DIST COLOURING in the polynomial hierarchy. Therefore, since DIST COLOURING lies in coNP = Π_1^p, k-DISTINGUISHING lies in Σ_2^p.

Russell and Sundaram [30] prove various complexity results about distinguishability, with the primary result that k-DISTINGUISHING lies in the complexity class AM, which contains problems decidable by a probabilistic polynomial time verifier after a single interaction with a computationally unbounded prover. The specific details of problems in AM are beyond the scope of this thesis, but sources such as [2] contain a thorough treatment. The reduction in [30] depends on a reduction of $\overline{\text{DIST COLOURING}}$ to GRAPH AUTOMORPHISM which contains an error. That error is rectified here to prove that $\overline{\text{DIST COLOURING}}$ is Automorphism-complete. It is unclear whether there are any other errors in [30] which affect their proof of membership in AM, although the article does make the repeated claim that AM $\subseteq \Sigma_2^p \cap \Pi_2^p$, which is not appropriately justified[2] and may be spurious or result from the use of a non-standard definition of the class AM.

Theorem 3.15 (Russell and Sundaram [30], proof expanded)**.** $\overline{\text{DIST COLOURING}}$ *is Automorphism-complete.*

Proof. Given a graph G on n vertices $\{v_1, \ldots, v_n\}$ and a k-colouring $C = c_1 c_2 \ldots c_n$ of the vertices of G, where $k \leq n$ and the colour values c_i are in the range $0, \ldots, k-1$, construct a new graph G' as follows:

[2]Relevant citations for the claim in [30] cite papers on other probabilistic complexity classes and say that a proof of the claim can be made with similar constructions, but the claim itself is not proven or repeated in other complexity literature, including more recent publications.

- For each vertex v of G, construct a graph H_v consisting of a q-clique where $q = n + c_v$, with vertices numbered $h_1, \ldots, h_q$. To each vertex h_i, attach a path of length i. The total number of vertices in H_v is at most $(n + c_v)^2 \leq (n + n)^2 \in O(n^2)$.

- For each edge $uv \in E(G)$, add edges between every vertex of the central cliques of H_u and H_v.

None of the subgraphs H_v have any non-trivial internal automorphisms, since the clique in H_v must map back to itself, and each vertex in the clique has a path of different length attached to it. Therefore, if the graph G' has any non-trivial automorphisms, they must map one of the subgraphs H_v to some other subgraph H_u, for $u \neq v$. Since $H_u \cong H_v$ only if $c_u = c_v$, any automorphism of G' must correspond to a colour-preserving automorphism of G, so if G' has a non-trivial automorphism, C is not distinguishing. Conversely, if C is distinguishing, there cannot exist a non-trivial automorphism of G', since such an automorphism would carry some H_v to an H_u of a different size. Therefore, a single query to GRAPH AUTOMORPHISM on the graph G' (which has $O(n^3)$ vertices) is sufficient to solve $\overline{\text{DIST COLOURING}}$.

The reverse reduction is trivial. An instance of GRAPH AUTOMORPHISM can be solved by querying $\overline{\text{DIST COLOURING}}$ with $C = 00\ldots0$. □

The reduction in the proof of Theorem 3.15 implies an algorithm to test whether a colouring is distinguishing with a graph automorphism algorithm. However, the graph produced by the reduction is very large and may be too cumbersome for practical use. The GROUP INTERSECTION problem is computationally more difficult than GRAPH AUTOMORPHISM [24], but backtracking algorithms have been formulated which are very fast in practice [32]. Theorem 3.16 gives a reduction from DIST COLOURING to GROUP INTERSECTION for which the inputs to GROUP INTERSECTION are not polynomially larger than the inputs to DIST COLOURING, which may lead to a more practical algorithm than the reduction to GRAPH AUTOMORPHISM.

Theorem 3.16. DIST COLOURING *is polynomial-time reducible to* GROUP INTERSECTION.

Proof. Given a graph on n vertices, a polynomial-sized set of generators for $\mathrm{Aut}(G)$ can be found with an instance of GRAPH AUTOMORPHISM GROUP (which can be reduced to GROUP INTERSECTION [24]).

Given a colouring $C = c_1 \dots c_n$ of the vertices of G, where the colours c_i are assumed to be in the range $0, \dots, k-1$ for some $k \leq n$, let $F \leq S_n$ be the group of all permutations which fix C. No element of F carries an element of one colour class to another, but every possible permutation of elements within colour classes is in F. If the number of elements receiving colour i is denoted by s_i, the group F is isomorphic to a direct product

$$S_{s_1} \times S_{s_2} \times S_{s_3} \times \ldots \times S_{s_k}$$

Since the symmetric group can be represented by a generating set of size 2, a generating set of size $2k$ can be created for F.

If C is a distinguishing colouring, no permutation which fixes C (besides the identity) can lie in $\mathrm{Aut}(G)$, so $\mathrm{Aut}(G) \cap F$ must equal $\{e\}$. A query to GROUP INTERSECTION can be used to find a set of generators for $\mathrm{Aut}(G) \cap F$ and return that C is distinguishing if the only generator is e. □

The decision problem TRIVIAL INTERSECTION, which tests whether the intersection of two permutation groups H_1 and H_2 is the trivial group $\{e\}$, can be easily reduced to GROUP INTERSECTION, since the intersection is trivial if its only generator is e. In the reduction given in the proof of Theorem 3.16, no information is needed about the intersection $\mathrm{Aut}(G) \cap F$ except whether or not it is trivial. Although it is not possible to replace the GROUP INTERSECTION instance at the beginning of the reduction (which computes $\mathrm{Aut}(G)$) with a TRIVIAL INTERSECTION query, a formulation of DIST

COLOURING in which a polynomial-sized generating set for Aut(G) is provided can be reduced to a single TRIVIAL INTERSECTION query. None of the literature surveyed for this thesis discussed the complexity of the TRIVIAL INTERSECTION problem, but it may be possible to adapt a fast backtracking algorithm for GROUP INTERSECTION to give a practical test for trivial intersections.

3.4 List-distinguishing Colourings

In 2011, Ferrara, Flesch and Gethner, in [10], formalized a generalized version of the distinguishing colouring problem. Given a graph G on n vertices, an assignment $L = L_1, \ldots, L_n$ of lists[3] L_i with $|L_i| = k$ is a *list-distinguishing colouring* of order k if G has a distinguishing colouring $C = c_1 \ldots c_n$, with $c_i \in L_i$. The *list-distinguishing number* of G, denoted $D_l(G)$, is the minimum k such that every assignment of lists of size k yields such a colouring.

Besides defining list-distinguishability, [10] proved that for graphs G which realize a dihedral group, $D_l(G)$ is always equal to $D(G)$. In [11], Ferrara, Gethner, Hartke, Stolee and Wenger prove, among other things, that $D_l(T) = D(T)$ for all trees T by extending the approach used by Cheng in [8]. Whether $D_l(G) = D(G)$ for all graphs G remains an open question, and was the initial motivation for the research in this thesis.

[3]The 'lists' used for list-distinguishing colouring are sets. The choice of the term 'list' seems to be aesthetic.

Chapter 4

Generating Colourings up to Isomorphism

This chapter presents a backtracking algorithm to generate the lexicographically least representative of each equivalence class of the k-colourings of a graph G under the automorphisms of the graph. Section 4.1 establishes foundations for counting the equivalence classes. A previously published algorithm is described briefly in Section 4.2. The new algorithm is described in Section 4.3, which also contains the theoretical background to the generation process. Practical optimizations to the algorithm are described in Section 4.4 and applications of the algorithm to other labelling problems, including distinguishing colouring, are described in Section 4.5.

4.1 Counting Minimal Colourings

The number of equivalence classes of the k-colourings of a group $G \leq S_n$ can be counted with a theorem traditionally called 'Burnside's lemma', which is actually due to Cauchy.

Theorem 4.1 (Burnside's lemma [36])**.** *Let $G \leq S_n$ and $k \geq 1$. The number of equivalence*

classes of the k^n colourings of G is

$$\frac{1}{|G|}\sum_{g\in G} Fix(g)$$

where $Fix(g)$ is the number of k-colourings fixed by the permutation g.

□

Evaluating the sum in Burnside's lemma was proven to be #P-Hard by Goldberg in [16]. The class #P contains the counting analogue of decision problems in NP (for example, 'How many solutions does the 3-CNF formula ϕ have?' instead of 'Does ϕ have a solution?'), and is at least as hard as any decision problem in the polynomial hierarchy [2]. The author reserves judgement on whether an efficient algorithm to count equivalence classes exists, but notes that such an algorithm would require the collapse of the polynomial hierarchy. In this chapter, it is sufficient to rely on the following bounds for the number of equivalence classes.

Lemma 4.2. *If G is a non-trivial subgroup of S_n, the number of lexicographically minimum k-colourings is at least*

$$\binom{n+k-1}{n}$$

and at most

$$\frac{k^n + k^{n-1}}{2}.$$

Proof. If $G = S_n$, the minimum colourings are the set of strings

$$1^{l_1}2^{l_2}\ldots k^{l_k}$$

where $l_1 + l_2 + \ldots + l_k = n$. Since the values l_i completely determine the string, the number of possible strings is the number of ways n can be written as an ordered sum of the set of l_i

values, which is

$$\binom{n+k-1}{n}.$$

The smallest non-trivial subgroup of S_n is the group $\langle\sigma\rangle$ generated by a transposition σ. The $n-2$ elements fixed by σ can be coloured in k^{n-2} ways. Since $\langle\sigma\rangle$ is isomorphic to S_2, the upper bound proven above gives that the number of ways to colour the 2 elements transposed by σ is

$$\binom{2+k-1}{2} = \binom{k+1}{2} = \frac{k(k+1)}{2} = \frac{k^2+k}{2}$$

giving a total of

$$k^{n-2}\frac{k^2+k}{2} = \frac{k^n+k^{n-1}}{2}$$

colourings. □

4.2 A Previous Generation Algorithm

A backtracking algorithm to generate all independent sets of a graph up to isomorphism was proposed by Myrvold and Fowler in [27]. With minor adjustments, the algorithm in [27] can also generate all colourings with k symbols up to isomorphism. The algorithm stores the automorphism group $G \leq S_n$ as a list of permutations (requiring $\Omega(n|G|)$ space), making it unsuitable for groups too large to fit in memory. However, when the group size is small, the relative simplicity of the algorithm enables very high performance, as shown in the benchmarks of Chapter 5.

4.3 Sims Table Backtracking Generation Algorithm

Using a Sims table representation of the group $G \leq S_n$, a new colouring algorithm was developed, which requires $O(n^3)$ space and can therefore be used with large groups. The developed algorithm is summarized at the end of this section as Algorithm 8. Improvements to Algorithm 8 for certain types of groups are studied in Section 4.4. Note that all of the algorithms in this section use 1-based indexing for all arrays.

Algorithm 8 recursively traverses a Sims table of G, choosing a colour for one vertex at each recursive step until all vertices are coloured. It then outputs those colourings which are lexicographically minimum. The algorithm hinges on the following lemma, which provides a foundation for the backtracking process.

Lemma 4.3 (Prefix condition)**.** *Let $G \leq S_n$, let T be a Sims table for G over the base $X = 1, 2, \ldots, n$ and let $C = c_1 \ldots c_n$ be a colouring of G. Let $\pi = \sigma_{1i_1}\sigma_{2i_2}\ldots\sigma_{qi_q}$ be a permutation such that each σ_{ri_r} is a valid entry of T. If $c_{\pi(j)} = c_j$ for all $j < q$, and $c_{\pi(q)} > c_q$, then $C_{\pi\alpha} > C$ for all $\alpha \in G_{q+1}$.*

Proof. Suppose $c_{\pi(q)} > c_q$ and consider $\alpha \in G_{q+1}$ such that $C_{\pi\alpha} \leq C$. Since $\alpha(j) = j$ for all $j \leq q$, $\pi\alpha(j) = \pi(j)$, so

$$c_{\pi\alpha(q)} = c_{\pi(q)} > c_q.$$

Therefore, $C_{\pi\alpha} \neq C$ and there must exist some $p < q$ such that $c_{\pi\alpha(p)} = c_{\pi(p)} < c_p$, which is a contradiction. □

The prefix condition of Lemma 4.3 allows a colouring to be verified as lexicographically minimum without necessarily having to test it against all permutations in the group G. Algorithm 6 gives pseudocode which tests whether a given colouring is lexicographically minimum. Algorithm 6 can be used as the basis for a simple generation algorithm which generates all k^n possible colourings and outputs those which pass the test. The function

TestMinimum in Algorithm 6 takes a Sims table for G and the colouring C to be tested as its arguments. The algorithm recursively generates prefixes from the Sims table, testing each generated prefix against the colouring C. At recursion depth i, a prefix π_i of length i is generated by multiplying the prefix π_{i-1} from the previous level by an element of row i of the Sims table. The first i elements of the permuted colouring C_{π_i} are then compared to the original colouring C. If C_{π_i} is greater, recursion unwinds, since by the prefix condition C must be less than all permutations prefixed by π_i. If C_{π_i} is less, the algorithm terminates and returns `false` since C is not minimum. If the two colourings are equal, recursion proceeds to depth $i + 1$.

One advantage of the Sims table representation is that at step i, it is only necessary to compare the element c_i to $c_{\pi_i(i)}$ rather than comparing every previous element of the two colourings, since permutations in row i of the Sims table fix all elements $l < i$. Consequently, to test the prefix

$$\pi_{i-1} \circ T[i][j],$$

it is not necessary to actually compute the product of the two permutations (which requires $O(n)$ time) since the image of i (that is, $\pi_{i-1}(j)$) is all that is needed. The product computation can therefore be deferred to the case where recursion must continue.

Algorithm 6 Test whether a colouring is lexicographically minimum

```
1: procedure TestMinimumRecursive(n, T, C, π, i)
2:     if i = n + 1 then
3:         return true
4:     end if
5:     for j ← i, i + 1, ..., n do
6:         if T[i][j] = × then
7:             Continue
8:         end if
9:         {Get the image of i under π ∘ T[i][j]}
10:        q ← π(j)
11:        if C[q] < C[i] then
12:            return false
13:        else if C[q] > C[i] then
14:            return true
15:        else
16:            α ← π ∘ T[i][j]
17:            if TestMinimumRecursive(n, T, C, α, i + 1) = false then
18:                return false
19:            end if
20:        end if
21:    end for
22:    return true
23: end procedure
24: procedure TestMinimum(n, T, C)
25:    π ← Identity permutation e
26:    TestMinimumRecursive(n, T, C, π, 1)
27: end procedure
```

As mentioned above, Algorithm 6 can be used to generate all minimum colourings by brute-force. The prefix condition provides a convenient way to avoid testing some permutations against the colouring. The brute-force approach tests all k^n possible colourings, and by Lemma 4.2, there may be $\Theta(k^n)$ minimum colourings of a non-trivial group G. However, the lower bound of Lemma 4.2,

$$\binom{n+k-1}{n} \in \Theta(n^k)$$

is polynomial in n. For $k = 2$, there are $n + 1$ minimum colourings of S_n, so the vast majority of colourings tested by the brute-force approach fail the minimality test. Like the permutations in G, the minimum colourings of G have a structure which allows large portions of the search space to be omitted. A complementary prefix condition for colourings can be used to avoid testing certain colourings, which can improve on the performance of the brute-force approach.

Lemma 4.4. *Let $C_r = c_1, \ldots, c_r$ be a partial assignment of colours to elements of a group $G \leq S_n$. There is a lexicographically minimum colouring prefixed by C_i only if there does not exist a permutation $\pi \in G$ such that $\pi(r) < r$, $c_{\pi(r)} > c_r$ and for all $j < \pi(r)$,*

$$\pi(j) < r, \text{ and}$$

$$c_{\pi(j)} = c_j.$$

Corollary 4.5. *To test whether C_r prefixes a lexicographically minimum colouring, it is sufficient to consider permutations of the form*

$$\pi = \sigma_{1i_1} \ldots \sigma_{ri_r}$$

which are products of entries from the first r rows of a Sims table for G.

□

A generation algorithm can be built around Lemma 4.4 in a similar vein to Algorithm 6, generating colourings one position at a time, only assigning colours which do not violate the conditions of the lemma. Algorithm 7 gives a conceptual outline, omitting the generation of prefixes π, which can be done with an algorithm similar to Algorithm 2. The effect of the prefix-generation loop in Algorithm 7 is similar to that of Algorithm 6 with the maximum

depth restricted to level i.

Algorithm 7 Simple Sims table generation algorithm

```
1: procedure GenerateColouringsRecursive(n, T, k, C, i)
2:     if i = n + 1 then
3:         Output the colouring C
4:         return
5:     end if
6:     for c ← 0, ..., k − 1 do
7:         C[i] ← c
8:         for each prefix π = σ_{1l_1} ... σ_{il_i} do
9:             if π violates Lemma 4.4 then
10:                Continue the outer loop to the next value of c.
11:            end if
12:        end for
13:        {If the lemma was not violated, recurse to the next entry of C}
14:        GenerateColouringsRecursive(n, T, k, C, i + 1)
15:    end for
16: end procedure
17: procedure GenerateColourings(n, T, k)
18:    C ← Array of size n
19:    for c ← 0, ..., k − 1 do
20:        C[1] ← c
21:        GenerateColouringsRecursive(n, T, k, C, 2)
22:    end for
23: end procedure
```

The approaches of Algorithms 6 and 7 can be combined into a single recursive algorithm which leverages both prefix conditions to prune away both infeasible colourings and infeasible permutations. The result is Algorithm 8. Fundamentally, Algorithm 8 can be seen as an outgrowth of Algorithm 6, since both perform a depth-first traversal of the Sims table. Unlike Algorithm 6, Algorithm 8 does not sweep across each row of the Sims table at each level of recursion and return from recursion when finished at each level. Instead, Algorithm 8 recurses into the Sims table, and when finished at a given level, recurses *upward* to the previous level. Branches in the recursion tree are created when a colour is assigned to an element and end when either the colouring is found to be minimum (having been tested

against all possible prefixes of permutations in G) or a permutation which maps the colouring to a lexicographically smaller colouring is found.

The index i tracks the current row of the Sims table. The active permutation, which stores the prefix of $i-1$ entries from the previous rows of the Sims table, is stored in a vector

$$\texttt{perm} = T[1][\texttt{path}[1]] \circ T[2][\texttt{path}[2]] \circ \ldots \circ T[i-1][\texttt{path}[i-1]]$$

and the vector `path` stores the index of the permutations in each row of the Sims table which comprise `perm`.

The elements of the colouring C are initially set to the first colour (represented by 0). A boolean vector `fixed` tracks which elements of C have been fixed to a colour (since elements may not be fixed in a predictable order). When $\texttt{fixed}[i] = \texttt{true}$, the colour value $C[i]$ has been fixed to a colour by an earlier stage of recursion and cannot be modified by lower levels of recursion.

Before the recursive algorithm begins, the first element c_1 must be fixed to a colour. The recursive algorithm begins with i set to 1, `perm` set to the identity, and `path` set to all zeros. At each step, it is assumed that Lemma 4.4 is not violated for $c_1, c_2, \ldots, c_i$ and that the colour of all elements less than i is fixed. The main operation at each recursive step is as follows:

1. The smallest index $j \geq \texttt{path}[i]$ such that $T[i][j] \neq \times$ is found.

2. The image q of i under $\texttt{perm} \circ T[i][j]$, which is equivalent to $\texttt{perm}[j]$, is found.

3. If $C[q] < C[i]$, Lemma 4.4 is violated for $c_1, \ldots, c_q$, so element q must be fixed to a colour greater than or equal to $C[i]$. Note that the invariant above implies that $q > i$. If position q is already fixed, it is impossible to change the colour of $C[q]$, so the branch of recursion terminates. Otherwise, $\texttt{fixed}[q]$ is set to `true` and new branches of recursion are started for each possible colour assignment to $C[q]$.

4. If $C[q] > C[i]$, then by Lemma 4.3, no permutation prefixed by $\texttt{perm} \circ T[i][j]$ will map C to any lexicographically smaller permutation, so $T[i][j]$ does not need to be considered further and $\texttt{path}[i]$ is set to $j + 1$ and recursion continues.

5. If $C[q] = C[i]$, then new branches of recursion are created for all assignments to $C[i]$ and $C[q]$ such that $C[q] \geq C[i]$.

In the case where no suitable value j can be found in step 1 of the above (that is, when $T[i][j] = \times$ for all possible j), row i of the Sims table has been exhausted, so the algorithm returns to the previous level of the Sims table. To do so, the permutation `perm` is multiplied on the right by $T[i-1][\texttt{path}[i-1]]^{-1}$ to return it to the state it was in before entering level i. Then, $\texttt{path}[i]$ is reset to 0 and $\texttt{path}[i-1]$ is incremented. Finally, the algorithm recurses to level $i - 1$. In the event that $i = -1$, all possible permutations have been tested against the active colouring and it can be output as a minimum colouring.

When $i = n$, the permutation `perm` contains a fully formed entry of G (in which every row of the Sims table is represented[1]), and since recursion continued to the bottom of the table, the permutation must fix the entire colouring (since if `perm` carried C to a lesser or greater colouring, the algorithm would not have descended to row n). This is not significant when lexicographically minimum colourings are desired, but this part of the algorithm is relevant to the study of distinguishing colourings (see Chapter 3).

[1]Row n is not explicitly represented, but it only contains the identity e.

Algorithm 8 Generate all labelings up to isomorphism (Sims table version)

```
1: procedure LABEL(n, T, k, C, fixed, i, perm, path)
2:     if i = −1 then
3:         Output the colouring C
4:         return
5:     end if
6:     if i = n + 1 then
7:         {The permutation currentPerm fixes the colouring C}
8:         {Continue traversing the table}
9:         path[i] ← 1
10:        if i ≥ 2 then
11:            perm ← perm ∘ T[i − 1][path[i − 1]]^{−1}
12:            path[i − 1] ← path[i − 1] + 1
13:        end if
14:        LABEL(n, T, k, C, fixed, i − 1, perm, path)
15:        return
16:    end if
17:    j ← path[i]
18:    while j ≤ n and T[i][j] = × do
19:        j ← j + 1
20:    end while
21:    if j = n + 1 then
22:        path[i] ← 1
23:        if i ≥ 1 then
24:            perm ← perm ∘ T[i − 1, path[i − 1]]^{−1}
25:            path[i − 1] ← path[i − 1] + 1
26:        end if
27:        LABEL(n, T, k, C, fixed, i − 1, perm, path)
28:        return
29:    end if
30:    path[i] ← j
31:    q ← perm[j]
```

```
32:     if C[q] < C[i] then
33:         if not fixed[q] then
34:             fixed[q] ← true
35:             prev_cq ← C[q]
36:             for z ← C[i], C[i] + 1, ..., k − 1 do
37:                 C[q] ← z
38:                 Create copies newPerm and newPath of perm and path
39:                 LABEL(n, T, k, S, C, fixed, i, newPerm, newPath)
40:             end for
41:             C[q] ← prev_cq
42:             fixed[q] ← false
43:         end if
44:         return
45:     else if C[q] > C[i] then
46:         path[i] ← path[i] + 1
47:         LABEL(n, T, k, S, C, fixed, i, perm, path)
48:     else
49:         if fixed[q] then
50:             LABEL(n, T, k, S, C, fixed, i + 1, perm ∘ T[i][j], path)
51:             return
52:         end if
53:         i_fixed ← false
54:         if fixed[i] = false then
55:             fixed[i] ← true
56:             i_fixed ← true
57:         end if
58:         fixed[q] ← true
59:         prev_cq ← C[q]
60:         for z ← C[i], C[i] + 1, ..., k − 1 do
61:             C[q] ← z
62:             Create a copy newPath of path
63:             LABEL(n, T, k, S, C, fixed, i + 1, perm ∘ T[i][j], newPath)
64:         end for
65:         C[q] ← prev_cq
66:         fixed[q] ← false
67:         if i_fixed = true then
68:             fixed[i] ← false
69:         end if
70:     end if
71: end procedure
```

```
72: procedure GenerateColourings(n, T, k)
73:     C ← Array of size n
74:     fixed ← Array of size n, initialized to false.
75:     fixed[1] ← true
76:     for c ← 0, ..., k − 1 do
77:         C[1] ← c
78:         path ← Array of size n, initialized to 0
79:         perm ← Identity permutation on n items
80:         Label(n, T, k, C, fixed, 1, perm, path)
81:     end for
82: end procedure
```

4.4 Optimizations to the Backtracking Algorithm

A transcription of the pseudocode for Algorithm 8 into a computer language such as C reveals several performance bottlenecks which can be expected to affect all variants of the algorithm. The presentation of the pseudocode in Algorithm 8 was designed to give a relatively high-level outline of the algorithm's behavior and avoid overspecialization to allow easy adaptation of the algorithm to other problems, such as generation of independent sets (see Section 4.5.1) or the distinguishing colouring problem discussed in Chapter 3. For specific problems, there may be many avenues for specific optimizations. In general, there are a few simple modifications which can improve practical performance. None of these modifications will provide an asymptotic improvement, or even consistently improve performance across different architectures. Some improvements might be automatically added by a sufficiently advanced optimizing compiler. However, for the purposes of quickly generating colourings at the possible expense of code readability, they may be useful. All of the line numbers referenced in this section refer to the pseudocode of Algorithm 8.

4.4.1 Recursive Call Removal

Algorithm 8 contains two intertwined patterns of recursion. When a colour entry is fixed, recursion branches up to k ways for the possible colour values, with each branch being given a private copy of the permutation and path vectors. All other recursive calls are non-branching and immediately followed by a return statement, which is a pattern called *tail recursion*. Tail recursion can be optimized automatically into non-recursive control flow logic by most optimizing compilers[2]. To avoid the overhead of function calls, as well as potential stack-overflow resulting from the high recursion depth, the entire algorithm can be implemented iteratively, using a stack to implement the branching recursive calls and manually optimizing away the tail recursive calls.

4.4.2 Ignoring Invalid Row Entries

When choosing a permutation at depth i to append to the active prefix, Algorithm 8 scans through row i of the Sims table to find the first valid permutation $T[i][j]$ with $j \geq \texttt{path}[i]$. Lines 17 to 20 of the pseudocode perform this scan with a simple loop over the row entries. Since this scan may be performed many times over the course of the algorithm, and since some rows may be sparsely populated, the loop may perform a large amount of unnecessary work. This can be remedied by pre-computing a table `nextIdx` of pointers, where $\texttt{nextIdx}[i][q]$ gives the index of the first valid $T[i][j]$ with $j \geq q$. Such a table can be computed in $O(n^2)$ time and allows the loop in lines 17 to 20 to be replaced with the constant-time lookup

$$j \leftarrow \texttt{nextIdx}[i][\texttt{path}[i]]$$

[2]Tail recursion optimization is a well-established area of programming language optimization, due to the large amount of tail recursion in functional languages such as LISP. Among others, the ubiquitous `gcc` compiler supports tail recursion optimization (see [15]).

4.4.3 Ignoring Trivial Rows

A *trivial row* of the Sims table is a row i in which the only valid permutation is σ_{ii}. When LABEL is called with an argument i which is a trivial row, the algorithm will immediately recurse downward (via lines 48 through 70). Depending on whether $C[i]$ is already fixed, the downward recursion may fix $C[i]$ to all possible values and branch. When a lower level recurses back up to level i, the algorithm immediately recurses upward (via lines 21 through 29).

Similar to the invalid row entries in Section 4.4.2, trivial rows can be discovered in advance, and a pointer-based approach can be used to skip over them. This can be implemented with a vector `nextRow`, in which the value `nextRow`$[i]$ is the index of the next non-trivial row after row i (or $n+1$ if i is the last non-trivial row) and a complementary vector `previousRow` in which `previousRow`$[i]$ gives the index of the previous non-trivial row (or -1 if row i is the first non-trivial row). To incorporate this optimization into the pseudocode in Algorithm 8, all calls to LABEL which recurse to level $i+1$ must be changed to recurse to level `nextRow`$[i]$, and all calls which recurse to level $i-1$ must instead recurse to level `previousRow`$[i]$.

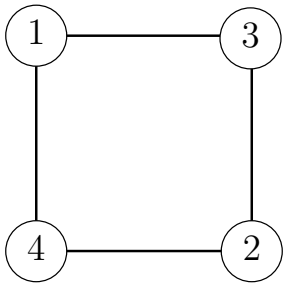
When the unoptimized algorithm recurses downward to a trivial row i, it may fix $C[i]$ to all possible colours before continuing recursion (if $C[i]$ is not fixed already). During processing of a trivial row, the algorithm also tests $C[i]$ against the active permutation, and may terminate the branch of recursion. To ignore trivial rows, it is necessary to duplicate this behavior when recursing downward across trivial rows. This can be accomplished with the addition of the following extra logic after line 5:

for $r =$ `previousRow`$[i]+1, \ldots, i-1$ **do**
  $q \leftarrow$ `perm`$[i]$
  if $C[q] > C[r]$ **then**
    `path`$[i] \leftarrow 1$
    `perm` $\leftarrow$ `perm` $\circ\, T[$`previousRow`$[i],$ `path`$[$`previousRow`$[i]]]^{-1}$
    `path`$[$`previousRow`$[i]] \leftarrow$ `path`$[$`previousRow`$[i]] + 1$
    LABEL$(n, T, k, C,$ `fixed`$,$ `previousRow`$[i],$ `perm`$,$ `path`$)$

```
    else if not fixed[q] then
        fixed[q] ← true
        prev_cq ← C[q]
        for z ← C[r], C[r] + 1, ..., k − 1 do
            C[q] ← z
            Create copies newPerm and newPath of perm and path
            LABEL(n, T, k, S, C, fixed, i, newPerm, newPath)
        end for
        C[q] ← prev_cq
        fixed[q] ← false
        return
    else if C[q] < C[r] then
        return
    end if
end for
```

Since this optimization essentially creates separate copies of the functionality of the algorithm for trivial and non-trivial rows, it is entirely a pragmatic attempt to improve practical performance and does not reduce the depth or breadth of recursion.

Another way to resolve the issue of trivial rows is to renumber elements of the input group G to ensure that all non-trivial rows are adjacent at the top of the table. Figure 4.1 shows the Sims table for the automorphism group of a 4-cycle with a particular vertex numbering. Rows 2 and 4 are trivial.



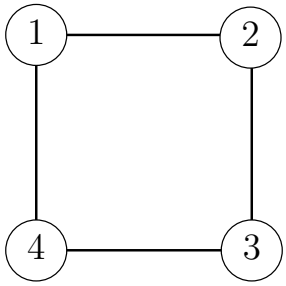
(a) Graph

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} & \sigma_{14} \\ \times & \sigma_{22} & \times & \times \\ \times & \times & \sigma_{33} & \sigma_{34} \\ \times & \times & \times & \sigma_{44} \end{pmatrix}$$

(b) Sims Table

Figure 4.1: Sims table and permutations for C_4

Figure 4.2 shows a renumbered 4-cycle with the Sims table for its automorphism group. After renumbering, rows 3 and 4 are trivial.



(a) Graph

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} & \sigma_{14} \\ \times & \sigma_{22} & \times & \sigma_{24} \\ \times & \times & \sigma_{33} & \times \\ \times & \times & \times & \sigma_{44} \end{pmatrix}$$

(b) Sims Table

Figure 4.2: Sims table and permutations for C_4 after renumbering

Once trivial rows are moved to the bottom of the table, Algorithm 8 can be divided into two phases, one for the upper, non-trivial rows (which use the full pseudocode given in the algorithm) and a second phase for the lower, trivial rows, which are processed by the condensed pseudocode given earlier in this section.

4.5 Specializations of the Backtracking Algorithm

Many specific graph problems take the form of labellings of the vertices of a graph with k colours. Algorithm 8 can be used to generate all possible instances of a given problems up to isomorphism by simply generating all colourings up to isomorphism and discarding inadmissible labellings. In cases where the number of instances is relatively small compared to the number of lexicographically minimum labellings, this approach is computationally wasteful. For some problems, Algorithm 8 can be modified to terminate recursion early when a partial labelling becomes inadmissible. This section presents three such modifications. The first modification generates all independent sets (of any size), the second generates all distinguishing colourings, and the third generates all proper colourings.

4.5.1 Independent Sets

The algorithm given by Myrvold and Fowler in [27], which was summarized in Section 4.2, was originally designed to enumerate the independent sets of large graphs. The output took the form of 2-colourings of the input graph, where the vertices with colour 1 formed an independent set. Algorithm 8 can be adapted to produce the same output by adding logic to set an element $C[i]$ to colour 1 only if no neighbour of vertex i has already been fixed to colour 1.

This can be accomplished by creating an auxiliary subroutine to test whether $C[i]$ can be set to a colour z:

```
procedure CanSet(C, fixed, i, z)
    for Each neighbour j of i do
        if fixed[j] and C[j] = z then
            return false
        end if
    end for
    return true
end procedure
```

and changing the loops which set colours (lines 36 to 40 and 60 to 64) to

```
if C[i] = 0 then
    C[q] ← 0
    Create a copy newPath of path
    Label(n, T, k, S, C, fixed, i + 1, perm ∘ T[i][j], newPath)
end if
if CanSet(C, fixed, q, 1) then
    C[q] ← 1
    Create a copy newPath of path
    Label(n, T, k, S, C, fixed, i + 1, perm ∘ T[i][j], newPath)
end if
```

As the performance benchmarks in Chapter 5 confirm, the algorithm in [27] is significantly faster for graphs with small groups. However, when the group is relatively large for the number of vertices, the adapted form of Algorithm 8 has a performance advantage.

Additionally, the algorithm in [27] can only process groups which are explicitly stored in memory, which limits the maximum group size. Algorithm 8 has no such restriction.

4.5.2 Distinguishing Colourings

To generate only distinguishing colourings, Algorithm 8 can be modified to terminate recursion whenever a non-trivial permutation fixing the current labelling is found. To do this, the algorithm returns when recursion reaches a point where i equals n instead of continuing recursion (which amounts to deleting lines 7 through 14).

When execution reaches the point where i equals n, every element of the colouring will be fixed, and the permutation `perm` will be fully formed (containing one generator, which may be the identity, from each level of the Sims table). It may be possible to identify whether a colouring is non-distinguishing sooner (for example, if suitable conditions on what partial colourings can prefix a distinguishing colouring are found), but such an optimization was not studied for this application.

4.5.3 Proper Colourings

A *proper k-colouring* of a graph G on n vertices is a k-colouring $C = c_1c_2 \dots c_n$ such that if vertices i and j are neighbours, then $c_i \neq c_j$. To generate all proper k-colourings with Algorithm 8, an approach similar to that used to generate independent sets in Section 4.5.1 can be used, except with restrictions placed on the use of all colours. When the colour of a vertex is fixed, only colours which are not already used by the neighbours of the vertex are considered.

Using the CanSet function given in Section 4.5.1, it is only necessary to change the loops which set colours (lines 36 to 40 and 60 to 64) to

for $z \leftarrow C[i], C[i]+1, \dots, k-1$ **do**

```
    if CanSet(C, fixed, q, z) then
        C[q] ← z
        Create copies newPerm and newPath of perm and path
        Label(n, T, k, S, C, fixed, i, newPerm, newPath)
    end if
end for
```

The definition of 'minimum colouring' given in Section 1.1.4 specifies that colour classes are distinct, so a k-colouring $C = c_1 c_2 \dots c_n$ is considered to be different than any colouring

$$C' = \kappa(c_1)\kappa(c_2)\dots\kappa(c_n)$$

obtained by applying some permutation $\kappa \in S_k$ to the colour classes. As a result, a proper colouring on k colours will be generated $k!$ times using the modification above. Additionally, the definition of colouring used in Section 1.1.4 permits colour classes to be empty (so a colouring such as '0011' is a valid 3-colouring). If these two properties are not desirable, the output phase of the algorithm (line 3) can be modified to only output colourings in which the first appearance of a colour $z+1$ must come after the first appearance of colour z, and all k colours are present.

Chapter 5

Computational Results

This chapter summarizes the results of computational benchmarks and data collection for different types of labellings up to isomorphism using the backtracking algorithm in Chapter 4 (Algorithm 8) and the algorithm given by Myrvold and Fowler in [27] (hereafter abbreviated to the 'MF algorithm'). Section 5.1 describes the input data used for these experiments. Sections 5.2 and 5.3 summarize the performance benchmarks, Section 5.4 summarizes the collected data, and Section 5.5 uses the generated data to compare the accuracy of the various distinguishing number bounds in Chapter 3.

5.1 Benchmark Dataset

Let G be a graph whose automorphism group partitions the vertices into r orbits:

$$O_1, O_2, \ldots, O_r.$$

Since no automorphism in $\mathrm{Aut}(G)$ carries an element of O_i to a different orbit O_j, the set of lexicographically minimum k-colourings of G is the Cartesian product of the minimum k-colourings of the orbits. As a result, for the purpose of evaluating the performance of

the generation algorithms, graphs with a single large orbit were used, under the assumption that such graphs would be 'harder' to process due to the larger orbit size. No large-scale testing was performed with multiple-orbit graphs due to time constraints, although there are certain optimizations that only apply to such graphs. It is also important to note that while minimum labellings of a multiple-orbit graph can be constructed from the cartesian product of the labellings of the orbits, it is not, in general, possible to generate other labellings, such as independent sets and proper colourings, this way.

A graph G is *vertex transitive* (or just 'transitive') if, for every pair of vertices $u, v \in V(G)$, there exists some $\pi \in \mathrm{Aut}(G)$ such that $\pi(v) = u$. The automorphism group of a vertex transitive graph has a single orbit containing all vertices. The data used for the benchmarks in this chapter consists of the set of all vertex transitive graphs on up to 20 vertices with automorphism groups of size at most 10!, compiled by Royle[1]. Since the MF algorithm requires a list of all automorphisms to be stored in memory, groups larger than 10! could not be accommodated. The `nauty` program[2] was used to find a generating set for the automorphism group of each graph.

Part of `nauty`'s computation finds a base $X = k_1, \ldots, k_m$ for the automorphism group $\mathrm{Aut}(G)$ of its input graph G. The produced generating set is stratified according to this base in a similar manner to a Sims table. Each generator is associated with a level i and fixes every element k_j with $j < i$. As a result, it is easy to produce a Sims table with all m non-trivial rows at the top. To produce a Sims table for each automorphism group, the vertices were renumbered to give the base vertices the first m indices, and then Algorithm 5 (the Schreier-Sims algorithm) was used to produce the Sims table.

The MF algorithm takes the automorphism group as a list of permutations. To produce such a list for each input group, Algorithm 2 was run on the Sims table for the group. The

[1]Described in [29], available online at `http://school.maths.uwa.edu.au/~gordon/`. The data used in the benchmarks was accessed in May 2012.

[2]Described in [25], available online at `http://cs.anu.edu.au/~bdm/nauty/`. The version used for the benchmarks was 2.4r2, accessed in June 2012.

input data provided to both algorithms used the same vertex numbering.

5.2 Benchmark Setup

Three algorithm implementations were tested for three different labelling types on the test dataset. Two implementations of Algorithm 8 were tested, one with the recursion optimization described in Section 4.4.1 (and no other optimizations) and one with all of the optimizations described in Section 4.4. In the following sections, these two implementations will be referred to as 'Sims' and 'Optimized Sims', respectively. An implementation of the MF algorithm, supplied by the authors of [27], was the third algorithm tested.

Each algorithm was used to generate and count all lexicographically minimum labellings, distinguishing colourings and independent sets of the graphs in the test dataset, and the time taken on each graph was recorded. The test platform had an Intel Core 2 Duo E8400 processor and 4GB of memory (although no test case on any algorithm used more than 2GB). Each algorithm was run 10 times on each input graph, and the average time was computed for comparison against the other algorithms. Timings were collected using the POSIX real-time extension library (historically called `librt`). For each algorithm, timing began just before the first processing step of an input graph, after the graph and its group had been read (so the timing information excludes all I/O time). For the Optimized Sims algorithm, the time spent preparing the tables of valid entry and non-trivial row data was included in the timing data.

5.3 Benchmark Results

The benchmark results broadly implied that Algorithm 8 is significantly faster than the MF algorithm on graphs with large automorphism groups, and generally slower on graphs with smaller automorphism groups, particularly when the number of vertices is large. It is

not possible to meaningfully encapsulate the benchmark results in a simple table or plot, due the wide variety of input data. Instead, the data have been summarized with three different types of plots. Timing plots, including plots of the differences in timing between the algorithms, can be found in Appendix A. Since the goal of the benchmarks in this section is to ascertain which algorithm is best for different types of input, this section contains 2-dimensional plots indicating which algorithm performed best on inputs with a given number of vertices and automorphism group size. These plots were constructed by dividing the input dataset according to the number of vertices (ranging from 1 to 20), then dividing the graphs with each vertex count into 20 bins based on the size of their automorphism group. The boundaries between each bin were evenly spaced from 1! to 10! on a logarithmic scale. The colours assigned to each bin by the plots below indicate which algorithm had the lowest total time on the set of graphs in each bin. Bins with no input graphs are coloured white. Figures 5.1, 5.2, and 5.3 summarize the benchmark results for the generation of all lexicographically minimum 2-colourings, 2-distinguishing colourings and independent sets, respectively.

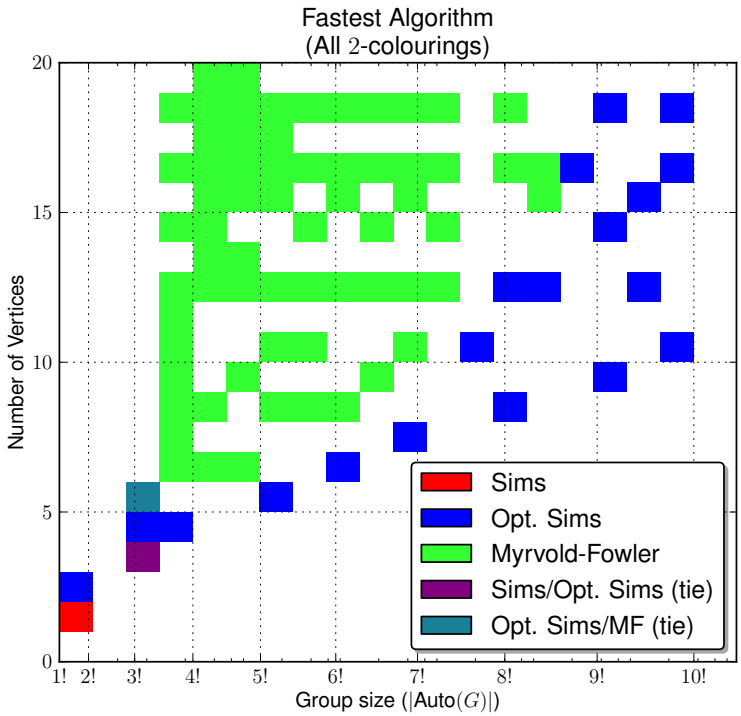


Figure 5.1: Benchmark Results - All lexicographically minimum 2-colourings.

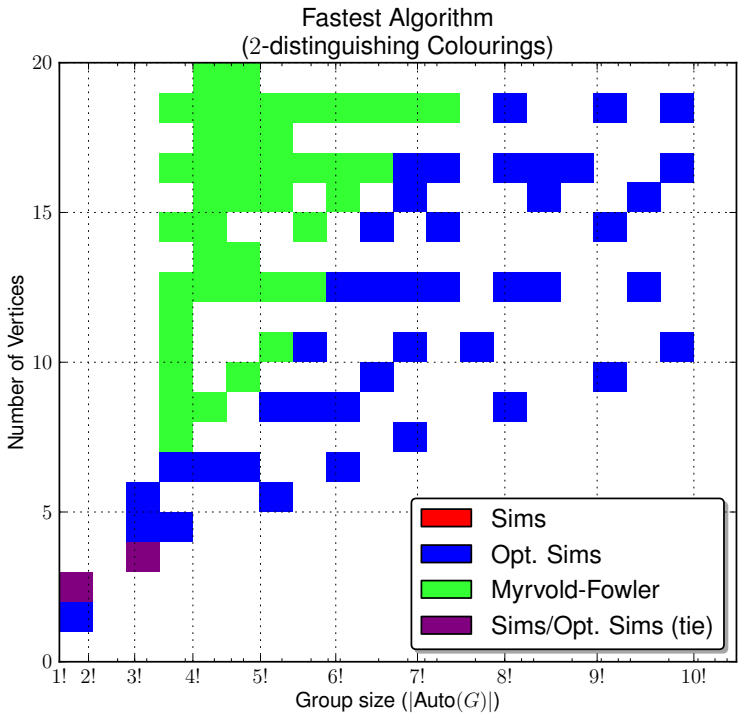


Figure 5.2: Benchmark Results - All lexicographically minimum 2-distinguishing colourings.

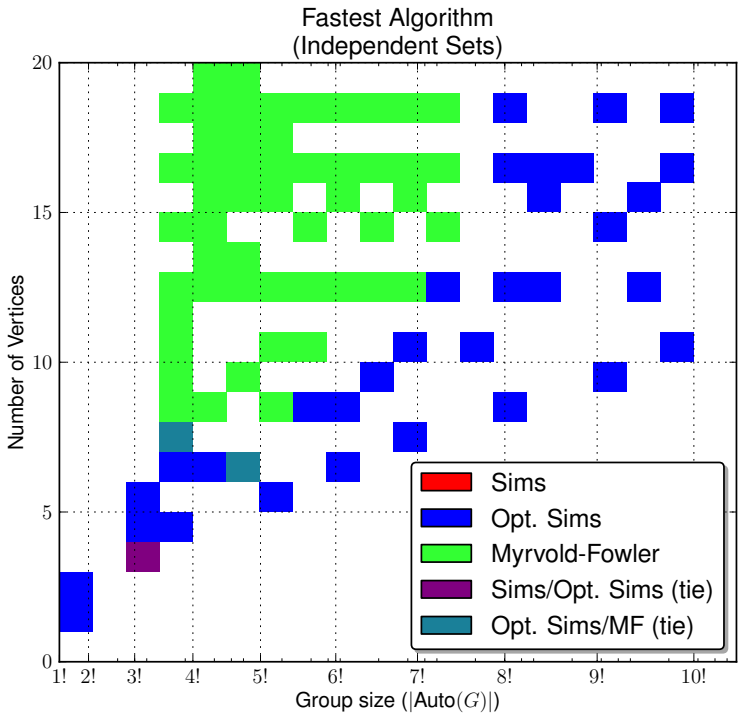


Figure 5.3: Benchmark Results - All lexicographically minimum independent sets.

The benchmark results for all three applications support the hypothesis that the Optimized Sims algorithm tends to perform better on larger groups, while the MF algorithm tends to perform better for small groups. Additionally, the Optimized Sims algorithm tends to be fast when the group size is close to the maximum possible for a given number of vertices. The benchmark results also indicate that the overhead of the optimizations to the Sims algorithm is not generally an impediment to the performance of the algorithm, except on very small graphs where the unoptimized algorithm performs better. All of the ties occur in bins which contain only one or two graphs and where the algorithms are so fast that the timer resolution is insufficient to differentiate between them.

Compared to the generation of all colourings, the Optimized Sims algorithm performs better on graphs with medium-sized groups when generating distinguishing colourings compared to the MF algorithm. This is likely due to its ability to discover non-distinguishing

colourings relatively early in the recursive sequence and backtrack without checking all permutations. The MF algorithm, in contrast, can only identify non-distinguishing colourings at the last level of recursion.

Overall, on the inputs tested, the Optimized Sims algorithm seems to be the best choice on a graph G with n vertices when

$$\frac{n}{\log\left(|\mathrm{Aut}(G)|\right)} < 1$$

and, for all other graphs, the MF algorithm performs better. However, since the MF algorithm can only process groups which can be stored explicitly in memory, it is not a suitable choice for graphs with very large group sizes.

5.4 Generated Data

The generation algorithms were also used to count the 2- and 3-distinguishing colourings of all transitive graphs up to 20 vertices, as well as the set of all transitive graphs on 24 vertices, to provide data which might assist in the construction of a counterexample to the conjecture that the distinguishing number $D(G)$ equals all $D_l(G)$ for all graphs G (see Section 3.4).

Using algorithm 8 and the bounds given in Section 3.2, as well as Theorems 3.2 and 3.13 when applicable, the distinguishing number of all transitive graphs on up to 20 vertices was found (including those graphs with automorphism groups larger than 10!), using the graph catalogue assembled by Royle [29]. Table 5.1 summarizes the distinguishing number data.

Number of Vertices \ Distinguishing Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1																			
2		2																		
3			2																	
4			2	2																
5			1		2															
6		2	2	2		2														
7		2					2													
8		4	4	2	2			2												
9		4	1	2					2											
10		8	8	2		2				2										
11		6									2									
12		48	12	8	2		2					2								
13		12											2							
14		44	6		2			2						2						
15		38	2	2	2	2									2					
16		250	22	4	6				2							2				
17		34															2			
18		342	16	12	4		2			2								2		
19		58																	2	
20		1150	44	6	6	4					2									2
Total	1	2004	122	42	26	10	6	4	4	4	4	2	2	2	2	2	2	2	2	2

Table 5.1: Distinguishing numbers of all transitive graphs on $1 - 20$ vertices

5.5 Evaluation of Bounds on D(G)

Computing the distinguishing numbers of a large collection of graphs allowed the bounds given in Chapter 3 to be systematically evaluated. For some graphs G, the upper and lower bounds on $D(G)$ were equal, allowing the distinguishing number to be determined without any further computation. The only lower bound available for general graphs is Theorem 3.10. Chapter 3 contains several upper bounds on $D(G)$:

Theorem 3.1 [35]: If $|\mathrm{Aut}(G)| \leq k!$, then $D(G) \leq k$.

Theorem 3.3 [1]: $D(G) \leq \lfloor \log_2(|\mathrm{Aut}(G)|) \rfloor + 1$.

Theorem 3.8 [30]: If $\text{CycleNorm}(G) > \log_k(\text{Aut}(G))$, then $D(G) \leq k$.

Theorem 3.9 [6]: If the largest orbit of $\text{Aut}(G)$ has size M and p is the smallest prime dividing $|\text{Aut}(G)|$, then $D(G) \leq \lceil \frac{M}{p-1} \rceil$

Theorem 3.12: If q is the number of equivalence classes of the k^n k-colourings of G under the action of $\text{Aut}(G)$, and

$$q < \frac{2k^n}{|\text{Aut}(G)|}$$

then $D(G) \leq k$.

Counts of the number of graphs on which each upper bound gave the most accurate result are given in Tables 5.2 and 5.3, ordered by vertex count and distinguishing number, respectively. When two or more bounds gave the most accurate result on a given graph, all of them were credited with being the best bound for that graph in the table. For some graphs, the upper bounds were met at equality. Counts of these cases are contained in Tables 5.4 and 5.5, again ordered by vertex count and distinguishing number. Detailed tabulations of the performance of each individual bound against the real distinguishing number can be found in Appendix B.

To compute the bounds in Theorems 3.8 and 3.12 for a graph G, it was necessary to generate and inspect each element of $\text{Aut}(G)$. As a result, it was not possible to compute these bounds for graphs with extremely large automorphism groups. The data summarized by Tables 5.2 and 5.4 only includes the bounds of Theorems 3.8 and 3.12 for graphs G where $|\text{Aut}(G)| \leq 12!$.

Number of Vertices	Best Upper Bound: Thm. 3.1	Thm. 3.3	Thm. 3.8	Thm. 3.9	Thm. 3.12
1	1	1	1	1	0
2	0	2	0	0	0
3	0	2	0	0	0
4	2	2	0	2	0
5	2	1	0	2	1
6	6	4	0	2	2
7	2	0	0	2	2
8	8	2	0	2	6
9	4	0	0	2	5
10	8	0	0	2	14
11	2	0	6	2	6
12	20	0	24	2	54
13	2	0	11	2	12
14	10	0	36	2	46
15	8	0	24	2	40
16	28	0	144	2	258
17	2	0	34	2	34
18	28	0	250	2	354
19	2	0	58	2	58
20	42	0	950	2	1186
Total	177	14	1538	35	2078
Graphs Tested	2245	2245	2213	2245	2213

Table 5.2: Number of graphs for which each upper bound on $D(G)$ was lowest over transitive graphs on $1 - 20$ vertices, ordered by vertex count.

Distinguishing Number (D(G))	Best Upper Bound: Thm. 3.1	Thm. 3.3	Thm. 3.8	Thm. 3.9	Thm. 3.12
1	1	1	1	1	0
2	0	4	1537	0	2002
3	70	9	0	0	62
4	34	0	0	2	8
5	22	0	0	2	6
6	10	0	0	2	0
7	6	0	0	2	0
8	4	0	0	2	0
9	4	0	0	2	0
10	4	0	0	2	0
11	4	0	0	2	0
12	2	0	0	2	0
13	2	0	0	2	0
14	2	0	0	2	0
15	2	0	0	2	0
16	2	0	0	2	0
17	2	0	0	2	0
18	2	0	0	2	0
19	2	0	0	2	0
20	2	0	0	2	0
Total	177	14	1538	35	2078
Graphs Tested	2245	2245	2213	2245	2213

Table 5.3: Number of graphs for which each upper bound on $D(G)$ was lowest over transitive graphs on $1 - 20$ vertices, ordered by distinguishing number.

Number of Vertices	Lower Bound	Upper Bounds				
	Thm. 3.10	Thm. 3.1	Thm. 3.3	Thm. 3.8	Thm. 3.9	Thm. 3.12
1	1	1	1	1	1	0
2	2	2	0	0	2	2
3	2	2	0	0	2	0
4	4	2	2	0	2	0
5	2	2	1	0	2	1
6	6	2	0	0	2	0
7	4	2	0	0	2	2
8	8	2	0	0	2	6
9	6	2	0	0	2	5
10	12	2	0	0	2	10
11	8	2	0	6	2	6
12	56	2	0	24	2	36
13	14	2	0	11	2	12
14	48	2	0	36	2	36
15	40	2	0	24	2	34
16	256	2	0	144	2	206
17	36	2	0	34	2	34
18	344	2	0	250	2	312
19	60	2	0	58	2	58
20	1164	2	0	950	2	1086
Total	2073	39	4	1538	39	1846
Graphs Tested	2245	2245	2245	2213	2245	2213

Table 5.4: Number of graphs for which each bound on $D(G)$ was attained at equality over transitive graphs on $1 - 20$ vertices, ordered by vertex count.

Distinguishing Number (D(G))	Lower Bound	Upper Bounds				
	Thm. 3.10	Thm. 3.1	Thm. 3.3	Thm. 3.8	Thm. 3.9	Thm. 3.12
1	1	1	1	1	1	0
2	2004	2	0	1537	2	1840
3	32	2	3	0	2	6
4	4	2	0	0	2	0
5	2	2	0	0	2	0
6	2	2	0	0	2	0
7	2	2	0	0	2	0
8	2	2	0	0	2	0
9	2	2	0	0	2	0
10	2	2	0	0	2	0
11	2	2	0	0	2	0
12	2	2	0	0	2	0
13	2	2	0	0	2	0
14	2	2	0	0	2	0
15	2	2	0	0	2	0
16	2	2	0	0	2	0
17	2	2	0	0	2	0
18	2	2	0	0	2	0
19	2	2	0	0	2	0
20	2	2	0	0	2	0
Total	2073	39	4	1538	39	1846
Graphs Tested	2245	2245	2245	2213	2245	2213

Table 5.5: Number of graphs for which each bound on $D(G)$ was attained at equality over transitive graphs on $1 - 20$ vertices, ordered by distinguishing number.

The data in Table 5.4 suggests that the lower bound of Theorem 3.10 is often attained at equality. However, since the lower bound is at least 2 for any graph with non-trivial automorphism, and since the vast majority of tested graphs were 2-distinguishable, the accuracy of the lower bound may be exaggerated in Table 5.4, as the alternative classification in Table 5.5 shows. Examination of the data for graphs with larger distinguishing numbers (see Table B.1) reveals that the lower bound tends to drift further from the distinguishing number as the latter increases. Since very few graphs in the tested dataset had distinguishing numbers greater than 5, it is not possible to draw any conclusions about whether (and by

how much) the lower bound diverges from the distinguishing number from the data generated here.

The upper bound of Theorem 3.12 was met at equality for a majority of graphs, and easily prevailed as the best upper bound tested. However, the conditions of the theorem can only be applied for a candidate value k when the number of equivalence classes of the k-colourings is known. , which requires the use of Burnside's Lemma. As the discussion in Section 4.1 notes, counting the equivalence classes is a #P-Hard problem, which is at least as difficult as finding the distinguishing number, so the upper bound is not likely to be feasible in practice for graphs with large automorphism groups. However, if a computationally efficient approximation to Burnside's Lemma is found, the upper bound of Theorem 3.12 could be useful to reduce the search space for distinguishing number computations.

For graphs with larger distinguishing numbers, the best upper bounds were the simple factorial bound of Theorem 3.1 and the group-theoretic condition of Theorem 3.9. As with the lower bound, it is not necessarily possible to draw any conclusions about the efficacy of these bounds on graphs with large distinguishing numbers from the data presented here, since not many such graphs were tested, and those that were tended to be the disjoint union of one or more complete graphs (that is, graphs which meet the conditions of Theorem 3.13). The data does show that the other upper bounds, particularly the cycle-structure based bound of Theorem 3.8, diverge severely from the actual distinguishing number when the group size becomes large. The tables in Appendix B show this divergence to some extent for both Theorem 3.8 and Theorem 3.12. For both theorems, the upper bound would occasionally be much larger than the number of vertices. Theorem 3.12 gave values as high as 62 (on the graph K_{10}). Theorem 3.8 often gave an upper bound of $|\mathrm{Aut}(G)| + 1$ on graphs with large automorphism groups. Since the bound given by Theorem 3.8 is based on the cycle norm of the automorphism group (with small cycle norms giving higher upper bounds), it is unsurprising that the bound becomes unreasonably large on some large groups, since as

group size increases, the cycle norm tends to be lower. Theorem 3.8 is better suited to graphs with relatively small or specialized automorphism groups where the cycle norm is higher.

Chapter 6

Conclusions and Open Questions

The main algorithm presented in this thesis, Algorithm 8, was shown to have high performance on large groups, for which no algorithm previously existed. Several applications for the algorithm were described and the algorithm was used to generate a catalogue of data on distinguishing colourings which may be useful for future research.

On the labelling generation front, the main question for further research is whether an asymptotically faster algorithm exists to generate any of the types of labellings studied here (colourings, distinguishing colourings, independent sets or proper colourings) up to isomorphism given the automorphism group of the input graph. Some research has already been done on counting the number of labellings of various types, and another open question is whether the number of certain types of labellings can be counted efficiently. As Section 4.1 points out, counting the number of equivalence classes of all colourings is #P-Hard, making an efficient counting algorithm unlikely in general (since #P contains the polynomial hierarchy [2]). The complexity of counting distinguishing colourings is unknown (although it is obviously at least as difficult as computing the distinguishing number).

The distinguishing colouring variant of Algorithm 8 can be used to test whether $D(G) \leq k$ for a given k by attempting to generate a single k-distinguishing colouring (and terminating

as soon as one is found). Since Algorithm 8 is designed for exhaustive generation, it is not optimized for this application. Therefore, a natural avenue for future research is whether it can be optimized further or whether a new backtracking algorithm to compute the distinguishing number faster can be formulated.

Besides Algorithm 8, the main theoretical contributions of this thesis are the new bounds on distinguishing numbers and new complexity results proven in Chapter 3. The original motivation for this research was the study of list distinguishing colourings, particularly the open question of whether $D_l(G) = D(G)$ for all graphs G. Although there were no breakthroughs on that front, the distinguishing colouring data collected during this research may be helpful in guiding the search for a counterexample (if one exists), and the complexity results in Chapter 3 may be useful in establishing the computational complexity of determining the list-distinguishing number $D_l(G)$. Since list-distinguishability is a relatively new and unexplored topic, a major open question is what bounds exist on $D_l(G)$. One starting point to find bounds on $D_l(G)$ would be to study whether the bounds on $D(G)$ given in Chapter 3 can be generalized. If bounds on $D_l(G)$ are found which seem to differ from the bounds on $D(G)$ on certain graphs, those graphs may be candidates for a counterexample to the conjecture $D(G) = D_l(G)$. As far as the conjecture itself is concerned, an immediate question is whether anything can be proven about the structure of a counterexample, such as a lower bound on its size, the size of its automorphism group, or its distinguishing number. For example, if $|\mathrm{Aut}(G)| = 2$, it can be shown that $D(G) = D_l(G) = 2$ (since any list of size 2 will allow each vertex to take a different colour than its image under the single automorphism in the group), so a counterexample must have a group size of at least 3.

There are published results which classify the distinguishing number of certain families of graphs. Besides the distinguishing number $D(G)$, another interesting property of a graph is the number of distinct distinguishing colourings with $D(G)$ colours. A graph G with $D(G) = k$ that has only one k-distinguishing colouring (up to isomorphism) can be considered

'minimally k-distinguishing'. An open question is whether graphs with this property can be classified, and also whether the number of distinguishing colourings has any impact on list-distinguishability.

Theorem 3.15 proves that distinguishing colouring is fundamentally equivalent to the graph automorphism problem. In the event that automorphism and isomorphism are inequivalent, a natural question is whether a generalization of distinguishability can be formulated which is equivalent to isomorphism, or if an existing variant of distinguishability, such as list-distinguishability, already is equivalent to isomorphism.

The new upper bound on the distinguishing number presented in Theorem 3.12 is proven with a counting argument using the number of equivalence classes of the k-colourings of the graph under the action of its automorphism group. Extending this counting argument to find the size of the individual equivalence classes would give a new method for computing the distinguishing number. Although such an approach would still require some variant of Burnside's Lemma (or more generally, Pólya Theory [36]), it would provide a practical alternative to the methods used here when the group size is significantly smaller than the total number of k-colourings, since it would only require examining each group element, not potentially generating all possible k-colourings.

Appendix A

Benchmark Timing Data

This appendix summarizes the timing data for the benchmarks in section 5.3. For each application of the algorithms, this section contains plots of the actual timings of each algorithm on each input, as well as the difference in timings between pairs of algorithms. Figures A.1, A.2 and A.3 contain the plots for all lexicographically minimum 2-colourings, 2-distinguishing colourings and independent sets, respectively.

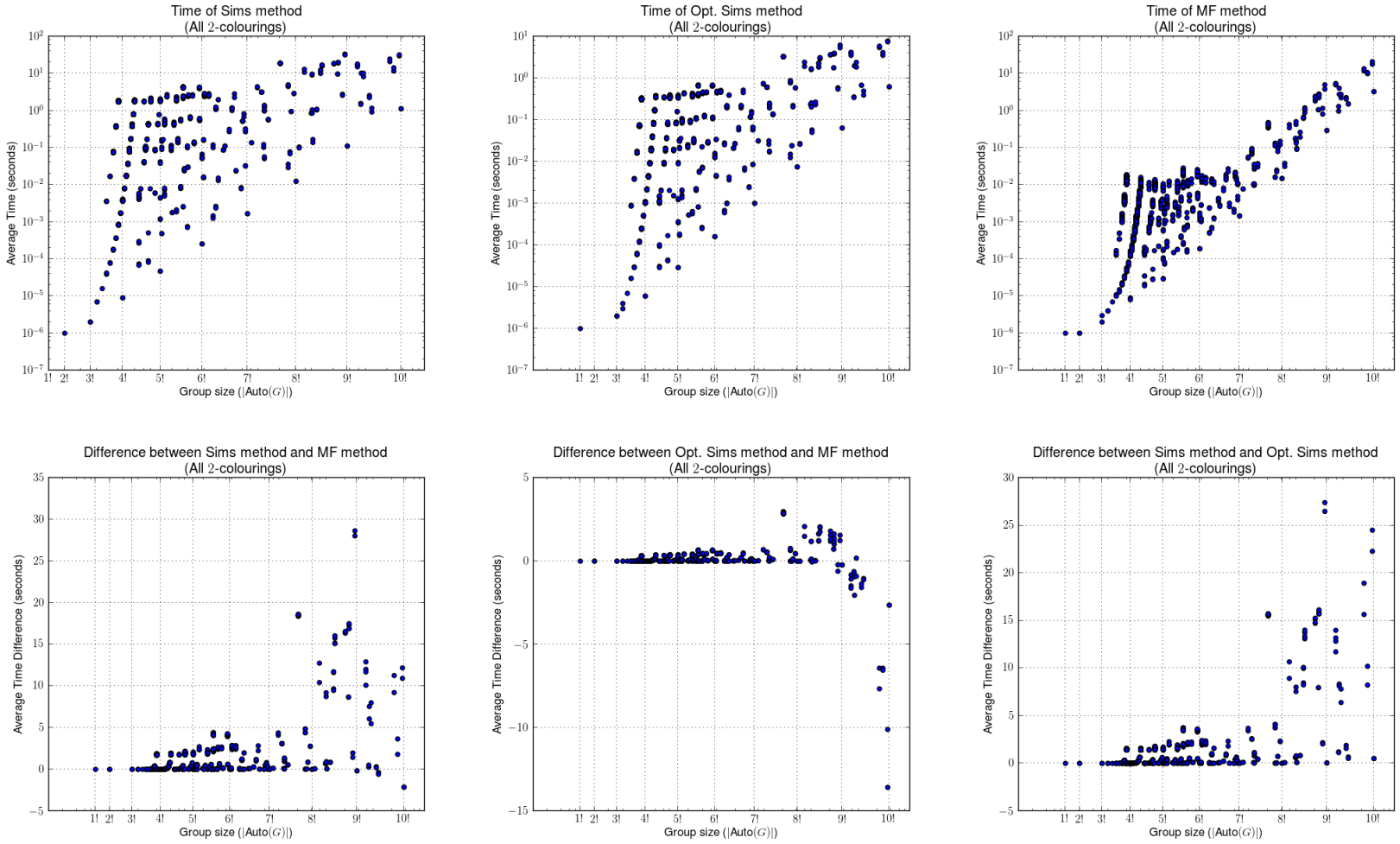


Figure A.1: Timing data for the generation of all 2-colourings.

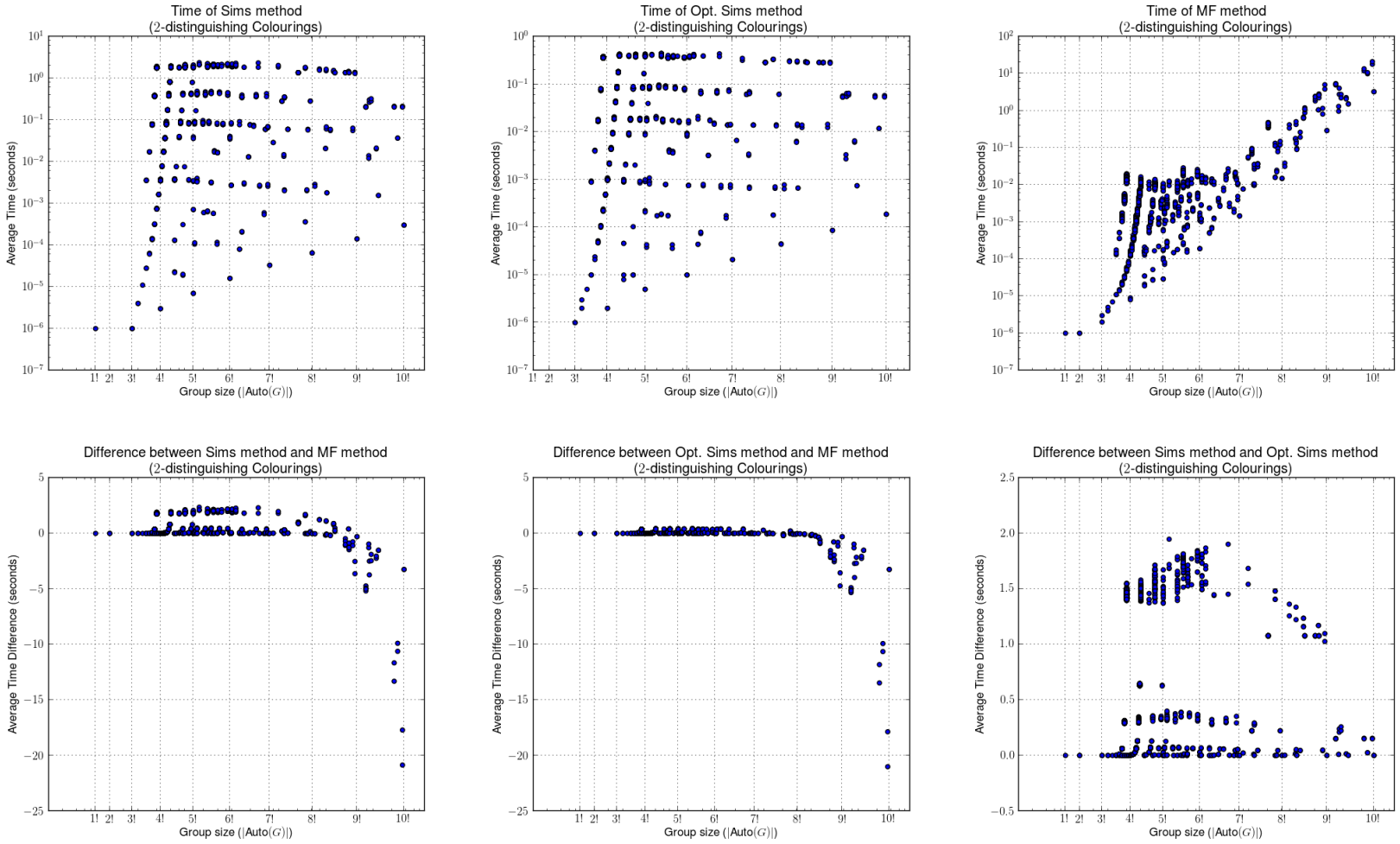


Figure A.2: Timing data for the generation of all 2-distinguishing colourings.

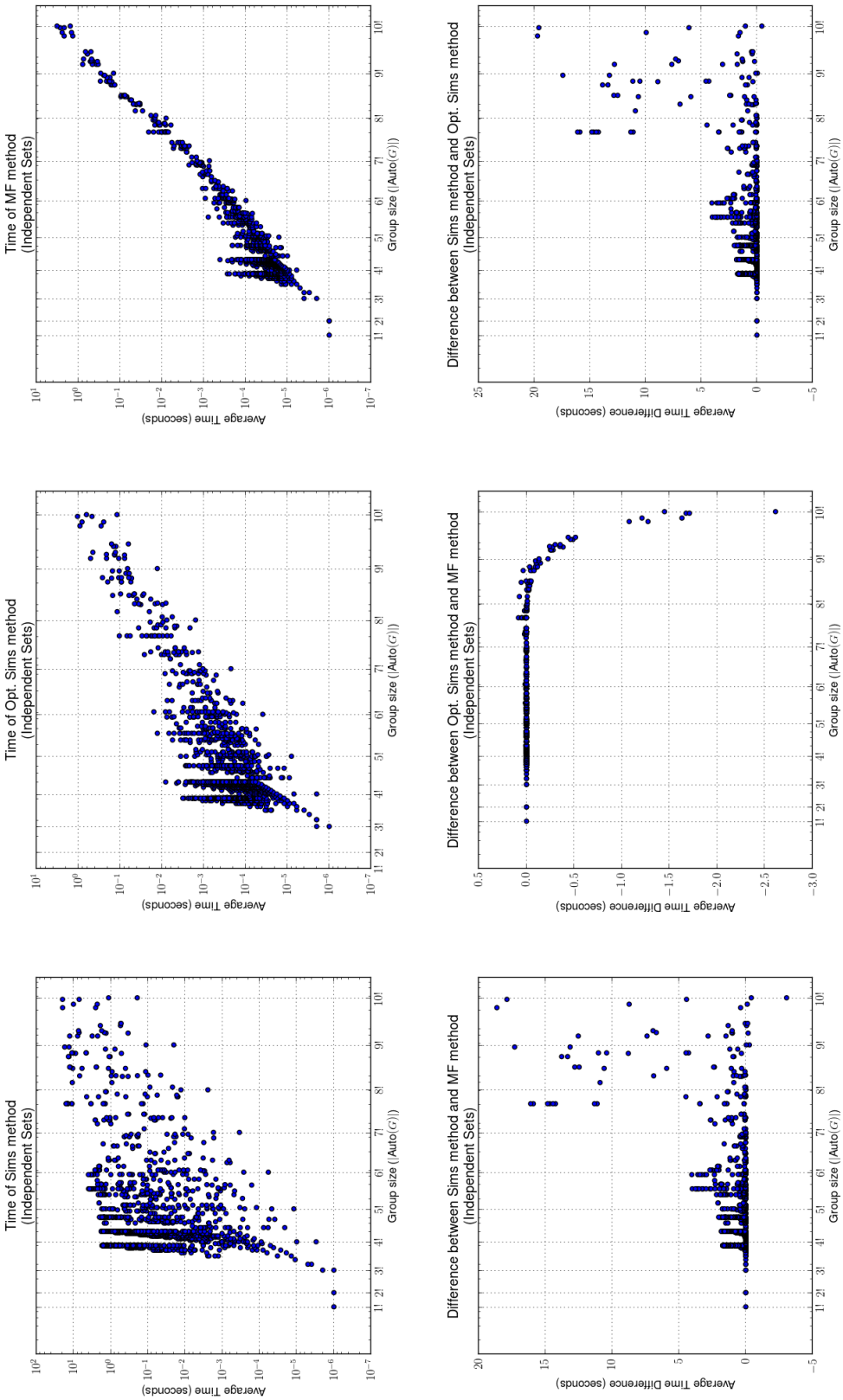


Figure A.3: Timing data for the generation of all independent sets.

Appendix B

Bound Accuracy Data

This appendix contains tables showing the performance of each of the bounds tested in Section 5.5 on the set of all transitive graphs on 20 or fewer vertices. As in Section 5.4, the bounds given by Theorems 3.8 and 3.12 were only computed for graphs G where $|\mathrm{Aut}(G)| \leq 12!$ due to computation constraints.

Distinguishing Number (D(G)) \ Lower Bound on D(G) (Theorem 3.10)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1	1																			
2		2004																		
3		90	32																	
4			38	4																
5			2	22	2															
6				6	2	2														
7					4		2													
8						2		2												
9						2			2											
10							2			2										
11								2			2									
12												2								
13													2							
14														2						
15															2					
16																2				
17																	2			
18																		2		
19																			2	
20																				2

Table B.1: Comparison of the lower bound given by Theorem 3.10 with the actual distinguishing number $D(G)$.

Distinguishing Number (D(G)) \ Upper Bound on D(G) (Theorem 3.1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1	1																			
2		2		806	945	201	48	2												
3			2	3	7	12	26	46	26											
4				2	2	2	6	4	8	12	6									
5					2		2		2	6	6	4	4							
6						2		2			2		4							
7							2			2			2							
8								2				2								
9									2				2							
10										2					2					
11											2						2			
12												2								
13													2							
14														2						
15															2					
16																2				
17																	2			
18																		2		
19																			2	
20																				2

Table B.2: Comparison of the upper bound given by Theorem 3.1 with the actual distinguishing number $D(G)$.

Distinguishing Number (D(G)) \ Upper Bound on D(G) (Theorem 3.3)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1	1																			
2	2		778		771	243		118	66		22	4								
3		2	3		4	5		10	4		8	16	36		10	12		12		
4					2	2		2			2	4	2		4	4		2	8	10
5						2					2					2			4	16
6									2						2					6
7												2							2	2
8															2					2
9																		2		2
10																				4
11																				4
12																				2
13																				2
14																				2
15																				2
16																				2
17																				2
18																				2
19																				2
20																				2

Table B.3: Comparison of the upper bound given by Theorem 3.3 with the actual distinguishing number $D(G)$.

Distinguishing Number (D(G)) \ Upper Bound on D(G) (Theorem 3.8)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1	1																			
2		1537	249	22	28	44	16	12	10	2		16	2				10	24		32
3				1	3		4		2						2	2	4			104
4																				42
5																				22
6																				6
7																				4
8																				4
9																				2
10																				2
11																				2
12																				2
13																				
14																				
15																				
16																				
17																				
18																				
19																				
20																				

Table B.4: Comparison of the upper bound given by Theorem 3.8 with the actual distinguishing number $D(G)$.

Distinguishing Number (D(G)) \ Upper Bound on D(G) (Theorem 3.9)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1	1																			
2		2				2	2	4	4	8	6	48	12	44	38	250	34	342	58	1150
3			2	2	1	2		4	1	8		12		6	2	22		16		44
4				2		2		2	2	2		8			2	4		12		6
5					2			2				2		2	2	6		4		6
6						2				2					2					4
7							2					2						2		
8								2						2						
9									2							2				
10										2								2		
11											2									2
12												2								
13													2							
14														2						
15															2					
16																2				
17																	2			
18																		2		
19																			2	
20																				2

Table B.5: Comparison of the upper bound given by Theorem 3.9 with the actual distinguishing number $D(G)$.

Distinguishing Number (D(G)) \ Upper Bound on D(G) (Theorem 3.12)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	≥20
1		1																		
2		1840	148	16																
3			6	18	14	14	6		34	6		14		10						
4								8	6	4	8	4	4	2			4		2	
5									2		4			4	4		4		2	2
6																		2		4
7																				4
8																				4
9																				2
10																				2
11																				2
12																				2
13																				
14																				
15																				
16																				
17																				
18																				
19																				
20																				

Table B.6: Comparison of the upper bound given by Theorem 3.12 with the actual distinguishing number $D(G)$.

Bibliography

[1] Michael O. Albertson and Karen L. Collins. Symmetry breaking in graphs. *Electronic Journal of Combinatorics*, 3(R18):1–17, 1996.

[2] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, Cambridge, 2009.

[3] Vikraman Arvind, Christine T Cheng, and Nikhil R Devanur. On computing the distinguishing numbers of planar graphs and beyond: a counting approach. *SIAM Journal on Discrete Mathematics*, 22(4):1297–1324, 2008.

[4] B. Bogstad and L.J. Cowen. The distinguishing number of the hypercube. *Discrete Mathematics*, 283(1):29–35, 2004.

[5] Gregory Butler. *Fundamental Algorithms for Permutation Groups*. Springer-Verlag, Berlin, 2003.

[6] Melody Chan. The maximum distinguishing number of a group. *Electronic Journal of Combinatorics*, 13(R70):1–8, 2006.

[7] Melody Chan. The distinguishing number of the augmented cube and hypercube powers. *Discrete Mathematics*, 308(11):2330 – 2336, 2008.

[8] Christine T. Cheng. On computing the distinguishing numbers of trees and forests. *Electronic Journal of Combinatorics*, 13(R11):1–12, 2006.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[10] M. Ferrara, B. Flesch, and E. Gethner. List-distinguishing colorings of graphs. *Electronic Journal of Combinatorics*, 18(P161):1–17, 2011.

[11] Michael Ferrara, Ellen Gethner, Stephen G. Hartke, Derrick Stolee, and Paul S. Wenger. List distinguishing parameters of trees. *Discrete Applied Mathematics*, 161(6):864 – 869, 2013.

[12] Robert Frucht. Graphs of degree three with a given abstract group. *Canadian Journal of Mathematics*, 1:365 – 378, 1949.

[13] Merrick Furst, John Hopcroft, and Eugene Luks. Polynomial-time algorithms for permutation groups. In *Foundations of Computer Science, 21st Annual Symposium on*, pages 36 –41, Oct. 1980.

[14] Joseph A. Gallian. *Contemporary Abstract Algebra*. Brooks-Cole, Belmont, CA, USA, 2010.

[15] GNU Project. GCC 4.8.0 manual. Retrieved May 7, 2013.

[16] L.A. Goldberg. Automating Pólya theory: The computational complexity of the cycle index polynomial. *Information and Computation*, 105(2):268 – 288, 1993.

[17] Marshall Jr. Hall. *The Theory of Groups*. Macmillan, New York, 1959.

[18] Mark Jerrum. A compact representation for permutation groups. *Journal of Algorithms*, 7(1):60 – 78, 1986.

[19] D.E. Knuth. Efficient representation of perm groups. *Combinatorica*, 11(1):33–43, 1991.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley, Reading, Massachusetts, 2011.

[21] W. Kocay. On writing isomorphism programs. *Computational and Constructive Design Theory*, 368:135–175, 1996.

[22] W. Kocay. A modification of the Schreier-Sims algorithm utilising the transitivity of the stabiliser subgroups. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 31:193–206, 1999.

[23] Anna Lubiw. Some NP-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10(1):11–21, 1981.

[24] Eugene M. Luks. Permutation groups and polynomial-time computation. In *Groups and Computation: Workshop on Groups and Computation, October 7-10, 1991*, volume 11, page 139. American Mathematical Society, 1993.

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

[26] Charles F. Miller III. Decision problems for groups - survey and reflections. In Gilbert Baumslag and Charles F. Miller III, editors, *Algorithms and Classification in Combinatorial Group Theory*, volume 23 of *Mathematical Sciences Research Institute Publications*, pages 1–59. Springer New York, 1992.

[27] Wendy Myrvold and Patrick Fowler. Fast enumeration of all independent sets of a graph up to isomorphism. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 85:173–194, May 2013.

[28] Karen S Potanka. Groups, graphs, and symmetry-breaking. Master's thesis, Virginia Polytechnic Institute and State University, 1998.

[29] Gordon F. Royle and Cheryl E. Praeger. Constructing the vertex-transitive graphs of order 24. *Journal of Symbolic Computation*, 8(4):309 – 326, 1989.

[30] A. Russell and R. Sundaram. A note on the asymptotics and computational complexity of graph distinguishability. *Electronic Journal of Combinatorics*, 5(R23):1–7, 1998.

[31] Otto Schreier. Die untergruppen der freien gruppen. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 5:161–183, 1927.

[32] Ákos Seress. *Permutation Group Algorithms*. Cambridge University Press, Cambridge, 2003.

[33] C. C. Sims. Computation with permutation groups. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 23–28. ACM, 1971.

[34] Charles C. Sims. Computational methods in the study of permutation groups. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon Press, 1970.

[35] Julianna Tymoczko. Distinguishing numbers for graphs and groups. *Electronic Journal of Combinatorics*, 11(R63):1–13, 2004.

[36] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics, 2nd Edition*. Cambridge University Press, Cambridge, 2006.

[37] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall Inc., Upper Saddle River, New Jersey, 1996.