9-9-2024

Minimal Separating Sets in Surfaces

Christopher Nelson Aagaard
Portland State University

Minimal Separating Sets in Surfaces

by

Christopher Nelson Aagaard

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Mathematical Sciences

Dissertation Committee:
J.J. Peter Veerman, Chair
John Caughman
Derek Garton
Jeffrey Ovall

Portland State University
2024

Abstract

Given a connected topolgical space X, we say that $L \subseteq X$ is a *minimal separating set* if removing L from X gives a disconnected surface, but removing any proper subset of L leaves the surface connected. We classify which embeddings of topological graphs are minimal separating in an orientable surface X with genus g, and construct a computer program to compute the number of such embeddings, and the number of topological graphs which admit such an embedding for $g \leq 5$.

This dissertation is dedicated to Tessa and to my parents. I've been so fortunate to receive their love, encouragement, and support.

Acknowledgements

First I would like to thank my advisor Peter Veerman. Peter introduced me to this area of research and a provided a plan for me to get started, and since then he's been an excellent mentor. His questions always help me to separate what I understand from what I only think I understand. He has spent so many hours reviewing and editing work with me, and teaching me how to communicate mathematics clearly. Anything intelligble that I write is likely due to his help. Thank you so much.

Thank you to John Caughman, Derek Garton, and Jeffrey Ovall, the other members of my committee. Not only did you consider my research to be worthwhile, but each of you has given me valuable help and feedback during the course of my dissertation, and also in the courses I took with you.

Thank you to the Portland State University Math Department for the financial support provided to me, first as an Enneking fellow and later as a teaching assistant. This support has made it possible for me to stay in school all of this time, and I've learned a great deal in both roles.

Thank you to everyone at Research Computing at Portland State University, not only for letting me use enough computing resources to make this work possible, but also for your generous assistance whenever I've needed software updates or runtime extensions.

Finally, thank you to my fellow graduate students in the Portland State University

Math Department. You've been very kind and helpful to me during my time here, helping not just as fellow students of math, but as friends and comrades.

Table of Contents

List of Tables

List of Figures

1 Introduction

A *mediatrix* informally is the set of points in some space that are equidistant from two given points (or more generally from two given sets of points). The study of mediatrices has a long history, as lines and conic sections are all examples of mediatrices in the plane with the standard Euclidean metric. The mediatrix of two points is a line (the perpendicular bisector of the segment connecting them), and one definition of the conic sections is as follows:
A parabola is the mediatrix of a line and a point not on that line. An ellipse is the mediatrix of a circle and a point in the interior of the circle. A hyperbola is the mediatrix of a circle and a point lying outside the region bounded by the circle (this follows from the definition of hyperbola in terms of foci).

Mediatrices even appear in international law, as Article 15 of the United Nations Convention on Law of the Sea states that (barring previous agreement) the territorial sea border of two nations lies on *the median line every point of which is equidistant to the nearest points to each country* [20], which is precisely a mediatrix! Although mediatrices themselves are not the focus of this dissertation, they provide the motivation. Rather than a mediatrix, our focus is on a related object which we call a *minimal separating set*:

Definition 1.1. A subset L of a connected topological space X is *minimal separating* if $X \setminus L$ is disconnected, but for any proper subset $L' \subset L$, $X \setminus L'$ is connected.

It [32] it was shown that for Brillouin spaces, a large class of metric spaces which

includes all Riemannian manifolds, mediatrices are minimal separating, and it was then shown in [3] that the mediatrix of any two points in a 2 dimensional Riemannian manifold is homeomorphic to a finite topological graph. Then in [10] these results were extended even larger classes of metric spaces (length spaces without branching geodesics and 2-dimensional Alexandrov spaces respectively).

Much stranger sets than finite topological graphs may be realized as minimal separating sets, such as the Lakes of Wada [35], but we are not interested in these as according to [3] and [10] these cannot be realized as mediatrices in fairly "nice" metric spaces. From now on we will freely use the phrase "minimal separating set" in place of the tediously long "minimal separating set which may be realized as a mediatrix of a 2-dimensional Alexandrov space".

The focus of this dissertation is on classifying and computing the topological graphs and graph embeddings which can be realized as minimal separating sets in a surface of genus g. Since the question of whether a graph can even be embedded in a given surface is non-trivial [13], and the fastest algorithm we know of [23] involves actually constructing the embedding, there is no reason to believe that one can find, or even count, the graphs which can be embedded in a minimal separating set in a surface without using a computer to construct the embeddings. Therefore our approach is to find properties that allow us to classify these graphs and graph embeddings, and then use this classification to write a computer program which will actually compute them.

The dissertation is organized as follows: In Chapter 2 we establish relevant back-

ground on graph theory, topology, and topological graph theory. In the final part of this chapter we also define several sets relevant to the enumeration of minimal separating sets and their computation.

In Chapter 3 we establish relationships between the sets of interest laid out at the end of Chapter 2, allowing us to find and count the elements in some of the larger sets while only directly computing the elements of a smaller set. We also establish a method for associating each embedding of interest to a unique cellular embedding.

In Chapter 4, we introduce the language and properties of *combinatorial maps* to efficiently store cellular embeddings in a computer, and lay out the properties of combinatorial maps associated to a minimal separating embedding. There is a brief discussion of existing techniques in the enumeration of combinatorial maps, but there appear to be significant obstacles to their use in our case, as we will observe. We also define the dual of a combinatorial map, and the properties of the dual of a combinatoral map associated to a minimal separating embedding.

In Chapter 5 we introduce *hypermaps*, a generalization of the combinatorial maps from Chapter 4, and relate each minimal separating embedding to a hypermap through the duals of combinatorial maps discussed in Chapter 4. The replacement of combinatorial maps with hypermaps will be key to writing an efficient algorithm to find all minimal separating embeddings in a surface. As we will observe, this dramatically reduces the size of the space we will need to search for minimal separating embeddings.

In Chapter 6 we use the association of each minimal separating embedding with

a hypermap to construct an algorithm for their computation.

Chapter 7 describes some technical obstacles to actually running the algorithm from Chapter 6 and how they are handled. It also presents the final results from running the algorithm from Chapter 6.

Chapter 8 discusses some possibilities for future work and briefly discusses applications to Hurwitz theory and the computation of matrix integrals in physics.

Appendix A gives a very brief introduction to the representation theory of finite groups. Almost all of the preceding chapters are independent of this appendix, but it establishes a formula due to Frobenius which we modify to compute some bounds for memory allocation needed in Chapter 7. We also use the material from this section to compute the exact number of minimal separating embeddings satisfying some specific properties, which allows us to verify the accuracy of some of our computations and also leads to a nice congruence result about sums of reciprocals of binomial coefficients. Appendix A has a substantial amount of material including original results, but we choose to place it as an appendix, rather than a primary chapter, because the material is so different from the rest of the material in the dissertation, and it is really only used for memory estimates in the computer program from chapter 7 and in verifying some computed results.

Finally Appendix B provides the full code for the computer program to do the actual computations.

One issue we have encountered, and which we wouldd like to warn the reader about is the amount of variation in both terminology and precise definitions used across sources. The bulk of this dissertation will be focussed on graph embeddings

in surfaces. Much of the existing literature focusses on the case of what we will call 'cellular embeddings' (see Definition 2.10) where there are natural bijections between the sets of some of the objects we consider, leading to their names being used interchangeably. For example graph embedding, ribbon graph, topological map, and combinatorial map (all to be defined later) are often used as synonyms. Other choices, such as whether graphs are directed or undirected, whether graphs are labelled or not, and the backgrounds and interests of authors lead to a great deal of variation in language. A reader looking at some cited theorems in their original context will notice that they may appear not to match their statement in this dissertation (for example theorems may refer to isomorphism classes of combinatorial maps in this dissertation, but to combinatorial maps in the cited document), and this is due to our modifying the language to match the particular versions of the definitions used here.

2 Preliminaries

2.a Graph Theory Background

In this section we present a few definitions and results from graph theory that will be useful throughout the dissertation. We will only consider as graphs what might be called *undirected graphs* in other literature.

Definition 2.1. A *graph* G consists of an edge set $E(G)$, a vertex set $V(G)$, and a function i_G which maps each edge to a pair of (not necessarily distinct) verticse. We call the elements of $E(G)$ *edges*, the elements of $V(G)$ *vertices*, and say that an edge e is *incident* to a vertex v if $v \in i_G(e)$. If $i_G(e) = (v, w)$ then we say that e *joins* v and w.

When there is no risk of ambiguity we will simply refer to $E(G)$,$V(G)$, and i_G as E, V, and i respectively. Sometimes we may refer to a graph as a *combinatorial graph* to emphasize the contrast with *topological graphs* (see Definition 2.9 in a later section).

It is tradtional to visualize a graph G with the vertices V as points in space, and each edge e as a line or curve through space with endpoints at $i(e)$. Figure 1 shows a drawing of a graph G in the plane. The dots represent the elements of V, the lines and arcs represent the elements of E, and for a given edge $e \in E$, $i(e)$ consists of the vertices connected by the line for e.

Here we define a few terms about graphs that we will make use of

Definition 2.2. [2] For a graph $G = (E, V, i)$

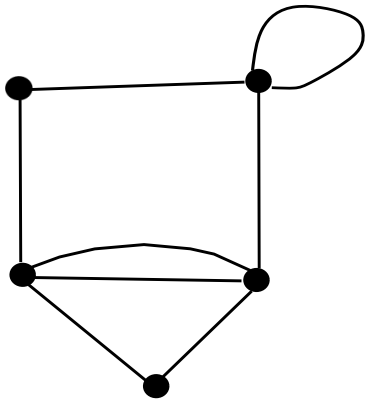

Figure 1: A graph with 4 vertices and 8 edges.

- We say an edge $e \in E$ is a loop if $i(e) = (v, v)$ for some $v \in V$.
- The *degree* of a vertex $v \in V$ is the number of times v appears in $i(E) = \{i(e) : e \in E\}$ (i.e. the number of edges incident to v with loops counted twice).
- For $u, v \in V$, a *path* from u to v is a sequence of vertices and edges:

$$v_0, e_1, v_1, e_2, \ldots, v_{k-1}, e_k, v_k$$

  where $i(e_i) = (v_{i-1}, v_i)$ for each i, $v_0 = u$, $v_k = v$, and $v_i \neq v_j$ whenever $i \neq j$.
- G is *connected* if for all distinct $u, v \in V$ there exists a path from u to v.

In many cases, including the problems we are interested in, the names of the elements in V and E are unimportant, and we might want to treat two graphs as if they were the same when the only difference is "relabeling" the elements of the vertex and edge sets. Here we make this precise with the following definition of isomorphism from [2]:

Definition 2.3. We say that two graphs G and H are isomorphic if there exist bijections $f_1 : E(G) \to E(H)$ and $f_2 : V(G) \to V(H)$ which preserve incidence and non-incidence. The last condition can be formalized as
For all $e \in E(G)$

$$i_H(f_1(e)) = (f_2 \times f_2)(i_G(e))$$

where $(f_2 \times f_2)(u, v) := (f_2(u), f_2(v))$.

At this point a reader familiar with graph theory might be noticing that the definitions of graph and isomorphism given here include more functions than the more familiar definitions. This is because we will need to consider graphs with loops (edges incident to a single vertex) and multiple edges (multiple elements of E with identical image under i). For graphs without loops or multiple edges both the edge sets and isomorphisms can be defined purely in terms of the vertex sets.

The following definition will be particularly important to us in later chapters.

Definition 2.4. An *proper n-vertex-coloring* of a graph G is a partition of the vertices of G into n blocks, such that each edge of G is incident to vertices in two distinct blocks. In the case where $n = 2$, any graph which can be properly 2-vertex-colored is called a *bipartite graph*.

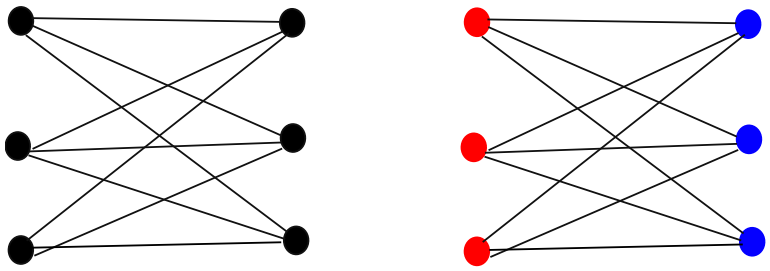

Figure 2: A bipartite graph (left) and a 2-coloring of that graph (right). This graph is called $K_{3,3}$, signifying that it can be properly 2-vertex-colored into two groups of 3 vertices, and each vertex is joined to all vertices of the opposite color class.

A consequence of this definition is that we cannot properly vertex-color any graph with loops. We provide an example of a proper 2-vertex-coloring of a graph in Figure 2.

2.b Topological Background and Notation

In this section we present background definitions and results on topological spaces for future reference. We assume familiarity with basic point-set topology, and results in this section are generally presented without proof, but with citation, for the interested reader.

The following definitions are from [16] (which also has definitions of the terms

used in the definitions).

Definition 2.5.

- An n-dimensional (topological) manifold is a Hausdorff topological space such that every point has a neighborhood homeomorphic to an n-dimensional open disk $D^n(0,r) = \{\mathbf{x} \in \mathbb{R}^n | \|\mathbf{x}\| < r\}$ for some $r > 0$.
- An n-dimensional manifold with boundary is a Hausdorff topological space such that every point either has a neigborhood homeomorphic to an n-dimensional open disk, or to an n-dimensional open half-disk $H^n(0,r) = \{\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n | \|\mathbf{x}\| < r \text{ and } x_n \geq 0\}$.
- A *surface* is a 2-dimensional manifold. A *surface with boundary* is a 2-dimensional manifold with boundary.
- A 2-dimensional manifold X is *orientable* if it does not have a subset homeomorphic to a Möbius band.

For the purposes of reasoning about orientable manifolds, we find the definition from [15] more useful, but it requires introducing a large amount of technical machinery that wouldn't be used anywhere else in this dissertation. Instead we summarize it in plain language in the case of surfaces: A surface is orientable if you can choose direction for clockwise around every point, and those choices are compatible over the whole surface (meaning that if one walks around the surface, what used to be clockwise does not suddenly become counterclockwise). Unless explicitly stated, we assume that an orientation has been chosen for our orientable surfaces, and throughout this dissertation we adopt the convention that counterclockwise is the 'positive direction' for rotation.

We aim to classify minimal separating sets for compact connected surfaces without boundary, so the classification of compact connected surfaces without boundary will naturally be useful.

Theorem 2.1. *[16] Every compact connected orientable surface (without boundary) is homeomorphic to a sphere or the connected sum of n tori. The genus of such a surface is defined to be 0, if it is homeomorphic to a sphere, or n if it is homeomorphic to the connected sum of n tori.*

While classifying minimal separating sets, we will find ourselves looking at connected components of $X \setminus L$ (where X is a surface and L is a minimal separating set). These are non-compact surfaces which get quite a bit more complicated than compact ones. To simplify things, we will compactify these surfaces by adding boundaries, so the related classification of compact connected surfaces with boundary will also be quite useful.

Theorem 2.2. *[16] A compact connected orientable surface with boundary is homeomorphic to a sphere or the connected sum of n tori, with a finite number of open disks removed. The genus of such a surface is defined to be 0, if it is homeomorphic to a sphere, or n if it is homeomorphic to the connected sum of n tori.*

We adopt the notation X_g for the orientable surface of genus g and $X_{g,i}$ for the orientable surface of genus g with i open disks removed. By the above theorems, these objects are unique up to homeomorphism.

Finally, in discussing connectedness of surfaces, we will sometimes want to refer

to *paths* in topological spaces.

Definition 2.6. Given a topological space X with points $x, y \in X$, a *path* from x to y is a continuous map $f : [0, 1] \to X$ with $f(0) = x$ and $f(1) = y$.

Whenever we use the word *path* from here on, context should make it clear whether we are using Defintion 2.2 or Definition 2.6. This next definition and the following theorem are what make paths particularly useful to us.

Definition 2.7. [1] We say a topological space X is *path-connected* if for all $x, y \in X$ there exists a path from x to y. We say X is *locally path-connected* if for each $x \in X$ and each neighborhood U of x, there is a path-connected neighborhood of x contained in U.

We have only been able to find the following result as an exercise, so we provide a brief proof

Theorem 2.3. *A manifold is connected if and only if it is path-connected.*

Proof. Let X be an n-dimensional manifold. Assume X is path-connected and let U, V be open subsets of X such that $X = U \cup V$. Take $u \in U$ and $v \in V$. By assumption there exists a path f from u to v. By continuity $f^{-1}(U)$ is an open subset of $[0, 1]$ and $f^{-1}(V)$ is an open subset of $[0, 1]$. Since $[0, 1]$ cannot be written as a disjoint union of nonempty open subsets, $U \cap V$ is nonempty and X is connected.

Now assume X is connected. Let $x \in X$ and define

$$P := \{y \in X | \text{ there is a path from } x \text{ to } y\}.$$

Since X is an n-manifold, each point $y \in P$ has a neighborhood homeomorphic to $\mathbb{R}^n$, a path-connected space. Therefore each $y \in P$ has a neighborhood U which is path-connected to y, and therefore path-connected to x (by joining the path from x to y with a path from y to a point in U). This shows P is open. Similarly, each point $y \in \bar{P}$ has a neighborhood V homeomorphic to $\mathbb{R}^n$, and if any point $z \in V$ were path connected to x we would have a path from x to y by joining the path from x to z with the path from z to y. Thus $\bar{P}$ is also open. This shows that P is both open an closed, and the only such sets of a connected space are the empty set and the entire space. $x \in P$, so $P = X$ and X is path-connected. $\square$

The following definitions and theorems will be useful in several later proofs:

Definition 2.8. Given a surface X, we call the connected sum of X with a torus the result of *adding a handle to* X.

Suppose X is a surface of genus $g > 0$ and let γ be a non-separating curve in X. Then we call the surface X' obtained by gluing a disk onto each hole in $X \setminus \gamma$ the result of *cutting a handle in* X.

Theorem 2.4. *[1] Let X be an orientable surface with genus g. The surface obtained by adding a handle to X has genus $g + 1$. If $g > 0$, it is possible to cut*

a handle of X, and the resulting surface has genus $g-1$.

Theorem 2.5. *Let X, Y be surfaces of genus g_1, g_2 respectively and let Z be their connected sum. The genus Z is $g_1 + g_2$.*

The proof follows immediately from Theorem 2.1 for surfaces without boundary and Theorem 2.2 for surfaces with boundary.

Notational Conventions For Genera: We will often be simultaneously consider the genera of multiple surfaces. To minimize confusion we will adopt the following conventions:

- X_g will be the closed orientable surface (without boundary) of genus g.
- We use g to represent the the genus of primary interest in contex (see next item for example), typically the genus of the of the surface being minimally separated.
- We use $\hat{g}$ to represent a genus that is varying. For example Theorem 2.7 will state
$$\mathcal{M}_g = \bigcup_{\hat{g}=0}^{g} \mathcal{L}_g$$
- We use g_R to denote the genus of the ribbon graph in context (see Definition 2.13).
- For an indexed set of objects $S_1, \ldots, S_k$ (where each object has a genus) we use g_i to denote the genus of S_i.

2.c Background and Terminology: Topological Graphs and Embeddings

From [33], we know that all minimal separating sets in the surfaces we consider are *finite closed 1-complexes*, which is a synonym for *finite topological graphs*. We will now define topological graphs:

Intuitively, we can imagine a graph as a geometric object, with the vertices as discrete points, and each edge $e \in E$ as an arc joining the vertices in $i(e)$. We can formalize this with the following definition of a *topological graph* (a specification of the general definition of a cell-complex in [15]):

Definition 2.9. A *topological graph* is a 1 dimensional cell-complex. That is, it is defined by a set V of points, a set E of copies of the closed interval $[0, 1]$, and a map $\phi_e : \{0, 1\} \to V$ for each $e \in E$ which maps the boundary of the edge e to 2 (possibly non-distinct) elements of V. This is made into a topological space by taking the disjoint union $V \sqcup E$ and taking the quotient by identifying the boundary points of each $e \in E$ with their image under ϕ_e.

This is a fairly technical definition, but informally: a topological graph is the topological space you get from a graph G, giving each loop the topology of the circle, and for each edge between distinct vertices, we give that edge together with its incident vertices the topology of the closed interval $[0, 1]$. A neighborhood of a vertex v is a union over the edges incident to v of open subsets of the edges, each containing the boundary point (or points) identified with v by the incidence function.

Before going any further, we want to make a note of an important distinction between topological graphs and more familiar combinatorial definition as a set of vertices and edges. Given a finite topological graph Γ with e an edge of Γ, we can consider the graph Γ' obtained by marking a vertex v somewhere in the middle of e, which splits e into two edges, e_1 and e_2. As combinatorial objects, Γ and Γ' wouldn't be considered isomorphic graphs, because Γ' has one more vertex and one more edge than Γ. In the combinatorial language, we would say that Γ' is a *subdivision of* Γ. However, since $e_1 \cup v \cup e_2$ is homeomorphic to e, and the rest of the Γ' is certainly homeomorphic to the rest of Γ, the two are homeomorphic and considered equivalent as topological graphs. As a consequence, every homeomorphism class of topological graph has a representative with no degree two vertices in any connected component which is not homeomorphic to the circle, and with exactly one vertex in each connected component homeomorphic to the circle.

Our interest in topological graphs is due to the following result from [10] (an extension of a similar result from [3]).

Theorem 2.6. *Let X be a compact Alexandrov*[1] *surface (possibly with boundary). For any pair of disjoint nonempty compact subsets $A, B \subseteq X$, the equidistant set of points equidistant from A and B is homeomorphic to a finite closed 1-dimensional cell-complex.*

[1]We do not give a precise definition of what it means for a space to be Alexandrov as it involves technical details and this theorem will be our final reference to them. For details on Alexandrov spaces we refer the reader to [4] and here we just note that Alexandrov spaces form a class of metric spaces which contain and generalize such familiar examples as Euclidean spaces and Riemannian manifolds.

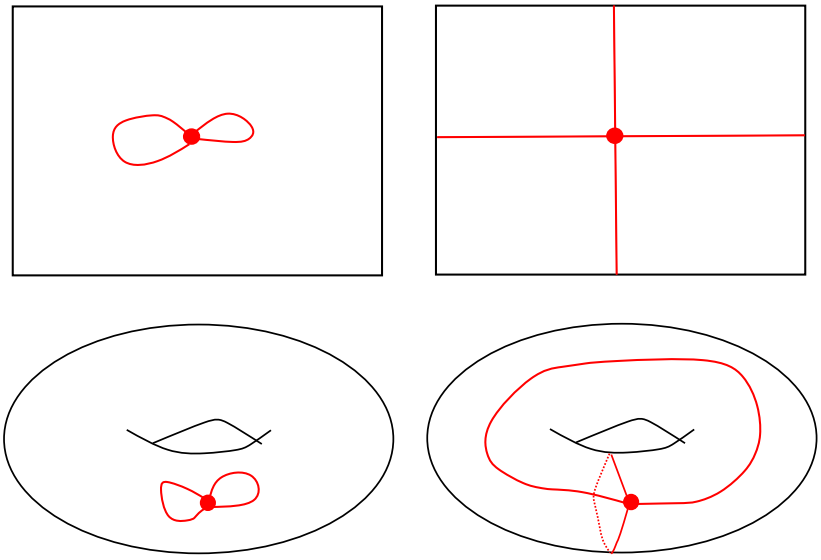

Figure 3: Two embeddings of the graph with one vertex and two loops into the torus are shown (the standard square depiction of the torus above and a sketch below). In the left figure, the complement of the embedding is homeomorphic to a punctured torus and two disks, so the embedding is noncellular. In the embedding on the right, the complement is homeomorphic to a disk, so the embedding is cellular.

In this dissertation we are only concerned with the case where A and B are points and the surface is compact with no boundary.

Definition 2.10. An *embedding* of a topological graph Γ into a topological space X is a continuous map $f : \Gamma \to X$ such that the restriction $\hat{f} : \Gamma \to f(\Gamma)$ is a homeomorphism. We say f is a *cellular embedding* if $X \setminus f(\Gamma)$ is homeomorphic to a collection of open disks.

Given a topological graph Γ, a surface X, and an embedding $f : \Gamma \to X$ with $f(\Gamma)$ a minimal separating set in X, there are infinitely many embeddings of Γ which are minimal separating in X, because we can consider deformations of $f(\Gamma)$ (Figure 4 illustrates this). For example, any simple closed curve drawn on

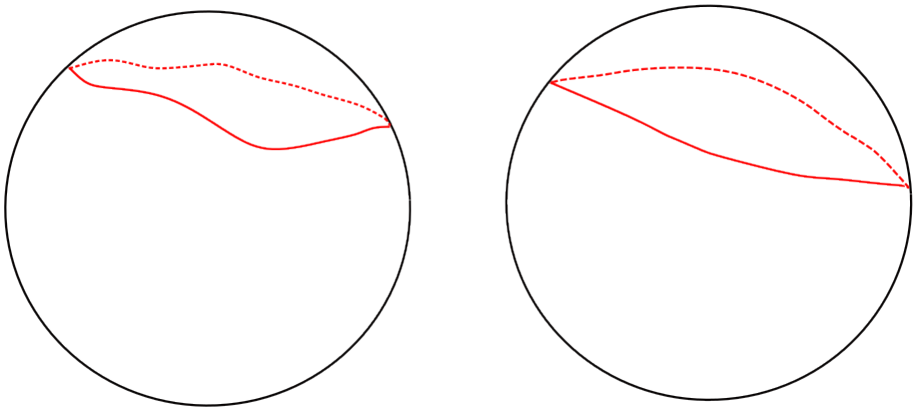

Figure 4: Two minimal separating sets are shown (red) on the sphere S^2. Although the curves are visibly different, both are homeomorphic to the circle S^1.

the sphere is homeomorphic to a topological graph with a single vertex and a single loop edge at that vertex. So, the idea of 'counting' minimal separating sets is fairly meaningless without a some equivalence relation on minimal separating sets.

One reasonable equivalence relation to use would be to consider two minimal separating sets $L_1, L_2 \subseteq X$ to be equivalent if the underlying topological graphs are homeomorphic. This is the approach taken in [31]. Classifying minimal separating sets up to this level of equivalence was our original goal, and we accomplish this in Chapter 7 with a computer program for surfaces of genus ≤ 5. With this equivalence relation in mind, we define the following three families of sets:

Definition 2.11. 1. $\mathcal{M}_g$ is the set of all graphs which can be realized as minimal separating sets in X_g.

2. $\mathcal{L}_g$ is the set of all graphs which can be realized as minimal separating sets in X_g, but not in $X_{\hat{g}}$ for $\hat{g} < g$. We say these are the graphs with *least separating genus* g.

3. $\mathcal{C}_g$ is the set of all *connected* graphs with least separating genus g.

It was observed in [3] that any graph graph which embeds as a minimal separating set in X_g also embeds as a minimal separating set in $X_{\hat{g}}$ for $\hat{g} \geq g$, so we immediately have

Theorem 2.7. $\mathcal{L}_g = \mathcal{M}_g \setminus \mathcal{M}_{g-1}$.

This motivates us to focus our efforts on the study of $\mathcal{L}_g$. In Chapter 4, we will introduce powerful tools for the study of embeddings of connected graphs, so a key step will be to relate the $\mathcal{L}_g$ to the $\mathcal{C}_g$.

However, it turns out that focusing on a finer equivalence relation not only makes the computation much faster, but also relates our research to enumeration of combinatorial maps (see Chapter 4), Hurwitz numbers, and matrix integrals (see Chapter 8 for the latter two topics). With this in mind we introduce the following type of equivalence:

Definition 2.12. For $f_1 : \Gamma \to X$, and $f_2 : \Gamma \to Y$ a pair of embeddings of a graph Γ, we say f_1 and f_2 are *equivalent embeddings* if there is an orientation-preserving homeomorphism $\varphi : X \to Y$ such that $f_2 = \varphi \circ f_1$.

From now on, we will frequntly use the much shorter phrase *minimal separating set* in place of the tediously long but more precise *class of minimal separating sets*

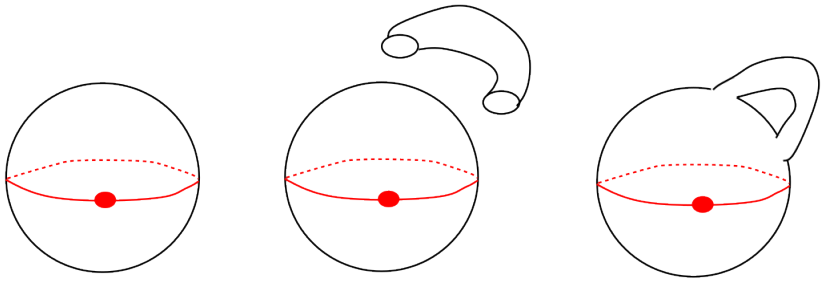

Figure 5: Left: A minimal separating embedding of the graph with a single loop in the sphere. Center and Right: Adding a handle to give a minimal separating embedding of the same graph in the torus.

which are equivalent embeddings of the same topological graph. This is closer to the kind of equivalence relation we would like, but still not quite right. Given any graph G embedded in a surface X as a minimal separating set, we can construct an infinite family of embeddings of G in surfaces of increasing genera that all "feel the same". The idea is that we can add as many handles as we want to a one of the connected components of $X \setminus G$ and retain a minimal separating set in the new surface. See Figure 5 for an example with the graph consisting of one loop.

These embeddings "feel the same" and we can make this more precise by observing that if we restrict our view of the surface to a small neighborhood of the embedding, these embeddings are all equivalent. There's a handy object from the study of graph embeddings in topology that captures what we are talking about, the ribbon graph. Intuitively, this object is a thickening of an embedding so that it becomes a submanifold of the surface. More formally:

Definition 2.13. Let $\eta : \Gamma \to X$ be an embedding of a graph Γ in a surface X.

1. The *ribbon graph* $R(\eta)$ is a neighborhood of $\eta(G)$ which is small enough to retract to $\eta(G)$. This is called a *collared neighborhood*.

2. X_η is the surface obtained by gluing a disk along each boundary component of $R(\eta)$, but 'remembering' the decomposition into disks and $R(\eta)$. We call those disks the *faces* of R).

Both $R(\eta)$ and X_η are treated as oriented surfaces inheriting their orientations from X. If G is a connected graph, then both $R(\eta)$ and X_η are connected so it makes sense to talk about the genus of X_η. That leads to the next definition:

Definition 2.14. Let $\eta : G \to X$ be an embedding of a *connected* graph G. Then we define the genus g_R of $R(\eta)$ to be the genus of X_η. Then it is given by the formula

$$g_R = \frac{2 - V + E - F}{2}$$

where V is the number of vertices of G, E the number of edges of G, and F the number of faces of $R(\eta)$.

Sometimes in other literature the ribbon graph is defined as X_η. This is because much work on graph embeddings focuses on the case of cellular embeddings, where any one of η, X_η, and $R(\eta)$ is enough to recover (up to homeomorphism) the other two. Unfortunately, we cannot restrict ourselves to cellular embeddings. See Figure 6 for an example of a noncellular embedding which gives a minimal separating set.

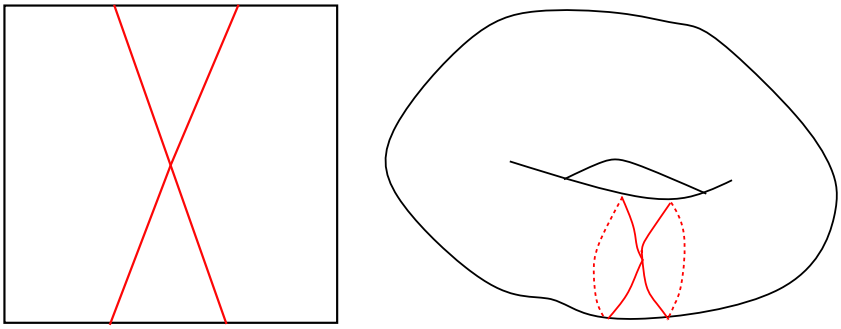

Figure 6: Above we see two depictions of an embedding of the graph with one vertex and two loops embedded as a minimal separating set in the torus. The complement of the embedding has two components, one homeomorphic to a disk, and the other homeomorphic to a punctured disk, giving an embedding which is both noncellular and minimal separating

From the definition, we immediately have that for any graph G with two equivalent embeddings $\eta_1 : G \to X$, $\eta_2 : G \to Y$, $R(\eta_1)$ is homeomorphic to $R(\eta_2)$ (by an orientation preserving homeomorphism). In fact, in the case where η_1, η_2 are both cellular embeddings, it follows from Theorems 1 and 2 of [7] that η_1 and η_2 are equivalent *if and only if* there is an orientation preserving homeomorphism from $R(\eta_1)$ to $R(\eta_2)$.

Considering the ribbon graph of an embedding, rather than the image of the embedding is often useful in proofs, because of the following issue: Suppose we have an embedding $\eta : G \to X$ and we want to examine a connected component A of $X \setminus \eta(G)$, A will not be compact. We will find ourselves wanting to use the classfication of surfaces, and want to find a compact subset of X related to A that still captures what we are looking at. A logical choice for this would be to

observe: The image $\eta(G)$ is the boundary of A in X, so $A \cup \eta(G)$ is a compact subset of X. Unfortunately, outside of some special cases, $A \cup \eta(G)$ is usually not a surface. If we look at a vertex of G along the boundary of $A \cup \eta(G)$, we see that unless the vertex has degree 1 or 2, it has no neighborhood homeomorphic to a half-disk, so $A \cup \eta(G)$ is not a surface with boundary (see Figure 6).

Looking at the ribbon graph $R(\eta)$ solves this issue. Since $R(\eta)$ retracts to $\eta(G)$, it is separating if and only if $\eta(G)$ is, and the components of $X \setminus R(\eta)$ are homeomorphic to the corresponding components of $X \setminus \eta(G)$. Now, rather than look at a component A of $X \setminus \eta(G)$, which is not compact, or a closure $A \cup \eta(G)$, which is not a surface, we can consider A with the interior of $A \cap R(\eta)$ removed. This is just a formal way of saying we are taking the component of $X \setminus R(\eta)$ that corresponds to A, and then we add its boundary with $R(\eta)$.

Now we can define one final family of sets which will be important to us, along with the $\mathcal{M}_g, \mathcal{L}_g$, and $\mathcal{C}_g$.

Definition 2.15. $\mathcal{R}_g$ is the set of *connected* ribbon graphs (up to homeomorphism) which can be realized as the ribbon graph of a minimal separating set in X_g, but not in X_{g-1}.

We can immediately relate $\mathcal{C}_g$ to $\mathcal{R}_g$ as follows:

Lemma 2.8. *A graph G is in $\mathcal{C}_g$ if and only if there is an embedding $\eta : G \to X_g$ with ribbon graph $R(\eta) \in \mathcal{R}_g$, but for any $\hat{g} < g$ there is no embedding $\psi : G \to X_{\hat{g}}$ with ribbon graph $R(\psi) \in \mathcal{R}_{\hat{g}}$.*

Proof. Suppose G is in $\mathcal{C}_g$. Then certainly there is an embedding $\eta : G \to X_g$ where $\eta(G)$ is a minimal separating set, and $R(\eta)$ is the ribbon graph of a minimal separating set in X_g. Since the topological graph underlying a ribbon graph is unique (up to homeomorphism), $R(\eta)$ cannot be realized as a minimal separating set in $X_{\hat{g}}$ for any $\hat{g} \leq g$, or else G would have a minimal separating embedding in a surface of genus $\hat{g}$, contradicting the *least separating genus* condition on $\mathcal{C}_g$. Thus $R(\eta) \in \mathcal{R}_g$. Similarly, if there was any embedding $\psi : G \to X_{\hat{g}}$ with $R(\psi) \in \mathcal{R}_{\hat{g}}$ for $\hat{g} < g$, we could recover from it a minimal separating embedding of G into $X_{\hat{g}}$, contradicting the least separating genus condition.

On the other hand, suppose that there is a surface X and embedding $\eta : G \to X$ such that $R(\eta) \in \mathcal{R}_g$, but there is no surface Y and embedding $\psi : G \to Y$ with $R(\psi) \in \mathcal{R}_{\hat{g}}$ for any $\hat{g} < g$. Since $R(\eta) \in \mathcal{R}_g$, we know that G embeds as a minimal separating set in a surface of genus g so $G \in \mathcal{C}_{\bar{g}}$ for some $\bar{g} \leq g$. For purposes of contradiction, suppose $\bar{g} < g$. Then there exists a surface Y of genus $\bar{g}$ and a minimal separating embedding $\psi : G \to Y$. We then have $R(\psi) \in \mathcal{R}_{\hat{g}}$ for some $\hat{g} \leq \bar{g} < g$, a contradiction. □

In [3] it is shown that

$$\mathcal{M}_0 = \mathcal{L}_0 = \mathcal{C}_0 = \{\text{the graph with a single loop on a single vertex}\}.$$

Since we only consider orientable surfaces, the only possible ribbon graph for the graph with a loop on a single vertex is the annulus, and $\mathcal{R}_0 = \{\text{the annulus}\}$. From now on we will focus our attention on $g > 0$ and and can assume, at least

for connected graphs, that no vertices have degree 2 (which excludes the graph consisting of one vertex with a single loop).

3 Topological Results

The main goal of this section is to establish the necessary topological results to show that we can determine the elements of the sets $\mathcal{M}_g$ and $\mathcal{L}_g$ from the elements of the $\mathcal{C}_{\hat{g}}$ where $\hat{g} \leq g$, and to show that although the minimal separating embeddings of a graph need not be cellular embeddings, we can associate them to cellular embeddings (via ribbon graphs) in a way which will help us to enumerate them.

First we establish a useful lemma which follows almost immediately from the definition of minimal separating and the fact that we only consider oriented surfaces.

Lemma 3.1. *An embedding $\eta(G)$ of a graph G into a surface X is minimal separating in X if and only if $X \setminus \eta(G)$ has two connected components, A and B, and for every edge e of G, $\eta(e)$ has A on one side and B on the other.*

Proof. Suppose $\eta(G)$ is minimal separating. Then by defintion $X \setminus \eta(G)$ has at least two connected components, which we call $A, B, \ldots$. By minimality of $\eta(G)$, for any edge e of G we have $(X \setminus \eta(G)) \cup \eta(e)$ is connected. This means that $\eta(G)$ is incident to all the components of $X \setminus \eta(G)$. Then there must be exactly two components A and B of $X \setminus \eta(G)$, one lying on either side of $\eta(e)$.

Now, assume that $\eta(G)$ is an embedding of G into X such that $X \setminus \eta(G)$ has two connected components, A and B, and for each edge e of G, $\eta(e)$ is incident to A on one side and incident to B on the other. By definition, $\eta(G)$ is separating, but we need to show that it is *minimal* separating. Every point $x \in \eta(G)$ lies in

$\eta(e)$ for some edge e. Since A lies on one side of $\eta(e)$ and B lies on the other, we have a path from any point in A to x, and a path from any point in B to x. Thus $(X \setminus \eta(G)) \cup \{x\}$ has a single connected component and $\eta(G) \setminus \{x\}$ is not separating for any $x \in \eta(G)$. This shows $\eta(G)$ is minimal separating. □

The following definition will let us restate the above lemma in a more concise form:

Definition 3.1. An *n-face-coloring* of a graph embedding $\eta : G \to X$ is a partition of the set of components of $X \setminus \eta(G)$ into n blocks. Traditionally each block is named with a color. We say an n-face-coloring is *proper* if every edge of G is incident to two components with distinct colors.

Then Lemma 3.1 can be restated as saying: An embedding $\eta(G)$ is minimal separating if and only if $X \setminus \eta(G)$ has 2 connected components and they can be properly 2-face-colored.

Lemma 3.2. *Let $G \in \mathcal{L}_g$, $\phi : G \to X_g$ be a minimal separating embedding of G into X_g, and A, B be the connected components of $X_g \setminus R(\phi)$. Then the surfaces $\bar{A}$ and $\bar{B}$ (the closure of A and B respectively) both have genus 0.*

Proof. If $\bar{A}$ has genus greater than 0, we could cut a handle in it and call the resulting surface $\bar{A}'$. The boundary would be unchanged, so we could glue $\bar{A}'$ back onto $R(\phi)$, and glue $\bar{B}$ back onto $R(\phi)$ to obtain new surface X', the result of cutting a handle in X_g. $R(\phi)$ would still minimal separating in this new surface of genus $g - 1$. This would contradct the assumption that $G \in \mathcal{L}_g$. □

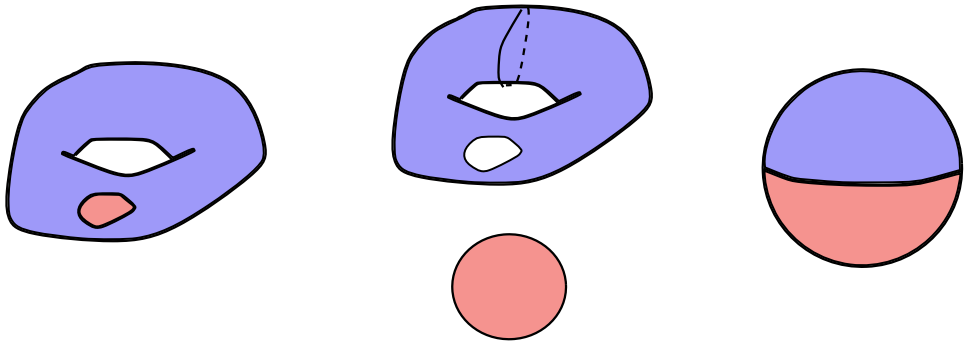

Figure 7: Here we see a circle S^1 embeded as a minimal separating set in the torus T. One component of $T \setminus S^1$ is homeomorphic to a puctured torus, but by "cutting it open" and gluing on disks to "plug" the holes we obtain a minimal separating embedding in the sphere.

As a corollary (of the proof) we have

Corollary 3.2.1. *Let R be a connected ribbon graph. $R \in \mathcal{R}_g$ if and only if R can be realized as the ribbon graph of a minimal separating set in X_g and $\bar{A}, \bar{B}$ have genus 0 (where A and B are the connected components of $X \setminus R$).*

As a further corollary we have

Corollary 3.2.2. *Let R be a connected ribbon graph and η an embedding with $R = R(\eta)$ and assume that the faces of X_η can be 2-colored so that each edge of R is incident to one red face and one blue face. Then $R \in \mathcal{R}_g$ if and only if*

$$F = 2 + g - g_R$$

where F is the number of faces of R. Since $F \geq 2$ this gives $g_R \leq g$.

Proof. Suppose $R \in \mathcal{R}_g$ and let η be a minimal separating embedding of a graph G into X_g with R as its ribbon graph. Let A, B be the connected components of $X_g \setminus R$. By the previous corollary, $\bar{A}$ and $\bar{B}$ both have genus 0 and are homeomorphic to spheres each with some number of disks removed. Specifically, they're homeomorphic to spheres with one disk removed for each component of their boundary with $R(\eta)$. With this in mind, we now reconstruct X_g from X_η while preserving the minimal separating status of $R(\eta)$ as follows:

Split the faces of X_η into two sets F_A and F_B according to whether they were glued to $R(\eta)$ along the boundary with A or the boundary with B. Now select a disk $D \in F_A$ as a 'base disk' and for each other disk D' in F_A we add a handle joining D to D' in X_η. By connecting each face in F_A to D in this fashion, we have connected those disks into a single connected surface. Since cutting any of the cylinders will disconnect D from the disk on the other end, we haven't added any genus and by the classification of surfaces with boundary the resulting surface must be a sphere with a boundary component for each boundary with $R(\eta)$. By the Corollary 3.2.1 this is homeomorphic to $\bar{A}$. Doing the same process to the disks of F_B gives a surface homeomorphic to $\bar{B}$ as well.

Now we consider what this operation has done to X_η. We added $|F_A| - 1$ handles in order to reconstruct $\bar{A}$ and $|F_B| - 1$ handles in order to reconstruct $\bar{B}$. Since R is connected, so was X_η and each handle contributed 1 to the genus of the resulting surface. Then we have

$$g = g_R + (|F_A| - 1) + (|F_B| - 1) = g_R + F - 2$$

giving $F = 2 + g - g_R$ as desired.

Now suppose $R = R(\eta)$ for some η, the faces of X_η can be 2-colored so that each edge of R is incident to a red face and blue face, and X_η has $2 + g - g_R$ faces. Do exactly the procedure described above to connect all the blue faces into a single connected component and all of the red faces into a single connected component by adding handles. As shown above, it produces the surface X_g, and contracting R to its underlying topological graph gives a minimal separating set in X_g, by Lemma 3.1. This shows that $R \in \mathcal{R}_{g'}$ for some g' and then the applying the first part of the proof gives $g' = g$. □

Now we have a precise condition on whether a ribbon graph R is in any of the $\mathcal{R}_g$ for some g, and what that value of g would be. For $R = R(\eta)$ we construct the related surface X_η. If the faces of X_η can be colored red and blue such that each edge has a red face on on side and a blue face on the other, then $R \in \mathcal{R}_g$ for g determined by the above corollary.

Next we will describe the elements of the $\mathcal{L}_g$s in terms of the elements of the $\mathcal{C}_g$s and use this to find a formula for $|\mathcal{L}_g|$ in terms of $|\mathcal{C}_{\hat{g}}|$ for $\hat{g} \leq g$. Our first step is the following theorem and corollary:

Theorem 3.3. *Let G be a topological graph which is the disjoint union of graphs G_1 and G_2. Then*

$$G \in \mathcal{L}_g \Leftrightarrow (G_i \in \mathcal{L}_{g_i} \quad \textit{with } g = g_1 + g_2 + 1)$$

Proof. Let $G \in \mathcal{L}_g$ and η be a minimal separating embedding of G in X_g. We will let η_1 and η_2 be the restrictions of η to G_1 and G_2 respectively. Consider the decomposition of X_g into $R(\eta) = R(\eta_1) \cup R(\eta_2)$ and the two connected components of $X \setminus R(\eta)$ (see Figure 8).

By Lemma 3.2, we know that each component of $X_g \setminus R(\eta)$ has genus 0. So, we can draw a simple closed curve on each component of $X_g \setminus R(\eta)$ which separates the boundary components incident to $R(\eta_1)$ from the boundary components incident to $R(\eta_2)$. Cutting these two closed curves (one for each componet of $X_g \setminus R(\eta)$) and gluing disks onto the new boundaries we obtain two connected surfaces X' and X'', with $R(\eta_1)$ the ribbon graph of an embedding of G_1 in X' and $R(\eta_2)$ the ribbon graph of an embedding of G_2 in X'' (see Figure 8).

Observe that in X' each edge of $R(\eta_1)$ is still incident to the blue component and the red component, and our cutting has not connected the two components, so the induced embedding of G_1 in X' is minimal separating. Similarly the induced embedding of G_2 in X'' is minimal separating. Thus there exist g_1, g_2 with $G_1 \in \mathcal{L}_{g_1}$ and $G_2 \in \mathcal{L}_{g_2}$, and X' has genus at most g_1, while X'' has genus at most g_2. We recover X from X' and X'' by taking their connected sum (glued together within the blue component of each surface) and adding a handle connecting the red components. This recovers a surface where $R(\eta)$ contracts to a minimal separating set and by Theorem 2.5 and Theorem 2.4 it has genus $g_1 + g_2 + 1$. Thus

$$g \geq 1 + g_1 + g_2.$$

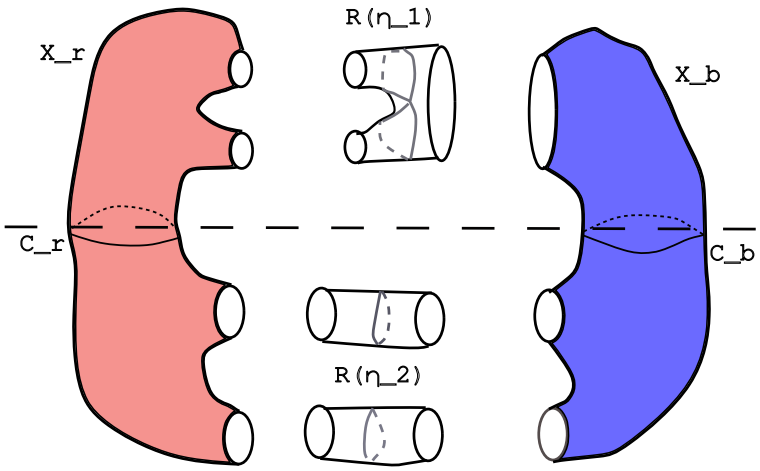

Figure 8: An example where G_1 is a single vertex with two loops, G_2 is a pair of isolated loops, and gluing the components together recovers a minimal separating embedding of $G = G_1 \cup G_2$ in a surface

Now G_1, G_2 be topological graphs with $G_1 \in \mathcal{L}_{g_1}$ and $G_2 \in \mathcal{L}_{g_2}$ and let G be the disjoint union of G_1 and G_2. Let $\eta_1 : G_1 \to X_{g_1}$ and $\eta_2 : G_2 \to X_{g_2}$ be minimal separating embeddings. Then we can construct a minimal separating embedding of G in $X_{1+g_1+g_2}$ as follows: Color the connected components of $X_{g_1} \setminus \eta_1(G_1)$ red and blue, and do the same for the components of $X_{g_2} \setminus \eta_2(G_2)$. Take the connected sum of X_{g_1} and X_{g_2} by gluing within the blue components and add a handle connecting the red components. We've now constructed a surface of genus $g_1 + g_2 + 1$ and an embedding of G into it with a2-face-coloring. By construction each edge of G is incident to the blue component on one side and the red component on the other, so the embedding is minimal separating. Thus $G \in \mathcal{L}_g$ for some g and we know that

$$g \leq 1 + g_1 + g_2.$$

Combining the two inequalities we obtain the desired result. □

The following corollary gives the relation between the $\mathcal{L}_g$s and $\mathcal{C}_g$s that we have been looking for

Corollary 3.3.1. *Let G be a topological graph with connected components $G_1, G_2, \ldots, G_k$. $G \in \mathcal{L}_g$ if and only if there exist $g_1, g_2, \ldots, g_k$ such that $G_i \in \mathcal{C}_{g_i}$ for each i and*

$$g = g_1 + g_2 + \ldots + g_k + (k - 1).$$

Proof. The proof follows from Theorem 3.3 by induction on the number of con-

nected components of G. $\square$

Now we can give the following formula for the cardinality of $\mathcal{L}_g$, which we denote by $|\mathcal{L}_g|$.

Theorem 3.4. *Let* $K_g = \{(k_0, k_1, \ldots, k_g) : g = (\sum_{i=0}^{g}(i+1)k_i) - 1\}$. *Then*

$$|\mathcal{L}_g| = \sum_{(k_0,\ldots,k_g)\in K_g} \prod i = 0^g \binom{|\mathcal{C}_i| + k_i - 1}{k_i}$$

Proof. From Corollary 3.3.1 we know that the graphs in $\mathcal{L}_g$ are precisely those whose connected components $G_1, G_2, \ldots, G_\ell$ satisfy $G_i \in \mathcal{C}_{g_i}$ for some i and

$$g_1 + g_2 + \ldots + g_\ell + (\ell - 1) = g.$$

Now for each $i \in \{0, 1, \ldots, g\}$ let $k_i = |\{G_1, G_2, \ldots, G_\ell\} \cap \mathcal{C}_i|$, the number of the G_j which are elments of $\mathcal{C}_{g_i}$. Then we have:

$$g = \left(\sum_{i=0}^{g} ik_i\right) + (\ell - 1)$$

and since each of the G_j is in exactly one of the $\mathcal{C}_i$ we have $\ell = \sum_{i=0}^{g} k_i$ giving

$$\begin{aligned} g &= \left(\sum_{i=0}^{g} ik_i\right) + \left(\sum_{i=0}^{g} k_i\right) - 1 \\ &= \left(\sum_{i=0}^{g} (i+1)k_i\right) - 1 \end{aligned}$$

This means that $G \in \mathcal{L}_g$ if and only if this equality holds. The remaining question to answer here is: For a given g, how many graphs G have k_i connected components in $\mathcal{C}_i$?

So, for a graph $G \in \mathcal{L}_g$ define $K(G) := (k_0, \ldots, k_g)$ where k_i is the number of connected components of G in $\mathcal{C}_i$. Since two graphs G and G' are isomorphic if and only if there is a bijection between their connected components which takes distinct components of G to distinct isomorphic components of G', the number of graphs G with $K(G) = (k_0, \ldots, k_g)$ is equal to the number of ways to choose (with replacement) k_0 elements of $\mathcal{C}_0$, times the number of ways to choose k_1 elements of $\mathcal{C}_1$,..., times the number of ways to choose k_g elements of $\mathcal{C}_g$.

The number of ways to choose k objects with replacement from a set of size n is equal to $\binom{n+k-1}{k}$, which can be seen as follows: Line up the n objects and each time you select an object, place a marker to the right of the object, so that any selection is represented as a string of $n + k$ symbols (n objects and k markers), with the only restriction being that the first position is not a marker. There are $\binom{n+k-1}{k}$ such choices [29]. This gives the desired formula:

$$|\mathcal{L}_g| = \sum_{(k_0,\ldots,k_g)\in K_g} \prod_{i=0}^{g} \binom{|\mathcal{C}_{g_i}| + k_i - 1}{k_i}.$$

□

4 Combinatorial Maps

In this section we introduce the language of *combinatorial maps* and *hypermaps* and state some of their relevant properties. Combinatorial maps provide an encoding of a cellular embedding of a connected graph in a surface as an ordered triple of permutations with properties that we will find very useful. Most obviously, an encoding of a graph embedding as a permutation allows us to store it efficiently in a computer. Additionally, two cellular graph embeddings are equivalent if and only if they are associated to isomorphic combinatorial maps. Furthermore, in some limited cases we can use the representation theory of the symmetric groups to enumerate minimal separating embeddings up to embedding equivalence without need of a computer. Hypermaps are a generalization of combinatorial maps. We will see later that translating from combinatorial maps to hypermaps adds some complications to our work but yields great computational benefits and simplifies some of the relevant representation theory.

A key motivating idea for combinatorial maps comes from the goal of recovering a ribbon graph from discrete data. Topologically, a ribbon graph is a collection of disks (one per vertex) with copies of $[0,1] \times [0,1]$ (one for each edge) glued to their boundaries. Since we only consider oriented surfaces, there are no half-twists on any of the edges, and knowing the cyclic ordering of the edges glued to each vertex is enough to recover the ribbon graph (up to homeomorphism).

Following [17] we give the following definition of a combinatorial map.

Definition 4.1. A combinatorial map is a triple of permutations $(\sigma, \alpha, \varphi)$ in the

symmetric group S_{2n} (for some n) satisfying:

1. α is an involution with no fixed points
2. $\varphi \circ \alpha \circ \sigma = ()$
3. The permutation group $\langle \sigma, \alpha, \varphi \rangle$ acts transitively on $\{1, \ldots, 2n\}$.

The last part of the defintion, that $\langle \sigma, \alpha, \varphi \rangle$ acts transitively will correspond to requiring that a ribbon graph is connected. While we allow ribbon graphs in general to be connected or disconnected, this part of the definition is convenient for us, as the sets $\mathcal{R}_g$ that we wish to enumerate consist of connected ribbon graphs.

Given a embedding η of a *connected* graph G in an oriented surface X, we can associate to η a combinatorial map as follows:

1. Let n equal the number of edges in G, and label each end of each edge with a distinct integer in $\{1, 2, \ldots, 2n\}$.
2. Define α as the permutation which swaps each edge end label with the label on the other end of its edge.
3. Define σ to be the permutation which maps edge end i to the next edge end at the same vertex, where 'next' is with respect to the orientation on X.
4. Define $\varphi = (\alpha \circ \sigma)^{-1}$

By this construction the cycles of α correspond to the edges of G and the cycles of σ correspond to the vertices. In the case where φ is a cellular embedding the

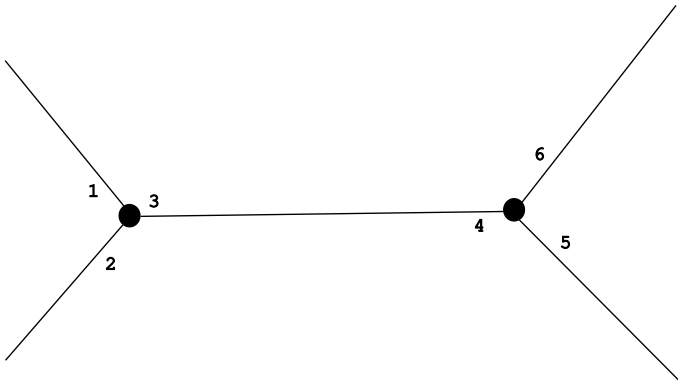

Figure 9: Considering the labels 2 and 4, which are drawn in the same face, we can see that applying α^{-1} to 4 would take it to 3, and the applying σ^{-1} would take it to 2, so $\varphi(4) = 2$. This matches the drawing precisely.

faces of the embedding correspond to the cycles of φ and in fact each cycle of φ gives the oriented boundary walk along its corresponding face. This is illustrated by Figure 9. For a proof, see [17] . In this figure (and all future figures of labelled graph embeddings) we are going to adopt a convention from [17]. Labels on edge ends will always be drawn on the left from the perspective of a walker who starts at that edge end and is walking to the other end of the edge.

We can reverse this process and go from a combinatorial map to a cellular graph embedding as well. We let $\{1, 2, \ldots, 2n\}$ be the set of edge ends of our graph. For each cycle of σ we make a vertex, and take the elements of that cycle (in order) as the edge ends placed incident to σ. Then we glue edge ends together into full edges according to the cycles of α, and finally glue disks onto the resulting topological graph according to the cycles of φ.

Ribbon Graph Property	Combinatorial Map Property
Number of Vertices	$c(\sigma)$
Number of Edges	$c(\alpha)$
Number of Faces	$c(\varphi)$

Table 1: Glossary for translating between ribbon graphs and combinatorial maps

This correspondence between cycles of σ, α, and φ and vertices, edges, and faces of a cellular embedding, means we will be frequently referring to the number of disjoint cycles of permutations. We introduce the following notation to make this more convenient:

Definition 4.2. Given any permutation ρ we define $c(\rho)$ to be the number of disjoint cycles of ρ.

The glossary in Table 1 gives a translation of some properties of a ribbon graph to properties of a corresponding combinatorial map.

Definition 4.3. The *genus* of a combinatorial map $(\sigma, \alpha, \varphi)$ is defined to be the genus g_R of the corresponding ribbon graph. It satisfies the equation:

$$g_R = \frac{2 - c(\sigma) + c(\alpha) - c(\varphi)}{2}.$$

Note that the formula above is a translation of the formula from Definition 2.14 via the glossary in Table 1.

Motivated by our desire to use a computer to find minimal separating sets and the fact that it is much easier to store and work with permutations on a computer than pictures or continuous maps from graphs to topological surfaces, we would

like to count the elements of $\mathcal{R}_g$ by counting combinatorial maps, rather than starting with ribbon graphs.

If we associate the same combinatorial map to a pair of cellular graph embeddings it follows that the embeddings are equivalent. The cycles of φ give a dissection of the underlying surface into polygons, and the cycles of α give the instructions on how to glue the polygons together. The reverse, that equivalent embeddings necessarily give rise to the same map, is not true. The first step in our construction of a combinatorial map from a graph embedding in a surface was a choice of labelling on the edge ends. Hence our next definition:

Definition 4.4. We say that two combinatorial maps $(\sigma_1, \alpha_1, \varphi_1)$ and $(\sigma_2, \alpha_2, \varphi_2)$ (both composed of elements of S_{2n}) are *isomorphic* if there exists permutation $\rho \in S_{2n}$ such that

$$\rho \circ \sigma_1 \circ \rho^{-1} = \sigma_2, \quad \rho \circ \alpha_1 \circ \rho^{-1} = \alpha_2, \text{ and therefore } \rho \circ \varphi_1 \circ \rho^{-1} = \varphi_2.$$

A useful way to think of this definition is that two combinatorial maps are isomorphic if and only if they differ only by the choice of labelling. To make this interpretation clearer: The symmetric group S_{2n} has a natural action on the integers $\{1, 2, \ldots, 2n\}$, so given $\rho \in S_{2n}$ and a labelling of edge ends on a cellularly embedded graph, we can interpret ρ as the following relabeling of edge ends: The edge end that was originally labelled i is now labelled $\rho(i)$.

Now when we restate the isomorphism condition as: $(\sigma_1, \alpha_1, \varphi_1) \simeq (\sigma_2, \alpha_2, \varphi_2)$ if

there exists $\rho \in S_{2n}$ such that

$$\rho \circ \sigma_1 = \sigma_2 \circ \rho, \rho \circ \alpha_1 = \alpha_2 \circ \rho, \rho \circ \varphi_1 = \varphi_2 \circ \rho,$$

and we can see that this means we can relabel edge ends according to ρ so that if i and j are two ends of the same edge (so $\alpha_1(i) = j$) then

$$\rho(j) = \rho(\alpha_1(i)) = \alpha_2(\rho(i)),$$

so that ρ takes the two ends of an edge in the first map to the two ends of an edge in the second map. An analogous result occurs for the ordering of edge ends about vertices (given by the σs) and for the boundary of each face (given by the φs).

The following theorem is a combination of Theorems 1 and 2 in [7], restated in our language. It is a consequence of the bijection between isomorphism classes of combinatorial maps and ribbon raphs sharing the same underlying graph (and the correspondence between ribbon graphs and cellular embeddings).

Theorem 4.1. *[7] Let G be a topological graph and let $f_1 : G \to X_1, f_2 : G \to X_2$ be cellular embeddings of G. Then f_1 and f_2 are equivalent embeddings if and only if any combinatorial maps associated to them are isomorphic.*

We can already naturally translate Corollary 3.2.2 into the following statement about combinatorial maps. It says that a ribbon graph R is in $\mathcal{R}_g$ if and only if the faces can be properly 2-colored and a combinatorial map $(\sigma, \alpha, \varphi)$ corresponding

to R satisfies

$$g = \hat{g} + c(\varphi) - 2$$

where $\hat{g}$ is the genus of $(\sigma, \alpha, \varphi)$. We will come back to this at the end of this chapter when we can restate Corollary 3.2.2 particularly cleanly.

At this point an algorithm to compute the elements of $\mathcal{R}_g$ starts to take shape. If we could bound the number of edges, E, for a ribbon graph in $\mathcal{R}_g$, which we will do in Lemmas 6.3 and 6.2, we could exhaustively search for combinatorial maps made of elements of S_E for each possible E. Without any modifications, such an algorithm would be wildly inefficient. We'd like to restrict the search space to a much smaller set of combinatorial maps and we can. For example, since we restrict all our connected topological graphs (and thus ribbon graphs) to have no degree two vertices (other than the graph which is one vertex with one loop), our algorithm shouldn't search through combinatorial maps with degree two vertices. We also want to require that our maps can be properly 2-face-colored. This condition is a little trickier to ensure ahead of time. The following definition of the *dual* of a combinatorial map will help us with this condition.

Definition 4.5. The *dual* of a combinatorial map $(\sigma, \alpha, \varphi)$ is the combinatorial map $(\varphi, \alpha, \sigma)$.

The definition is stated in the language of permutations, but there is an intuitive geometric interpretation. When we consider a combinatorial map as a cellular embedding of a graph in a surface, taking the dual exchanges faces with vertices, like the more familiar dual of polyhedra or a plane graphs. As shown in Figure 10

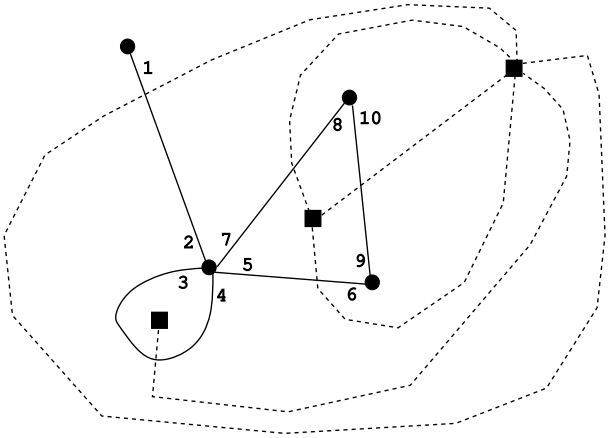

Figure 10: The circles and solid lines show the combinatorial map, $\sigma = (1)(2,3,4,5,7)(6,9)(8,10), \alpha = (1,2)(3,4)(5,6)(7,8)(9,10), \varphi = (1,7,10,6,4,2)(3)(5,9,8)$. The squares give the vertices of the dual map, and the dashed lines its edges. If we assign labels to the edges of the dual map to match those of the original map we see that the permutation describing the vertices of the dual is φ, α is still the permutation describing the edge-end pairings, and the permutation describing faces of the dual map is σ.

the vertices and faces of the original map are now, respectively, the faces and vertices of the new map, with each edge running between the vertices corresponding to the faces it was incident to in the original map.

Considering the dual graph allows us to translate our statement about 2-face-coloring into a statment about vertex coloring (Definition 2.4). Note that taking the dual does not change the genus of a combinatorial map (see Definition 4.3), since taking a dual does not change the number of edges and simply exchanges the number vertices with the number of faces.

We can now restate Lemma 3.1 in terms of dual combinatorial maps (for connected graphs only)

Lemma 4.2. *Let Γ be a connected graph. An embedding $\eta(\Gamma)$ is minimal separating if and only if the corresponding combinatorial map has a bipartite dual map.*

Proof. The lemma is nearly just a restatement of Lemma 3.1 in terms of dual maps. A proper 2-face-coloring of a map becomes a proper 2-vertex-coloring upon taking the dual. We add the hypothesis that Γ is connected because Definition 4.1 only allows us to define combinatorial maps for connected graphs. □

Now a proper 2-face-coloring of a combinatorial map becomes a proper 2-vertex-coloring of the dual map, and a combinatorial map can be 2-face-colored if and only if the dual is bipartite. In the next chapter, we will introduce a generalization of combinatorial maps that will allow us to take advantage of this bipartite condition to help us find the elements of $\mathcal{R}_g$ more efficiently.

We can similarly restate Corollary 3.2.2 now in terms of dual combinatorial maps, and it becomes much neater:

Corollary 4.2.1. *Let $R(\eta)$ be a ribbon graph, and let $(\sigma, \alpha, \varphi)$ be the corresponding combinatorial map and let $g > 0$. $R(\eta) \in \mathcal{R}_g$ if and only if $(\varphi, \alpha, \sigma)$ is bipartite, $c(\sigma)$ has no cycles of length 2, and*

$$g = \frac{-c(\sigma) + c(\alpha) + c(\varphi)}{2} - 1$$

Proof. Elements of $\mathcal{R}_g$ must be connected, so all of the ribbon graphs we are considering here do in fact have corresponding combinatorial maps. The condition

that $(\varphi, \alpha, \sigma)$ is bipartite comes from the equivalence between a combinatorial map being properly 2-face-colorable and the dual being bipartite (properly 2-vertex-colorable). The condition that $c(\sigma)$ has no cycles of length 2 comes from the fact that we are excluding $g = 0$ (so we exclude the ribbon graph which is a single vertex with a single loop edge). Then the relating g to $c(\alpha), c(\varphi)$, and $c(\sigma)$ comes from the statement of Corollary 3.2.2, which says $g - g_R + 2 = F$, where F is the number of faces of X_η. Since $c(\varphi)$ is the number of faces of X_η, we obtain the equality from substituting $g_R = \frac{2-c(\sigma)+c(\alpha)-c(\varphi)}{2}$ from Definition 4.3. □

Before we move on to the next chapter, we feel some remarks should be made about why we are not using existing methods from the study of map enumeration (and hypermap enumeration in the next chapter). After all, these are active fields of research, going back to the 1960s with work such as [14] and [30] and continuing to the present day. Early work in the field was mostly concerned with developing techniques for enumerating labelled maps, or rooted maps (a rooted map is a map with a distinguished edge end) to reduce symmetries of the problem. More recently great progress has been made in enumerating maps and hypermaps up to isomorphism as we would want to, such as in [18], [21] and [22], but unfortunately applying these methods to our problems has a major obstacle. The methods devised in those papers enumerate families of maps (and hypermaps in [22]) by first enumerating all the maps (resp hypermaps) which can be realized as quotients of the desired objects by an automorphism. Here when we refer to the quotient of a map by an automorphism, we mean the map obtained by identifying all edge ends in the same orbit of the action of the automorphism, as shown in Figure 11.

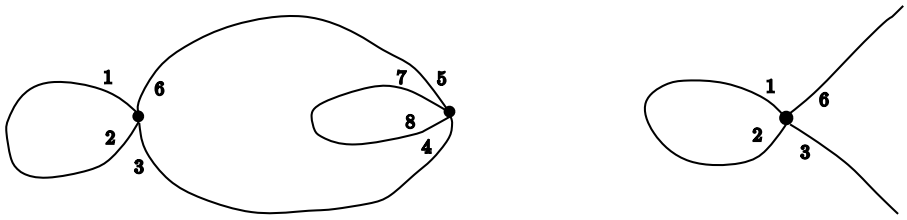

Figure 11: The combinatorial map $\sigma = (1,2,3,6)(4,5,7,8), \alpha = (1,2)(3,4)(5,6)(7,8)$ (left) and it is quotient (right) under the autmorphism $\rho = (1,7)(2,8)(3,4)(5,6)$. The unlabelled edge ends that occur are called *singular edges* in [18] and these would be fixed points under the involution α describing the quotient, which means it violates our definition of a combinatorial map. Instead [18] using an altered definition of combinatorial map which admits singular edges.

This method is clearly most practical when the quotients of the objects you're enumerating are a family of objects you've already counted. For example, it is very effective for enumerating maps with n edges, because a quotient necessarily has fewer edges, so if you work inductively you will have already enumerated the possible quotient objects. In our case, it is very possible that there may be quotients of a minimal separating embedding which are either not minimal separating or have degree 2 vertices, leading to an entirely new enumeration problem. In fact, as shown in Figure 11 they may not even be maps at all according to our definition. This related problem of determining precisely which maps and "map-like-objects" can be realized as quotients of minimal separating sets is a possible avenue for research and it may turn out to be a solvable problem which could lead to a way to enumerate elements of $\mathcal{R}_g$ without a computer, but for now we leave that investigation to future work.

5 Hypermaps

In this section we introduce a natural generalization of combinatorial maps which will allow us to more efficiently compute the elements of the $\mathcal{R}_g$. As a nice additional side benefit, they'll also simplify some of the representation theory used in Appendx A to determine how to preallocate memory.

In the definition of a combinatorial map, the first requirement, that α is an involution, is oddly specific, and combinatorial maps have a natural generalization that drops this requirement.

Definition 5.1. A *hypermap* is an ordered triple of permutations $(\sigma, \alpha, \varphi)$ in S_n satisfying

1. $\varphi \circ \alpha \circ \sigma = 1$.
2. The permutation group $\langle \sigma, \alpha, \varphi \rangle$ acts transitively on $\{1, 2, \ldots, n\}$.

Hypermaps are to combinatorial maps as hypergraphs are to graphs, so it is common to refer to the cycles of σ as the vertices of a hypermap, the cycles of α as the hyperedges, and the cycles of φ as the faces. In the context of hypermaps, the elements of $\{1, 2, \ldots, n\}$ are commonly referred to as *bits*, *brins*, or *darts*. This definition and the upcoming definitions of hypermap isomorphism and genus follow [17].

Just as was the case for combinatorial maps, we want to consider two hypermaps isomorphic if they differ by a relabeling. As a result we again say:

Definition 5.2. Two hypermaps $(\sigma, \alpha, \varphi)$ and $(\sigma', \alpha', \varphi')$ (both with n bits) *isomorphic* if there exists $\rho \in S_n$ such that

$$\rho \circ \sigma \circ \rho^{-1} = \sigma', \quad \rho \circ \alpha \circ \rho^{-1} = \alpha', \text{ and therefore } \rho \circ \varphi \circ \rho^{-1} = \varphi'.$$

We call such a ρ an *isomorphism of hypermaps*

The genus of a hypermap is defined as

Definition 5.3. The *genus* of a hypermap $(\sigma, \alpha, \varphi)$ with elements in S_n is define to be equal to

$$g_R = \frac{2 - c(\sigma) - c(\alpha) + n - c(\varphi)}{2}.$$

We denote this by g_R, because the next theorem will show that each hypermap is associated to a combinatorial map (and therefore a ribbon graph), and the hypermap genus as defined is equal to the genus of that ribbon graph.

In terms of the correspondence between graph embeddings and combinatorial maps, relaxing the requirement that α be an involution without fixed-points might be most naturally seen as allowing edges with more (or less) than 2 ends, and hypermaps would describe hypergraph embeddings (hence the name). Under this interpretation we would now call the cycles of α the *hyperedges* of the hypermap. The following result of Walsh gives another interpretation of hypermaps which is more connected with our investigation of minimal separating sets:

Theorem 5.1. *[34] There is a one-to-one correspondence between isomorphism classes of hypermaps and isomorphism classes of 2-vertex-colored bipartite com-*

Ribbon Graph	Combinatorial Map	Hypermap of Dual Map
Number of Vertices	$c(\sigma)$	$c(\bar{\varphi})$
Number of Edges	$c(\alpha)$	n
Number of Faces	$c(\varphi)$	$c(\bar{\sigma}) + c(\bar{\alpha})$

Table 2: A glossary for translating between ribbon graphs that can be properly 2-face-colored, their corresponding combinatorial map $(\sigma, \alpha, \varphi)$, and the hypermap $(\bar{\sigma}, \bar{\alpha}, \bar{\varphi})$ corresponding to a vertex-colored dual of the combinatorial map.

binatorial maps which preserves the genus and maps vertices, hyperedges, faces, and bits of a hypermap onto (respectively) the vertices of one color class, vertices of the other color class, faces, and edges of a 2-vertex-colored bipartite map.

Now the reasoning behind the definition of hypermap genus becomes clear. Since vertices and hyperedges of a hypermap $(\sigma, \alpha, \varphi)$ correspond to the two color classes of vertices in a 2-vertex-colored bipartite map, $c(\sigma)+c(\alpha)$ is the number of vertices in the corresponding map. The bits of a hypermap are mapped to edges of the bipartite map, so n is the number of edges, and similarly $c(\varphi)$ is the number of faces, since faces of the hypermap map to faces of the bipartite map. Then the hypermap genus formula is simply a translation of the genus formula for the corresponding combinatorial map. We organize this into a new glossary (but comparing hypermaps to the duals of combinatorial maps since that is our actual use case) in Table 2.

Not only does the correspondence from Theorem 5.1 exist, but the correspondence is relatively straightforward. Here we give a description of how to take a 2-vertex-colored bipartite combinatorial map and construct the corresponding hypermap, but a picture is the best kind of explanation. See Figure 12 for an example of a

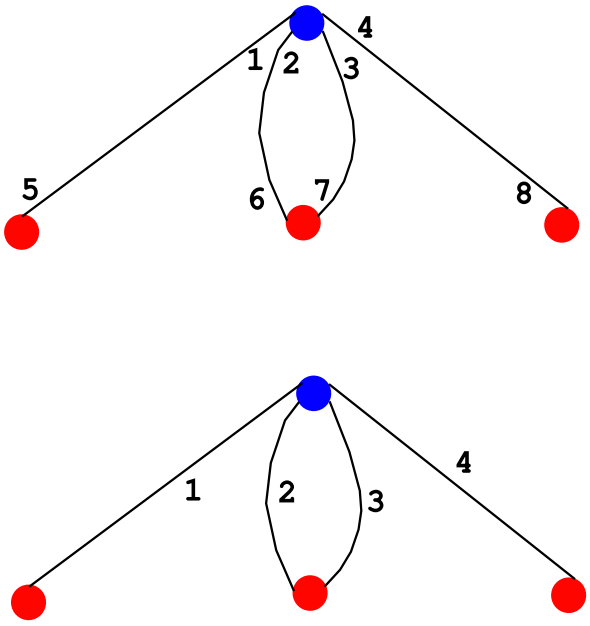

Figure 12: Above is a drawing of the combinatorial map $(\sigma, \alpha, \varphi)$ where $\sigma = (1,2,3,4)(6,7)$, $\alpha = (1,5)(2,6)(3,7)(4,8), \varphi = (1,5,4,8,3,6)(2,7)$. Below is the corresponding hypermap $(\bar{\sigma}, \bar{\alpha}, \bar{\varphi})$ where $\bar{\sigma} = (1,2,3,4)$, $\bar{\alpha} = (2,3)$, $\bar{\varphi} = (1,4,3)$. Each drawing can be viewed as an embedding in the sphere.

2-vertex-colored bipartite combinatorial map and a corresponding hypermap.

The idea is to take a 2-vertex-colored bipartite combinatorial map $(\sigma, \alpha, \varphi)$ and consider the cycles corresponding to the blue vertices separately from the cycles corresponding to the red vertices. Since each edge of a 2-vertex-colored bipartite graph is incident to exactly one blue vertex and one red vertex, the cycles of σ corresponding to blue vertices induce a permutation on the edges of the graph (with one cycle per blue vertex) and so do the cycles of σ corresponding to the red vertices (but with one cycle per red vertex now). We label the edges $1, 2, \ldots, n$ and now the permutations on $\{1, 2, \ldots, n\}$ induced by the action of σ restricted to the blue vertices and red vertices will be respectively the $\bar{\sigma}$ and $\bar{\alpha}$ of the corresponding hypermap $(\bar{\sigma}, \bar{\alpha}, \bar{\varphi})$. For now we simply define $\bar{\varphi} = (\bar{\sigma}\bar{\alpha})^{-1}$ and we will look at its properties more soon. The definition of a hypermap requires $\langle \bar{\sigma}, \bar{\alpha}, \bar{\varphi} \rangle$ acts transitively on $\{1, \ldots, n\}$, so we verify that now.

Transitivity follows from the fact that we started with a combinatorial map, so the underlying graph was connected. This means that for any two edges $i, j \in \{1, 2, \ldots, n\}$ there is some path: $e_0, e_1, \ldots, e_k$ with $e_0 = i$ and $e_k = j$. Consecutive edges in this path share a vertex, so they are in the same orbit of either $\bar{\sigma}$ or $\bar{\alpha}$ (depending on whether the shared vertex is red or blue). Then application of $\bar{\sigma}$ or $\bar{\alpha}$ the correct number of times maps any e_ℓ to $e_{\ell+1}$, and repeated application of $\bar{\sigma}$ and $\bar{\alpha}$ maps i to j, showing the action is transitive.

The process of going from a hypermap to the corresponding (isomorphism class) of 2-vertex-colored bipartite combinatorial map starts to show how we can gain

computationally from using hypermaps:

Starting with a hypermap $(\bar{\sigma}, \bar{\alpha}, \bar{\varphi})$ with n edges, we define $\alpha \in S_{2n}$ as the involution with $\alpha(i) = i + n$ for $1 \leq i \leq n$ and $\alpha(i) = i - n$ for $n + 1 \leq i \leq 2n$. Now we define $\sigma \in S_{2n}$ by:

$$\sigma(i) = \begin{cases} \bar{\sigma}(i) & \text{if } 1 \leq i \leq n \\ \bar{\alpha}(i-n) + n & \text{if } n+1 \leq i \leq 2n \end{cases}$$

The idea is as follows: We treat the permutations $\bar{\sigma}$ and $\bar{\alpha}$ as orderings of the incident edges around the blue and red vertices respectively. In a combinatorial map we label each edge end, rather than just each edge, so for the edge labelled i we now assign the label i to the end of edge i incident to a blue vertex, and the label $i + n$ to the end of edge i incident to a red vertex. Each edge now has ends labelled i and $i + n$ for some $i \in \{1, 2, \ldots, n\}$, so α, the permutation which swaps the two ends of each edge, should be precisely what we defined it to be. Similarly, σ has been constructed so that the cycles which contain elements of $\{1, 2, \ldots, n\}$ give the cyclic orderings of edge ends around blue vertices, and the cycles which contain elements of $\{n + 1, \ldots, 2n\}$ give the cyclic orderings of edges around the red vertices.

It is very worth noting though that Theorem 5.1 is about *isomorphism classes*. We chose a very specific way to label the edge ends when going from our hypermap to a combinatorial map. We could just as easily have a different convention for assigning the n bit labels to edge ends and then assigning the n additional

labels needed for the combinatorial map. Even if we imposed the condition that in the combinatorial map the end of each edge incident to a blue vertex (in the hypermap) gets labelled with the number of the corresponding bit i in the hypermap, we still have $n!$ ways to "extend" a labelling on a hypermap to one on a combinatorial map. This is why translating our problem into the language of hypermaps will be so useful. For enumerating elements of $\mathcal{R}_g$, we do not care about the choice of labelling. We want to enumerate isomorphism classes, so searching the much smaller space of labelled hypermaps will accomplish the same goal more efficiently.

Before going any further we should discuss how the cycles of φ in a hypermap $(\sigma, \alpha, \varphi)$ relate to the faces of the corresponding 2-vertex-colored combinatorial map. From Theorem 5.1 we know that the cycles of φ should be in bijection with the faces of the corresponding 2-vertex-colored combinatorial map. If n is the number of bits in our hypermap, then $\varphi \in S_n$, but the permutation describing the faces of the corresponding combinatorial map, which we will call φ' would be an element of S_{2n}. The bijection between faces says that $c(\varphi) = c(\varphi')$, so some of the cycles of φ must be shorter than the corresponding cycles in φ'. A reasonable question to ask would be "are they shorter in a predictable way?" Another reasonable question would be "The cycles of φ' do not just correspond to faces, they give an oriented boundary walk around the face. Does something similar happen with φ?"

The answer to both questions is yes. Suppose $(\sigma, \alpha, \varphi)$ is a hypermap corresponding to the 2-vertex-colored combinatorial map $(\sigma', \alpha', \varphi')$ and consider a fixed face

F of the map. The edges that appear in the corresponding cycle of φ are precisely those edges where F is to the left of the walker *while they walk from a blue vertex to a red one* during a walk along the boundary of F in the positive direction (recall we always counterclockwise is the positive direction). This is most clearly and easily explained by drawing a picture and adopting a convention for placement of labels in the picture. See Figure 13. By the convention adopted for placing labels, the label on an edge has been placed *inside* the face that is on the left as one walks along that edge from a blue vertex to a red vertex. Only a small bit of the hypermap is shown, because we only need to verify that $\varphi(1) = 3$ and the argument will apply to any face on any hypermap. Now, recalling that $\varphi = (\alpha \circ \sigma)^{-1} = \sigma^{-1} \circ \alpha^{-1}$ we observe that applying α^{-1} to 1 gives 2, the edge clockwise from 1 around the red vertex. Then applying σ^{-1} to 2 maps 2 to the edge clockwise from 2 around the blue vertex, which is 3 as desired. If we use our convention of label placement and think about what $(\alpha \circ \sigma)^{-1}(e)$ is for any edge e drawn inside F, we see that applying α^{-1} to e takes it to the next edge along the boundary of F, but that this edge will have the label drawn outside of F. Then applying σ^{-1} rotates around the blue vertex on the other end of $\alpha^{-1}(e)$, which again is along the boundary of F, this time with the label drawn inside F.

Since exactly half of the edges along the boundary of the face meet this condition, we can see that each cycle of φ is exactly half the length of the corresponding cycle in φ'. Of particular interest to us, it means that in a hypermap $(\sigma, \alpha, \varphi)$, φ has a fixed point (cycle of length 1) if and only if the corresponding 2-vertex-colored bipartite map has a face of degree 2. In our goal of enumerating elements of the

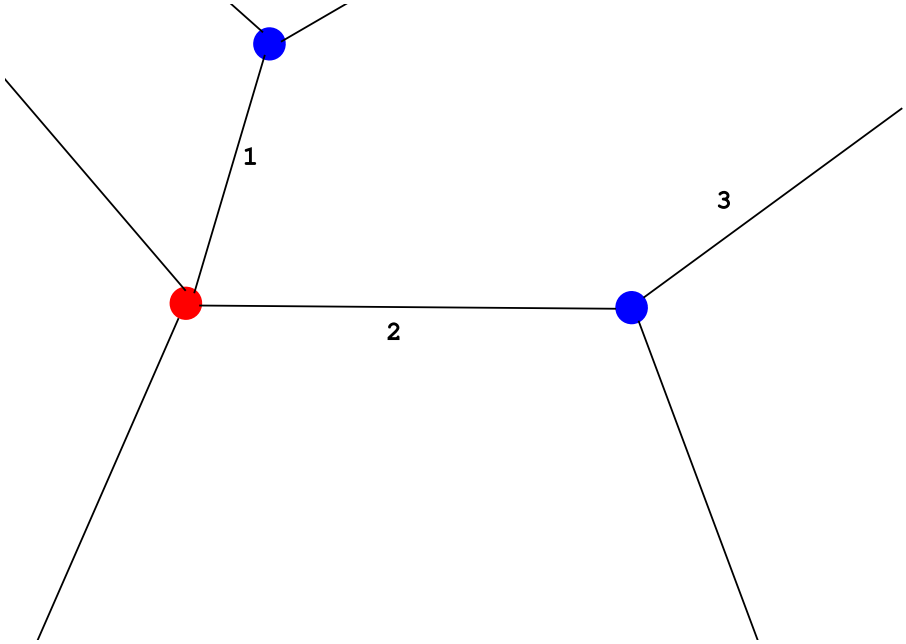

Figure 13: From the label placement, and recalling that σ acts on bits by mapping them to the next bit counterclockwise about and blue vertex and α acts by mapping a bit to the next bit counterclockwise about a red vertex, we see that $(\alpha \circ \sigma)^{-1} = \sigma^{-1} \circ \alpha^{-1}$ maps a bit to the next bit counter-clockwise about the face it is drawn inside.

$\mathcal{R}_g$s, we have already determined $\mathcal{R}_0$, and for $g > 0$ we know that the elements of $\mathcal{R}_g$ have no vertices of degree 2.

Before we go on to the next section, we use the language of hypermaps to restate Corollary 3.2.2 yet again, this time in the language of hypermaps.

Corollary 5.1.1. *Let $R(\eta)$ be a ribbon graph and $g > 0$. $R(\eta) \in \mathcal{R}_g$ if and only if the corresponding combinatorial map M has a bipartite dual, φ has no fixed points and*

$$g_R = \frac{-c(\varphi) + n + c(\sigma) + c(\alpha)}{2} - 1$$

where $(\sigma, \alpha, \varphi)$ is a hypermap corresponding to a 2-vertex-coloring of the dual of M.

Proof. The corollary follows immediately from Corollary 3.2.2 which says $g - g_R + 2 = F$ and Definition 2.14, and translating into the language of hypermaps according to Table 2. □

Now that we have established the necessary background on combinatorial maps and hypermaps, it is time to apply all of this to an algorithm to determine the elements of the $\mathcal{R}_g$s.

6 An Algorithm to Find the Elements of $\mathcal{R}_g$

In this section we use make use of Corollary 5.1.1 to establish conditions on the number of bits and cycle types of σ, α, φ for a hypermap $(\sigma, \alpha, \varphi)$ corresponding to an element of $\mathcal{R}_g$. We then construct an algorithm to enumerate the elements of $\mathcal{R}_g$ and $\mathcal{C}_g$ and use these bounds to restrict the search space to a manageable size.

Lemma 6.1. *Let $R \in \mathcal{R}_g$ and let $(\sigma, \alpha, \varphi)$ be a hypermap corresponding to R (i.e. it corresponds to the 2-vertex-colored dual of the combinatorial map associated to R). Then*

$$c(\sigma) + c(\alpha) = g - g_R + 2$$

Proof. From Corollary 3.2.2 we know that $F = g - g_R + 2$. Translating through Table 2 we have $F = c(\sigma) + c(\alpha)$. □

Lemma 6.2. *Let $(\sigma, \alpha, \varphi)$ be a hypermap with n bits and genus g_R. Let X be the dual of the corresponding 2-vertex-colored bipartite map. If X is a combinatorial map corresponding to an element of $\mathcal{R}_g$ then $n \leq 2g + 2g_R$.*

Proof. From Lemma 6.1 we have

$$c(\sigma) + c(\alpha) = g - g_R + 2$$

Since $g > 0$, Corollary 5.1.1 tells us that φ has no fixed points. Then each cycle of φ has length at least 2. Each of the n-bits appears in precisely one cycle of φ

so

$$c(\varphi) \leq \frac{n}{2}.$$

Then by the genus formula for $(\sigma, \alpha, \varphi)$ (see Definition 5.3) we have

$$g_R = \frac{2 + n - c(\alpha) - c(\sigma) - c(\varphi)}{2}$$

which simplifies to

$$\begin{aligned} c(\alpha) + c(\sigma) + 2g_R &= 2 + n - c(\varphi) \\ c(\alpha) + c(\sigma) + 2g_R &\geq 2 + n - \frac{n}{2} \end{aligned}$$

substituting $c(\alpha) + c(\sigma) = 2 + g - g_R$ we obtain

$$2 + g - g_R + 2g_R \geq 2 + \frac{n}{2}$$

which implies the result

$$2g + 2g_R \geq n$$

□

Lemma 6.2 now gives an upper bound on the size of the symmetric groups we need to search, so we now have a stopping condition for an exhaustive search. For $g = 5$ though, this still entails searching through symmetric groups up to S_{20}, so

we would like to further restrict the search space if possible. A lower bound on E would be nice.

Lemma 6.3. *Let $(\sigma, \alpha, \varphi)$ be a hypermap with n bits and genus g_R. Let X be the dual of the corresponding 2-vertex-colored bipartite map. If X is a combinatorial map corresponding to an element of $\mathcal{R}_g$ then $n \geq g + g_R + 1$.*

Proof. Again we use the genus formula for $(\sigma, \alpha, \varphi)$ and solve for n:

$$g_R = \frac{2 + n - c(\alpha) - c(\sigma) - c(\varphi)}{2}$$

simplifies to

$$n = c(\alpha) + c(\sigma) + c(\varphi) + 2g_R - 2$$

Then substituting Lemma 6.1 and the fact that $c(\varphi) \geq 1$ proves the statement. □

These two bounds are both sharp. Figure 14 shows two elements of $\mathcal{R}_1$ which achieve the two bounds.

Now we have sharp upper and lower bounds on the sizes of the symmetric groups S_n which we will need to search. We would like to restrict our search space within those groups though. Lemma 6.1 gives one restriction, that $c(\sigma)+c(\alpha) = 2+g-g_R$. Corollary 5.1.1 gives another, that φ has no 1-cycles. At this point we would like to make use of two facts about what we are counting: We are counting hypermaps *up to isomorphism* and we are counting hypermaps as a shortcut to counting

Figure 14: On the left is shown a 2-bit hypermap dual to an element of $\mathcal{R}_1$, with the corresponding graph emebedding shown below. In this case $n = 2, g = 1, g_R = 0$. On the right is shown a 4-bit hypermap dual to an element of $\mathcal{R}_1$, with the corresponding graph embedding below. Different line dashings are used to help identify edges since this embedding is nonplanar and must be drawn on the page with crossings. It is hard to see the faces here, but for this hypermap we have $\sigma = (1,2,3,4), \alpha = (1,2,3,4), \varphi = (1,3)(2,4)$ so $n = 4, g_R = 1$ and $g = 1$. These two examples show that the bounds from Lemma 6.3 and Lemma 6.2 are sharp.

bipartite combinatorial maps.

The first fact, that we are counting hypermaps up to isomorphism means we do not want to count different labellings of the same hypermap separately. Recall that two n-bit hypermaps $(\sigma, \alpha, \varphi)$ and $(\sigma', \alpha', \varphi')$ are isomorphic if and only if they differ by conjugation by an element $\rho \in S_n$. I.e if there exists $\rho \in S_n$ such that

$$\rho \circ \sigma \circ \rho^{-1} = \sigma', \quad \rho \circ \alpha \circ \rho^{-1} = \alpha', \quad \rho \circ \varphi \circ \rho^{-1} = \varphi'.$$

Since two permutations in S_n are conjugate if and only if they have the same cycle type there exists at least one $(\sigma', \alpha', \varphi')$ isomorphic to $(\sigma, \alpha, \varphi)$ for every σ' with the same cycle type as σ (and similar for α, φ). Thus within each triple of possible cycle types satisfying Corollary 5.1.1 we can fix one of σ, α, or φ to be a specific element of the desired cycle type.

The point of the second fact, that we are counting hypermaps as a shortcut to counting bipartite maps, is that the choice of 2-vertex-coloring does not matter. For a connected bipartite map, the choice of color for a single vertex determines a proper 2-vertex-coloring (since each vertex must be a different color from its neighbors), but this typically leads to a pair of distinct colorings, as shown in Figure 15, and thus 2 nonisomorphic hypermaps. It is possible for both colorings to give isomorphic hypermaps, as shown in Figure 16, so we will need to handle this carefully, but this gives us the freedom to require a basic relationship between σ and α. We can always require that there be no more blue vertices than red vertices (so $c(\sigma) \leq c(\alpha)$) and when there are equal numbers of vertices in each

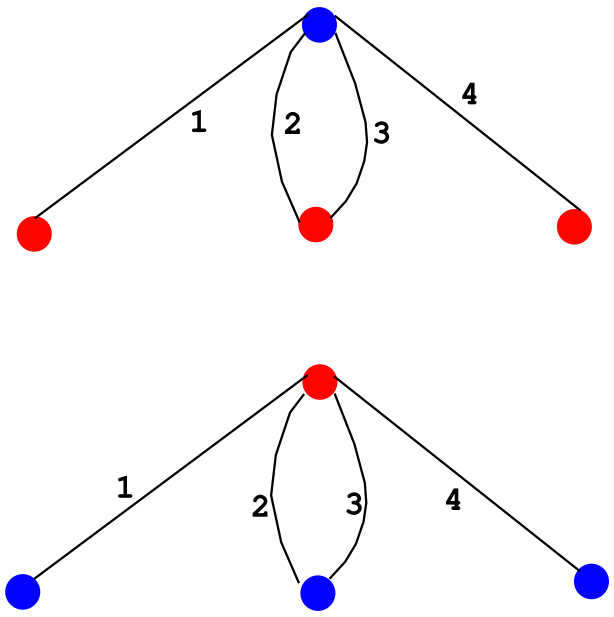

Figure 15: Two hypermaps corresponding to the same underlying bipartite combinatorial map, but with distinct colorings are shown.

Restrictions on n	Restrictions on (C_1, C_2, C_3)	Restrictions on σ, α, φ
$n \geq g + g_R + 1$	$c(C_1) + c(C_2) = 2 + g - g_R$	σ is a fixed element of C_1.
$n \leq 2g + 2g_R$	$g_r = \frac{2+n-c(C_1)-c(C_2)-c(C_3)}{2}$	α or φ varies.
	C_3 has no 1-cycles	
	$c(C_1) \leq c(C_2)$ If $c(C_1) = c(C_2)$, $\#C_1 \geq \#C_2$	

Table 3: A table of all the restrictions we have imposed on the space of permutations triples to search for hypermaps corresponding to elements of $\mathcal{R}_g$ with given ribbon graph genus g_R. Column 1 gives restriction on n, the number of bit. For fixed n, column 2 gives restrictions on the cycle types (C_1, C_2, C_3) to search for hypermaps $(\sigma, \alpha, \varphi)$. Here we use the notation $c(C_i)$ = the number of disjoint cycles in cycle type C_i, and $\#C_i$ = the number of permutations in S_n with cycle type C_i. Column 3 gives any possible restrictions on the values of σ, α, φ to search once the cycle types C_1, C_2, C_3 have been chosen.

color class we require that there not be more permutations with the cycle type of α than there are with the cycle type of σ. When σ and α have the same cycle type we will need to do some isomorphism testing.

Now we are ready to lay out an algorithm for finding all the elements of $\mathcal{R}_g$ for $g > 0$ (recall that $\mathcal{R}_0 = \{\text{the annulus}\}$). To avoid writing the algorithm with basically the entire algorithm nested inside a pair of outer loops, we first define subsets $\mathcal{R}_{g,\hat{g}}$ of $\mathcal{R}_g$ by:

$$\mathcal{R}_{g,\hat{g}} := \{R \in \mathcal{R}_g : g_R = \hat{g}\}.$$

Then, by Corollary 3.2.2 $\mathcal{R}_g$ is the disjoint union of the $\mathcal{R}_{g,\hat{g}}$ for $0 \leq \hat{g} \leq g$.

Next we define subsets $\mathcal{R}_{g,\hat{g},n}$ of $\mathcal{R}_{g,\hat{g}}$ by:

$$\mathcal{R}_{g,\hat{g},n} := \{R \in \mathcal{R}_g : R \text{ has } n \text{ edges}\}.$$

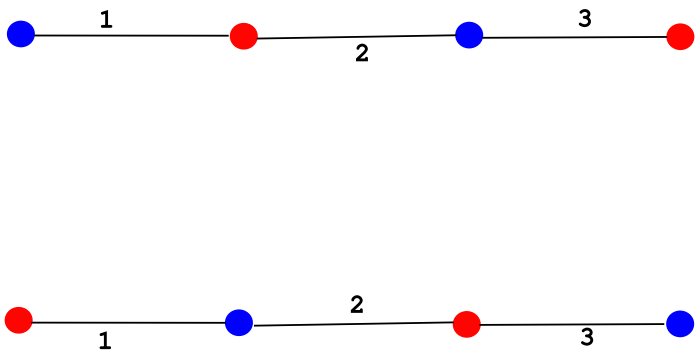

Figure 16: This figure shows two choices of 2-vertex-coloring of the same underlying combinatorial map which give rise to isomorphic hypermaps. The isomorphism can be seen visually as rotation by π about the midpoint of edge 2, or algebraically as conjugation by $\rho = (1, 3)$.

For $n_1 \neq n_2$ we certainly have that $\mathcal{R}_{g,\hat{g},n_1}$ and $\mathcal{R}_{g,\hat{g},n_2}$ are disjoint and by Lemmas 6.3 and 6.2 we have

$$\mathcal{R}_{g,\hat{g}} = \bigcup_{n=g+\hat{g}+1}^{2(g+\hat{g})} \mathcal{R}_{g,\hat{g},n}.$$

Now we give an algorithm to compute the elements of $\mathcal{R}_{g,\hat{g},n}$ for $g > 0$, $0 \leq \hat{g} \leq g$, and $g + \hat{g} + 1 \leq n \leq 2(g + \hat{g})$ and taking the union over valid choices of $\hat{g}$ and n will recover $\mathcal{R}_g$.

In the algorithm, whenever we write $\alpha < \beta$ for two permutations in S_n, the "$<$" is with respect to the lexicographic ordering on permutations. The ordering is defined by: $\alpha < \beta$ if $\alpha(i) < \beta(i)$ for the first integer i such that $\alpha(i) \neq \beta(i)$.

Given enough processing time and storage space the algorithm described above will find a single hypermap representative for each isomorphism class of graph embedding in $\mathcal{R}_g$. Due to the size of the search space, time and space do become concerns, which we address in the next section on computational methods and results. In practice, the number of elements in $\mathcal{R}_g$ grows quite rapidly (see the next section for just how large it becomes) and we are most interested in the sizes of the sets $\mathcal{R}_g, \mathcal{C}_g, \mathcal{L}_g$, and $\mathcal{M}_g$ so we made a change to the last portion of the algorithm. Rather than exhaustively check each hypermap against the others in our list to find a single representative from each isomorphism class we can simply determine which hypermaps $(\sigma, \alpha, \varphi)$ were counted twice and subtract one half this number from the total size of the list. When we compute $\mathcal{C}_g$ from our list we will have to test the underlying graphs for isomorphism anyways, so this will catch any duplicate graphs. Even though graph isomorphism testing is much

Algorithm 1

$r \leftarrow 1$ ▷ r for number of red vertices
while $r \leq \frac{2+g-\hat{g}}{2}$ **do**
    $CC \leftarrow \{(C_1, C_2, C_3)\}$ ▷ C_1 a conjugacy class in S_n with r disjoint cycles, C_2 a conjugacy class in S_n with $2 + g - \hat{g} - r$ disjoint cycles, C_3 has no fixed points, and $c(C_1) + c(C_2) + c(C_3) = 2 - 2\hat{g} + n$
    for $(C_1, C_2, C_3) \in CC$ **do**
        $\psi \leftarrow$ the first element in lex order of the largest of C_1, C_2, and C_3
        $H \leftarrow$ the set of elements of S_n that commute with ψ
        $C \leftarrow$ the smallest of C_1, C_2, C_3
        for $\theta \in C$ **do** $\bar{g} \leftarrow$ the genus of the hypermap determined by ψ and θ
            if $\bar{g}$ is equal to $\hat{g}$ **then** ▷ We only want one hypermap from each isomorphism class, so we pick the one where θ comes first in lex-order isMin $\leftarrow 1$
                for $\rho \in H$ **do**
                    if $\rho^{-1} \circ \theta \circ \rho < \theta$ **then** isMin $\leftarrow 0$
                    end if
                end for
                if isMin is 1 **then** Add the hypermap determined by ψ and θ to $\mathcal{R}_{g,\hat{g},n}$
                end if
            end if
        end for
    end for
end while ▷ At this point we have a representative of each isomorphism class, but may have counted elements of $\mathcal{R}_{g,\hat{g},n}$ twice if changing the coloring gave another element in our list
for $(\sigma, \alpha, \varphi) \in \mathcal{R}_{g,\hat{g},n}$ **do**
    for $(\sigma', \alpha', \varphi') \in \mathcal{R}_{g,\hat{g},n}$ **do**
        if $(\alpha, \sigma, \varphi) \simeq (\sigma', \alpha', \varphi')$ and $(\sigma', \alpha', \varphi') < (\sigma, \alpha, \varphi)$ **then** Remove $(\sigma, \alpha, \varphi)$ from $\mathcal{R}_{g,\hat{g},n}$
        end if
    end for
end for
return $\mathcal{R}_{g,\hat{g},n}$

more computationally difficult than hypermap isomorphism testing, $|\mathcal{R}_g|$ is so much larger than $|\mathcal{C}_g|$ that this turns out to be much faster.

7 Computational Methods and Results

Now that we have established a way to describe the elements of the $\mathcal{R}_g$s and a way to determine the contents of the $\mathcal{C}_g$s, $\mathcal{L}_g$s and $\mathcal{M}_g$s from those elements, we need a method to actually compute the elements of the $\mathcal{R}_g$. The search space we will need to look at consists of conjugacy classes of symmetric groups, and these grow quite large quite quickly, so this will be computationally expensive and will require a great deal of memory. To avoid both unnecessary memory usage and extra computation time from reallocating space to store the hypermaps being computed we would like to establish bounds on the numbers of hypermaps that will need to be stored. Theorem A.10 from the appendix on methods and results in representation theory gives us the following bounds for how many hypermaps we will need to store:

Bounds on the number of hypermaps to be stored: Let $\sigma \in S_n$, let C_2, C_3 be conjugacy classes in S_n, and let $\mathcal{N}(\sigma, C_2, C_3)$ be the number of isomorphism classes of hypermaps $(\sigma, \alpha, \varphi)$ on n bits with $\alpha \in C_2$ and $\varphi \in C_3$. Then from Theorem A.10 in the appendix on representation theory,

$$\frac{|C_2||C_3|}{n!} \sum_{V_i \in \text{Irr } S_n} \frac{\chi_i(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(1)}$$

provides an upper bound for the number of hypermaps we might need to store for given σ, C_2, C_3. Here $Z(\sigma)$ is the set of permutations in S_n that commute with σ and Irr S_n is the set of irreducible representations of S_n (see Definition A.6).

Unfortunately, as was discovered when attempting to put this bound into use, the amount of memory that would be needed to store a list of such size is impractical. Even storing the two permutations needed to define a hypermap in one line form as lists of 8-bit integers to minimize storage space, we would need to preallocate over 1000 GB of memory for the single case where σ is a 17-cycle, C_2 are the permutations in S_{17} which are 17-cycles, and C_3 are permutations in S_{17} consisting of a 4-cycle, a 3-cycle, and 5 2-cycles.

To deal with this problem, we made use of the following facts: From attempts to find the elements of the $\mathcal{R}_g$ before crashing due to lack of memory, we knew that memory space was not a problem until we consider the subset of hypermaps in $\mathcal{R}_5$ where the hypermaps themselves have genus 5. According to Corollary 3.2.2 and Lemmas 6.3 and 6.2 this means that memory trouble only occurs when σ is an n-cycle in S_n for $11 \leq n \leq 20$. After doing some Euler characteristic calculations to determine the allowable cycle types it follows from Theorem A.12 that if $n = 11$ the lower bound from Theorem A.10 differs from the actual number of hypermaps by 8, and that for $n = 13, 17$, and 19 the lower bound is exact. Theorem A.12 only applies when n is prime, so there is no reason to believe that the lower bound will be as close for non-prime n, but pre-allocating space according to the lower bound from Theorem A.10 leaves enough memory free to double the allocated space when the bound is not sharp. This solves the issue of memory usage for our computations.

Compute time was also a concern, since we couldn't find any way around searching whole conjugacy classes of symmetric groups. The procedure described in

Algorithm 1 chooses the smallest possible conjugacy classes to search but they are quite large and there are many of them. Fortunately, the bulk of the process described in Algorithm 1 consists of iterating through the elements in a conjugacy class of a symmetric group and doing the same computations on each element, so it is a natural case for using parallel computing. Distributed computing with separate nodes on each thread of many computers would have been a natural fit, but then our memory concerns return. There are only two high-memory nodes on our compute cluster and while we can afford the memory usage from doubling our array sizes on those machines, we do not *a priori* know how the hypermaps we are looking for are distributed within a conjugacy class. It is very possible that with the process running on n nodes there could be one or more nodes that wouldn't need to story any solutions, while another node might need to store far than $1/n$ of the total number. As a result, one of the smaller machines might need far more than the expected amount of memory. Instead we compromise between memory usage and maximum parallelism by using mulithreaded code on a single high-memory compute node.

We use the Julia programming language, a high-level language which is still extremely fast, and use the OSCAR computational algebra package [26] for some of the group theoretic computations. This package is the most well-developed computational algebra package for Julia right now, but unfortunately it relies on GAP for its group theoretic computations, and a version of GAP that supports multithreading is still in development [11]. For small objects, like centralizer subgroups of fixed permutations we pre-compute the needed objects with OS-

g	$\lvert\mathcal{R}_g\rvert$	$\lvert\mathcal{C}_g\rvert$	$\lvert\mathcal{L}_g\rvert$	$\lvert\mathcal{M}_g\rvert$
0	1	1	1	1
1	3	3	4	5
2	31	17	21	26
3	1831	164	191	217
4	462645	3096	3338	3555
5	255135636	318186	322179	325734

Table 4: Values of $|\mathcal{R}_g|, |\mathcal{C}_g|, |\mathcal{L}_g|$, and $|\mathcal{M}_g|$ for $0 \leq g \leq 5$

CAR/GAP before the parallel portion of the algorithm, but for iterating through conjugacy classes without storing the entire class in memory we needed to write our own iteration algorithm. This algorithm, along with several related iterative algorthms can be found in combinadics.jl in Appendix B. The algorithm is built off of standard constant-amortized-time algorithms for iterating through all k-subsets of $\{1, 2, \ldots, n\}$ and for iterating through all elements of S_n found in [27]. While the function to iterate through elements of a conjugacy class may not see much use, we noticed while working on it that our implementations of the algorithms for iterating through subsets and permutations run significantly faster (less than 1% of the runtime) and use less memory than the implementations in the standard Julia language combinatorics package and could be a valuable contribution to the language's ecosystem of packages.

With a parallel implementation of Algorithm 1 and memory allocation handled, we have a program that can compute the elements of the $\mathcal{R}_g$ and $\mathcal{C}_g$ for $1 \leq g \leq 5$. Then Theorems 2.7 and 3.4 give the sizes of the $\mathcal{L}_g$ and $\mathcal{M}_g$ (recall that $|\mathcal{R}_0| = |\mathcal{C}_0| = |\mathcal{L}_0| = |\mathcal{M}_0| = 1$ from [3]). These values are given in the Table 4:

None of the sequences $\{\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_5\}$, $\{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_5\}$, etc. from Table 4 appear in the Online Encyclopedia of Integer Sequences (OEIS) at this time, [24]. This is a bit disappointing, as it would have been fascinating to see them appear in another sequence and use this to find a way to compute some of these numbers without a computer search, but does provide a new sequence to add to the OEIS.

8 Applications and Directions for Future Work

Since we were only able to compute the sets $\mathcal{R}_g$ and $\mathcal{C}_g$ for $g \leq 5$, a natural extension of the work here would be to compute these values for $g = 6$. Based on the methods used here, this is impractical without tremendous computing resources, as there are triples of conjugacy classes for genus 6 where just storing the hypermaps for a single triple would take up multiple terabytes of memory. Conceivably we could adjust Algorithm 1 to only store graph isomorphism classes, but the growth in compute time would still make this likely implausible (computing $\mathcal{R}_4$ takes about 15 minutes, but computing $\mathcal{R}_5$ takes about a month on our current machine so a reasonable estimate for $\mathcal{R}_6$ would be about 300 years). Instead we see three different avenues to extend this work.

The first direction for future work is to attempt to resolve the difficulties with using the methods of [18], [21], and [22] to determine the sizes of the $\mathcal{R}_g$. This would involve a classification of what hypermaps can appear as a quotient by an automorphism of a hypermap corresponding to an element of $\mathcal{R}_g$, which would be an interesting challenge. As far as we know the methods in these papers have only been used to enumerate combinatorial maps and hypermaps with relatively general criteria, such as all hypermaps with fixed genus and fixed number of bits, or fixed number of bits and fixed number of faces, etc. so such a study could potentially yield useful tools for generalizing those methods to other families of combinatorial maps and hypermaps as well.

The next direction for future work would be to attempt to use and develop tech-

niques in the representation theory of groups and algebras with the goal of directly computing the values of the $|\mathcal{R}_g|$. In the appendix on representation theory we formulated this problem in terms of arithmetic in the group algebras $\mathbb{C}S_n$ (for a precise definition see Definition A.1), presented a formula due to Frobenius A.9 to solve a similar problem, and saw a major obstruction to adapting the formula to our case was that key elements in the adapted formula were not in the center of $\mathbb{C}S_n$. If a suitable sub-algebra could be found where these elements were central, then an analogous formula might be found, in terms of characters of that algebra. Another approach here would be to follow along that taken in Theorem A.12. In that formula we gave the number of isomorphism classes of p-bit hypermaps $(\sigma, \alpha, \varphi)$ when p is a prime, σ is a p-cycle, and the conjugacy classes of α and φ are given by using properties of p-cycles in S_p. In the case of n-bit hypermaps with σ and n-cycle when n is a product of distinct primes, or the square of a prime, similar methods combined with some of the techniques to deal with symmetries from [22] should work to generalize this formula. Such results may also yield new congruency results similar to Corollary A.12.1.

The final direction for additional work is to generalize the computer program developed here to make it potentially useful to other researchers in related fields. The field of Hurwitz theory, which studies maps between Riemann surfaces is very closely connected to the study of hypermaps. Even a moderately detailed exposition of the field is beyond the scope of this dissertation, but here we give a very brief description of a problem in the field (for more information, see [6]). Given positive integers n and k, and a k-tuple $\{C_1, C_2, \ldots, C_k\}$ of conjugacy

classes in S_k, the disconnected Hurwitz number $H^\bullet(C_1, C_2, \ldots, C_k)$ is $\frac{1}{n!}$ times the size of the set

$$\{(c_1, \ldots, c_k) | c_i \in C_i \text{ for all } i \text{ and } c_1 c_2 \ldots c_k = 1\}.$$

As discussed in the appendix on representation theory, this is certainly closely related to the hypermap counting that we do here. In fact a formula for the value of $H^\bullet(C_1, \ldots, C_k)$ (up to a factor of $n!$) is given by Theorem A.9, and its much faster to compute these numbers using that formula than our software. Some questions studied in the field include finding non-character based formulae for these numbers [8] [12], and studying their growth rates [5].

Our software is only written for the case where $k = 3$, but the degeneration formula for Hurwitz numbers allows all Hurwitz numbers to be determined from the case where $k = 3$ [25] so the specific case where our software work is a fundamental case for the field. Since our software computes the actual isomophism classes of the objects contributing to these sums, and they can be visualized as 2-vertex-colored embeddings of bipartite graphs, we hope that it may be useful as a tool for experimentation and visualization here. Lists of the objects may be useful for cases where the number is not too large, and if the software was extended to also render an image of the graph embeddings, then we hope that being able to see a collection of these objects might grant researches in the field additional insights. In addition to adding a visualization component, such an updated version of the software should give the number of labellings for each

isomorphism class, since they are considered distinctly in this field.

Another potential generalization of the computer program could be towards computation of matrix integrals in physics. As with Hurwitz theory, any significant digression into the study of matrix integrals and their connection with enumeration of combinatorial maps is beyond the scope of this dissertation, so we refer the reader to [9], [17], or [36] for an introduction to these connections. We simply observe without proof (see the cited texts, particularly [9]) that some integrals over spaces of Hermitian matrices can reformulated as problems of counting combinatorial maps. Our method of map counting by counting related hypermaps can only be applied when the either the maps of interest or their dual maps are bipartite, but when those conditions are satisfied and the more powerful methods from [18], [21], and [22] are not applicable our program could find use given some updates. For this more general use our program would need to be modified in a few ways. The simplest change would be to generalize some functions involved in memory management to accept arbitrary conjugacy classes of symmetric groups, rather than being specialized to the ones where memory management was a concern in our specific computations. The more substantial change would be to replace the specific conditions we imposed based on our focus on minimal separating embeddings with functionality to accept a range of conditions based on the maps of interest.

References

[1] M.A. Armstrong. *Basic Topology*. Undergraduate Texts in Mathematics. Springer New York, NY, 2010.

[2] Lowell W. Beineke and Robin J.Editors Wilson, editors. *Topics in Topological Graph Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

[3] James Bernhard and J.J.P. Veerman. The topology of surface mediatrices. *Topology and its Applications*, 154(1):54–68, January 2007.

[4] Dmitri Burago, Yuri Burago, and Sergei Ivanov. *A Course in Metric Geometry*, volume 33 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, Rhode Island, June 2001.

[5] Renzo Cavalieri, Paul Johnson, and Hannah Markwig. Wall crossings for double Hurwitz numbers. *Advances in Mathematics*, 228(4):1894–1937, 2011.

[6] Renzo Cavalieri and Eric Miles. *Riemann Surfaces and Algebraic Curves: A First Course in Hurwitz Theory*. London Mathematical Society Student Texts. Cambridge University Press, 2016.

[7] J.R. Edmonds. A Combinatorial Representation For Oriented Polyhedral Surfaces. Master's thesis, University of Maryland, 1960.

[8] Torsten Ekedahl, Sergei Lando, Michael Shapiro, and Alek Vainshtein. On Hurwitz numbers and Hodge integrals. *Comptes Rendus de l'Académie des*

Sciences - Series I - Mathematics, 328(12):1175–1180, 1999.

[9] Bertrand Eynard. *Counting Surfaces*, volume 70 of *Progress in Mathematical Physics*. Springer Basel, Basel, 2016.

[10] Logan S. Fox and J. J. P. Veerman. Equidistant sets on Alexandrov surfaces. *Differential Geometry and its Applications*, 90:102042, 2023.

[11] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.13.1*, 2024.

[12] I. P. Goulden, D. M. Jackson, and R. Vakil. Towards the geometry of double Hurwitz numbers. *Advances in Mathematics*, 198(1):43–92, 2005.

[13] J.L. Gross and T.W. Tucker. *Topological Graph Theory*. Dover books on mathematics. Dover Publications, 2001.

[14] Frank Harary, Geert Prins, and W. T. Tutte. The Number of Plane Trees. *Indagationes Mathematicae (Proceedings)*, 67:319–329, 1964.

[15] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[16] Christine L. Kinsey. *Topology of surfaces*. Undergraduate texts in mathematics. Springer-Verlag, New York, 1993.

[17] S. K. Lando and A. K. Zvonkin. *Graphs on surfaces and their applications*. Number v. 141. 2 in Encyclopaedia of mathematical sciences, Low-dimensional topology. Springer, Berlin ; New York, 2004.

[18] V. A. Liskovets. Enumeration of nonisomorphic planar maps: Nonisomorphic Planar Graphs. *Journal of Graph Theory*, 5(1):115–117, March 1981.

[19] Martin Lorenz. *A Tour of Representation Theory*. American Mathematical Society, 2018.

[20] Mario Ponce and Patricio Santibáñez. On Equidistant Sets and Generalized Conics: The Old and the New. *The American Mathematical Monthly*, 121(1):18, 2014.

[21] Alexander Mednykh and Roman Nedela. Enumeration of unrooted maps of a given genus. *Journal of Combinatorial Theory, Series B*, 96(5):706–729, September 2006.

[22] Alexander Mednykh and Roman Nedela. Enumeration of unrooted hypermaps of a given genus. *Discrete Mathematics*, 310(3):518–526, February 2010.

[23] Bojan Mohar. A Linear Time Algorithm for Embedding Graphs in an Arbitrary Surface. *SIAM Journal on Discrete Mathematics*, 12(1):6–26, January 1999.

[24] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2024. Published electronically at http://oeis.org.

[25] Andrei Okounkov and Rahul Pandharipande. Gromov–Witten theory, Hurwitz theory, and completed cycles. *Annals of Mathematics*, 163(2):517–560, March 2006.

[26] Oscar – open source computer algebra research system, version 0.12.2-dev, 2023.

[27] Frank Ruskey. *Combinatorial Generation*. 2003.

[28] Jean-Pierre Serre. *Linear representations of finite groups*. Number 42 in Graduate texts in mathematics. Springer-Verlag, New York, corr. 5th print edition, 1996.

[29] Richard P. Stanley. *Enumerative combinatorics*. The Wadsworth & Brooks/Cole mathematics series. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, Calif, 1986.

[30] W. T. Tutte. On the enumeration of planar maps. *Bulletin of the American Mathematical Society*, 74(1):64–75, January 1968.

[31] J. J. P. Veerman. Navigating Around Convex Sets. *The American Mathematical Monthly*, 127(6):504–517, July 2020.

[32] J. J. P. Veerman, Mauricio M. Peixoto, André C. Rocha, and Scott Sutherland. On Brillouin Zones:. *Communications in Mathematical Physics*, 212(3):725–744, August 2000.

[33] J.J.P. Veerman and James Bernhard. Minimally separating sets, mediatrices, and Brillouin spaces. *Topology and its Applications*, 153(9):1421–1433, March 2006.

[34] T.R.S Walsh. Hypermaps versus bipartite maps. *Journal of Combinatorial Theory, Series B*, 18(2):155–163, April 1975.

[35] Kunizô Yoneyama. Theory of continuous set of points (not finished). *Tohoku Mathematical Journal, First Series*, 12:43–158, 1917.

[36] A. Zvonkin. Matrix integrals and map enumeration: An accessible introduction. *Mathematical and Computer Modelling*, 26(8-10):281–304, October 1997.

Appendix A Representation Theory and Hypermap Enumeration

At this point we have nice descriptions of the elements in the $\mathcal{R}_g$s, a procedure to find the elements of the $\mathcal{C}_g$s from the $\mathcal{R}_g$s, and a way to determine the elements of the $\mathcal{L}_g$s and $\mathcal{M}_g$s from the $\mathcal{C}_g$s. We still need to actually compute the elements of the $\mathcal{R}_g$. Once we start doing the computing memory is going to be a big concern. The $\mathcal{R}_g$ get very large just for $g = 5$ and storing all of the embeddings takes hundreds of gigabytes of memory. This leads to a memory allocation problem. We will not know ahead of time how large these sets are, so we cannot preallocate space to store them. Without memory concerns, the natural choice would be to use *dynamically sized data structures*. Informally speaking, these are lists that hold a certain amount of data, and if more space is needed the computer finds a spot that can hold about twice as much data, copies the list there, and then can add entries into this new block of space. If that space fills up, it looks for a new bigger location and the process repeats. After writing a program to find $|\mathcal{R}_g|$ and $|\mathcal{C}_g|$ for $g \leq 4$ and trying to run it for $g = 5$, we discovered that even the highest memory node on our university compute cluster does not have enough memory space to follow this procedure that involves repeatedly looking for bigger blocks of memory.

The alternative to dynamic memory allocation is called *static allocation*. Speaking informally again, the idea here is to tell the computer ahead of time exactly how long our list will be and what kinds of objects will be in it. Then the computer can know exactly how much space to reserve and we avoid the 'space-doubling' issue.

The difficulty here is that it requires us to know how large the $|\mathcal{R}_g|$ will be in advance. As mentioned in Chapter 5, existing techniques for map and hypermap enumeration do not translate nicely to our problem so this information is not known ahead of time. The good news is, to avoid the 'space doubling' problem, we just need an upper bound on the length of our list which is small enough to fit in memory. We can compute such a bound using a little bit of the representation theory of finite groups.

Throughout this section when we consider whether a triple of permutations $(\sigma, \alpha, \varphi) \in S_n$ forms a hypermap, we will generally omit any consideration for whether the action of $\langle \sigma, \alpha, \varphi \rangle$ on $\{1, \ldots, n\}$ is transitive and only consider whether $\varphi \circ \alpha \circ \sigma = ()$. In any results depending on the assumption that this action is transitive, we will make note of being "under the transitivity hypothesis". Our primary motivation in this appendix is to establish bounds for memory usage and in all of the cases where memory scarcity is a concern, σ is an n-cycle, which guarantees that the $\langle \sigma, \alpha, \varphi \rangle$ acts transitively. Making this assumption will simplify both exposition and calculuations. Also, we replace the function composition notation for permutation multipilcation with the more standard group theoretic notation where $\alpha \circ \sigma$ becomes $\sigma\alpha$. There will be a great deal of formal multiplication of elements of S_n, but few concrete examples with cycle notation and we believe the more compact notation will be simpler to read.

We have assumed that the reader is familiar with some elementary group theory so far, but here we will introduce some basic definitions and results in the representation theory of finite groups, leading up to a formula we can adapt into an

approximation of $|\mathcal{R}_g|$. For more details on this material and proofs of results, see [28] and [19].

First we state the following definition from [19]

Definition A.1. Given a field k, a *k-algebra* is a ring A (with unit 1) together with a ring homomorphism $i : k \to A$ such that $i(k)$ is in the center of A (the center, $Z(A) := \{a \in A : ax = xa \forall x \in A\}$).

It is often helpful to think about a k-algebra A as a k-vector space equipped with a compatible multiplication operation. Most of what follows holds for arbitrary fields k, but we will only consider the case where $k = \mathbb{C}$. Before introducing the specific type of $\mathbb{C}$-algebra that will be of most interest to us, we will introduce a more familiar type of example which will also be of relevance to us:

Definition A.2. Given a $\mathbb{C}$-vector space V, the *endomorphism algebra of V*, denoted $\mathrm{End}(V)$ is defined as follows: As a set, $\mathrm{End}(V)$ is the set of all $\mathbb{C}$-linear transformations of V. The addition operation is given pointwise: For $S, T \in \mathrm{End}(V)$, $S + T$ is defined by:

$$(S + T)(v) = S(v) + T(v).$$

The multiplication operation is given by composition: $S \cdot T$ is defined by

$$(S \cdot T)(v) = S(T(v)).$$

The homomorphism $i : \mathbb{C} \to \mathrm{End}(V)$ is the map $z \mapsto zI$.

The $\mathbb{C}$-algebras that we will be particularly concerned with are called *group algebras*:

Definition A.3. The *group algebra* of a group G, denoted $\mathbb{C}G$ is the $\mathbb{C}$-algebra constructed as follows: As a $\mathbb{C}$-vector space, $\mathbb{C}G$ is the $\mathbb{C}$-vector space of formal $\mathbb{C}$-linear combinations of the elements of G. The multiplication operation is given as follows:

$$\left(\sum_{g\in G} a_g g\right)\left(\sum_{h\in G} b_h h\right) = \sum_{g,h\in G} a_g b_h (gh) = \sum_{x\in G}\left(\sum_{\substack{g,h\in G\\ gh=x}} a_g b_h\right) x.$$

We will take the elements of G (with Id denoting the identity) as our standard basis for $\mathbb{C}G$ as a $\mathbb{C}$-vector space.

Before going any further, let us motivate this definition through a quick example of how it relates to our goal of counting hypermaps. Suppose we fix an element $\sigma \in S_n$, and two conjugacy classes C_1, C_2 of S_n and want to know: How many choices of $(\alpha, \varphi) \in C_1 \times C_2$ result in $(\sigma, \alpha, \varphi)$ being a hypermap? Knowing the answer to this question would not tell us how many isomorphism classes there are with $\alpha \in C_1$ and $\varphi \in C_2$, since there may be distinct (α_1, φ_1) and (α_2, φ_2) such that $(\sigma, \alpha_1, \varphi_1)$ and $(\sigma, \alpha_2, \varphi_2)$ are isomorphic, but it would give us an upper bound since each isomorphism class would be counted at least once. The following lemma expresses the solution to this problem nicely in terms of arithmetic in $\mathbb{C}S_n$.

Lemma A.1. *Let σ, C_1, C_2 be as defined above. Under the transitivity hypothesis, the number of ways to choose $(\alpha, \varphi) \in C_1 \times C_2$ such that $(\sigma, \alpha, \varphi)$ is a hypermap*

is the coefficient of the identity in the following element of $\mathbb{C}S_n$*:*

$$\sigma \left(\sum_{\alpha \in C_1} \alpha \right) \left(\sum_{\varphi \in C_2} \varphi \right).$$

Proof. Starting from the expression in the lemma:

$$\begin{aligned} \sigma \left(\sum_{\alpha \in C_2} \alpha \right) \left(\sum_{\varphi \in C_3} \varphi \right) &= \sum_{\alpha \in C_2, \varphi \in C_3} \sigma \alpha \varphi \\ &= \sum_{\rho \in S_n} \left(\sum_{\substack{\alpha \in C_2, \varphi \in C_3 \\ \sigma\alpha\varphi = \rho}} 1 \right) \rho \end{aligned}$$

Then we observe that in the final expression, the coefficient on any $\rho \in S_n$ is the number of ways to pick $\alpha \in C_2, \varphi \in C_3$ such that $\sigma\alpha\varphi = \rho$. Setting $\rho = ()$ we have the desired result. □

Here's an example to make this more concrete:

Example: We will look at elments of S_4 and take $\sigma = (1\ 2\ 3\ 4)$, and let us consider hypermaps $(\sigma, \alpha, \varphi)$ where α is another 4-cycle and φ is a pair of 2-cycles. So

$$C_2 = \{(1\ 2\ 3\ 4), (1\ 2\ 4\ 3), (1\ 3\ 2\ 4), (1\ 3\ 4\ 2), (1\ 4\ 2\ 3), (1\ 4\ 3\ 2)\}$$

and

$$C_3 = \{(1\ 2)(3\ 4), (1\ 3)(2\ 4), (1\ 4)(2\ 3)\}$$

Then

$$\sigma\left(\sum_{\alpha\in C_2}\alpha\right)\left(\sum_{\varphi\in C_3}\varphi)\right)=\sigma\left(\sum_{\alpha,\varphi\in C_2\times C_3}\alpha\varphi\right)$$

multiplying out $\alpha\varphi$ for each pair (α,φ) and writing their sum in $\mathbb{C}S_4$ we have

$$\begin{aligned}
&=\sigma\big(2(1\ 2)+2(1\ 3)+2(1\ 4)+2(2\ 3)+2(2\ 4)+2(3\ 4)\\
&+(1\ 2\ 3\ 4)+(1\ 2\ 4\ 3)+(1\ 3\ 2\ 4)+(1\ 3\ 4\ 2)\\
&+(1\ 4\ 2\ 3)+(1\ 4\ 3\ 2)\big)\\
&=1()+2(1\ 3\ 4)+2(1\ 4)(2\ 3)+2(2\ 3\ 4)+2(1\ 2\ 4)\\
&+2(1\ 2)(3\ 4)+2(1\ 2\ 3)+(1\ 3)(2\ 4)+(1\ 3\ 2)\\
&+(1\ 4\ 2)+(1\ 4\ 3)+(2\ 4\ 3)
\end{aligned}$$

The coefficient on the identity permutation here is 1, showing that there is only a single pair (α,φ) in the specified conjugacy classes such that $\sigma\alpha\varphi=()$.

Ideally, we would have an explicit formula for the number of isomorphism classes of hypermaps (σ,α,φ) where σ,α, and φ have specified cycle-types, but it will turn out that outside of special cases we will need to settle for a formula for bounds of the type described above. Once we have such a formula, we will adapt it to a formula for the number of isomorphism classes of hypermaps in some very special cases and see the obstruction to finding a general formula this way. We need to introduce some more machinery first though. The method will essentially be representing group elements as invertible matrices.

Definition A.4. A *representation* ρ of a group G is a vector space V together with a group homomorphism $\rho : G \to \mathrm{GL}(V)$ (where $\mathrm{GL}(V)$ is the group of invertible linear transformations of V). Following common convention, $\rho : G \to \mathrm{GL}(V)$ will be denoted simply as V unless discussing the specific transformation $\rho(g)$ for some $g \in G$.

Much of what follows holds for vector spaces over any field, but from now on we assume that V is always a $\mathbb{C}$-vector space and G is a finite group. We are also going to adopt some common notational conventions: We will commonly denote a representation $\rho : G \to \mathrm{GL}(V)$ by simply V, and for any $v \in V, g \in G$ we will denote $\rho(g)(v)$ by gv.

Definition A.5. Given a group G and two representations V_1, V_2 of G, a *morphism of G-representations* $f : V_1 \to V_2$ is a linear transformation $f : V_1 \to V_2$ such that for all $g \in G, v \in V_1$ the following diagram commutes:

$$\begin{array}{ccc} V_1 & \xrightarrow{\rho_1(g)} & V_1 \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle f} \\ V_2 & \xrightarrow[\rho_2(g)]{} & V_2 \end{array}$$

Just as we look at subspaces of vector spaces and subgroups of groups, subobjects of representations are of interest.

Definition A.6. We say that U is a *subrepresentation* of a G-representation V if U is a subspace of V, and $gU = U$ for all $g \in G$. We say a G-representation V is *irreducible* if the only subrepresentations of V are $\{0\}$ and V.

Theorem A.2. *[28] Every representation is a direct sum of irreducible representations.*

The following results about irreducible representations will be particularly important to us. For proofs, see [28]

Lemma A.3 (Schur's Lemma)**.** *[28] Let V_1, V_2 be irreducible representations of a group G. If V_1 and V_2 are not isomorphic than any morphism $f : V_1 \to V_2$ is the zero map. Any morphism $f : V_1 \to V_1$ is a scalar multiple of the identity.*

Theorem A.4. *[28] The number of irreducible representations of G (up to isomorphism) is equal to the number of conjugacy classes of G.*

Finally, before relating representations to the group algebras we are interested in, we give one more definition that is going to be very important to us:

Definition A.7. The *character* of a G-representation $\rho : G \to \mathrm{GL}(V)$ is the function $\chi_\rho : G \to \mathbb{C}$ defined by:

$$\chi_\rho(g) = \mathrm{Trace}(\rho(g)).$$

Characters of representations are going to play a very important role in the upper bound formula we are working towards. Characters of representations have the

following very nice properties:

Lemma A.5. *[28] Let $\rho : G \to GL(V)$ be a G-representation. χ_ρ has the following properties:*

a $\chi_\rho(1) = dim(V)$ *(because $\rho(Id)$ is the identity matrix).*

b $\chi_\rho(g^{-1}) = \bar{\chi_\rho}(g)$ *(where $\bar{x}$ denotes the complex conjugate of x) for all $g \in G$ (because for finite groups all eigenvalues of $\rho(g)$ are lie on the unit circle and the eigenvalues of $\rho(g^{-1})$ are the complex conjugates of those of $\rho(g)$).*

c $\chi_\rho(g) = \chi_\rho(hgh^{-1})$ *for all $g, h \in G$ (because the trace of a matrix is invariant under conjugation)*

Of particular note to us is that property (c) of the lemma says that the trace of a representation is constant within any conjugacy class of G. Since characters are constant within a conjugacy class, it is common to abuse notation and write $\chi(C)$ (where χ is a character and C is a conjugacy class) to denote the value of χ evaluated at any element of C.

Noting that $\mathbb{C}G$ is a $\mathbb{C}$-vector space, there is a natural representation of G acting on $\mathbb{C}G$, where we take the elements of G as a basis for $\mathbb{C}G$. Since group elements act on themselves, $g \in G$ acts on the basis elements by left multiplication. This is called the *left-regular representation of G*. It is particulary easy to compute the character of this representation.

Lemma A.6. *Let χ_G be the character of the left-regular representation of G.*

Then

$$\chi_G(g) = \begin{cases} |G| & \text{if } g = Id \\ 0 & \text{if } g \neq Id \end{cases}$$

This follows from the observation that with respect to the standard basis on $\mathbb{C}G$, $\chi_G(g)$ is the number of elements $h \in G$ such that $gh = h$. Since the trace is a linear operator, we can define the trace of any element of $\mathbb{C}G$ as follows:

Definition A.8. The function Trace : $\mathbb{C}G \to \mathbb{C}$ is the linear function defined by $\text{Trace}(g) = \chi_G(g)$ and extending onto $\mathbb{C}G$ by linearity.

The following result ties together the irreducible representations of a group G and the group-algebra $\mathbb{C}G$.

Theorem A.7. *[28] Let Irr $G = \{V_1, V_2, \ldots, V_m\}$ be the set whose elements consist of a single representative from each isomorphism of class of irreducible representation of G. Then there is a cannoical isomorphism*

$$\tilde{\rho} : \mathbb{C}G \to \bigoplus_{V_i \in \, Irr\, G} End(V_i)$$

The proof of the theorem is found in [28] but we note that we can describe this isomorphism quite nicely. Letting $\rho_i : G \to \text{GL}(V_i)$ be the group homomorphism associated with the representation V_i, the isomorphism is given by:

$$g \mapsto (\rho_1(g), \rho_2(g), \ldots, \rho_m(g))$$

and extending onto all of $\mathbb{C}G$ by linearity. The following result builds on this and gives even more information that we will use in our proof of the next theorem:

Lemma A.8. *[28] Let $\tilde{\rho}$ be the canonical isomorphism between $\mathbb{C}G$ and $\bigoplus_{V_i \in \ Irr\, G} End(V_i)$ and let $\tilde{\rho}_i$ be the projection of $\tilde{\rho}$ onto $End(V_i)$. $\tilde{\rho}_i$ maps the center of $\mathbb{C}G$ into the set of scalar multiples of the identity on V_i and defines a $\mathbb{C}$-algebra homomorphism $\omega_i : Center(\mathbb{C}G) \to \mathbb{C}$.*

At this point, we have enough machinery to state the following formula due to Frobenius:

Theorem A.9. *Let G be a finite group and let $C_1, \ldots, C_k$ be arbitrary conjugacy classes in G. Define:*

$$N(C_1, \ldots, C_k) := |\{(g_1, \ldots, g_k) \in C_1 \times \ldots \times C_k : g_1 g_2 \ldots g_k = \ Id\}|$$

Then

$$N(C_1, \ldots, C_k) = \frac{|C_1| \ldots |C_k|}{|G|} \sum_{V_i \in \ Irr\, G} \frac{\chi_i(C_1) \ldots \chi_i(C_k)}{\chi_i(Id)^{k-2}}$$

The formula is stated in an exercise in[19], so we provide a proof here:

Proof. First we define the following family of elements in $\mathbb{C}G$: For any conjugacy class C of G, define

$$e_C = \sum_{g \in C} g.$$

The plan of the proof is to compute the trace of $e_{C_1} e_{C_2} \ldots e_{C_k}$ in two separate ways.

First we observe the following consequence of Lemma A.6. For any element $\sum_{g\in G} a_g g \in \mathbb{C}G$,

$$\text{Trace}\left(\sum_{g\in G} a_g g\right) = a_{\text{Id}}|G|.$$

As a consequence, $N(C_1,\ldots,C_k) = \frac{\text{Trace}(e_{C_1}e_{C_2}\ldots e_{C_k})}{|G|}$.

Next we observe that e_C commutes with all elements of $\mathbb{C}G$: Since the elements of G generate $\mathbb{C}G$, it is enough to show that e_C commutes with h for all $h \in G$. Take $h \in G$. Then

$$he_C = h\sum_{g\in C} g$$

and since C is a conjugacy class we have: $\sum_{g\in C} g = \sum_{g\in C} h^{-1}gh$, giving

$$\begin{aligned} he_C &= h\sum_{g\in C} h^{-1}gh \\ &= \sum_{g\in C} gh \\ &= e_C h \end{aligned}$$

Now, since e_C is in the center of $\mathbb{C}G$, by Lemma A.8 e_C acts on each V_i as scalar multiplication by some constant. Then for each $C_j \in \{C_1,\ldots,C_k\}$ we can compute the trace of e_{C_j} on any $V_i \in$ Irr G by

$$\text{Tr}_{V_i}(e_{C_j}) = c_{i,j}\cdot\dim(V_i) = c_{i,j}\chi_{V_i}(\text{Id})$$

for some constant $c_{i,j}$. Now we compute the $c_{i,j}$. Let χ_i be the character associated to V_i for each V_i in Irr G. For any $g \in G$, $\chi_i(g)$ is the trace of g on V_i, and since χ_i is constant on each conjugacy class, the trace of e_{C_j} acting on V_i is:

$$c_{i,j}\chi_i(\mathrm{Id})\mathrm{Tr}_{V_i}(e_{C_j}) = \sum_{g \in C_j} \chi_i(g) = \chi_i(C_j)|C_j|$$

and we obtain

$$\begin{aligned} c_{i,j}\chi_i(\mathrm{Id}) &= \mathrm{Tr}_{V_i}(e_{C_j}) \\ c_{i,j}\chi_i(\mathrm{Id}) &= \chi_i(C_j)|C_j| \\ c_{i,j} &= \frac{\chi_i(C_j)|C_j|}{\chi_i(\mathrm{Id})} \end{aligned}$$

Using the fact that each e_C acts on each V_i as a scalar multiple of the identity, $e_{C_1} \dots e_{C_k}$ acts on each V_i as $c_{i,1}c_{i,2} \dots c_{i,k}$ times the identity. Then $e_{C_1} \dots e_{C_k}$ also acts as scalar multiplication by $c_{i,1} \dots c_{i,k}$ on $\mathrm{End}(V_i)$, which has dimension $\dim(V_i)^2 = \chi_i(1)^2$ (since $\mathrm{End}(V_i)$ is the space of square matrices acting on V_i) and we have that the trace of $e_{C_1} \dots e_{C_k}$ acting on $\mathrm{End}(V_i)$ is:

$$\begin{aligned} \mathrm{Tr}_{\mathrm{End}(V_i)}(e_{C_1} \dots e_{C_k}) &= c_{i,1} \dots c_{i,k}\chi_i(\mathrm{Id})^2 \\ &= \left(\prod_{j=1}^{k} \frac{\chi_i(C_j)|C_j|}{\chi_i(\mathrm{Id})} \right) \chi_{\mathrm{Id}}(1)^2 \\ &= \frac{\prod_{j=1}^{k} \chi_i(C_j)|C_j|}{\chi_i(\mathrm{Id})^{k-2}} \end{aligned}$$

Since $\mathbb{C}G$ is the direct sum of the $\mathrm{End}(V_i)$, the trace of $e_{C_1} \dots e_{C_k}$ is the sum of

the traces over each $\mathrm{End}(V_i)$ and we have

$$\begin{aligned}\mathrm{Tr}(e_{C_1}\dots e_{C_k}) &= \sum_{V_i\in \mathrm{Irr}\ G} \frac{\prod_{j=1}^k \chi_i(C_j)|C_j|}{\chi_i(\mathrm{Id})^{k-2}} \\ &= \left(\prod_{j=1}^k |C_j|\right) \sum_{V_i\in \mathrm{Irr}\ G} \frac{\prod_{j=1}^k \chi_i(C_j)}{\chi_i(\mathrm{Id})^{k-2}}\end{aligned}$$

Since $N(C_1,\dots,C_k) = \frac{\mathrm{Tr}(e_{C_1}\dots e_{C_k})}{|G|}$ the theorem follows. □

In the case where $G = S_n$ and $k = 3$, this looks quite similar the upper bound for hypermaps (σ,α,φ) where σ is a given permutation and α,φ are required to be elements of specified conjugacy classes. The key difference is that in the formula due to Frobenius, we have dropped the requirement that σ be a given permutation and simply specify its conjugacy class. We modify the formula to obtain the following bounds:

Theorem A.10. *Let $\sigma \in S_n$, let C_2, C_3 be conjugacy classes in S_n and let $\mathcal{N}(\sigma, C_2, C_3)$ denote the number of isomorphism classes of hypermaps (σ,α,φ) such that $\alpha \in C_2$ and $\varphi \in C_3$. Under the transitivity hypothesis we have*

$$\frac{|C_2||C_3|}{|Z(\sigma)|\cdot n!} \sum_{V_i\in Irr\ S_n} \frac{\chi_i(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(Id)} \le \mathcal{N}(\sigma,C_2,C_3) \le \frac{|C_2||C_3|}{n!} \sum_{V_i\in Irr\ S_n} \frac{\chi_i(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(Id)}$$

where $Z(\sigma) = \{\rho \in S_n | \rho\sigma = \sigma\rho\}$.

Proof. The upper bound is simply Frobenius formula $N(C_1,C_2,C_3)$ in S_n (where C_1 is the conjugacy class of σ) divided by $|C_1|$. $N(C_1,C_2,C_3)$ counts the number of

labelled hypermaps $(\sigma', \alpha, \varphi)$ where $\sigma' \in C_1, \alpha \in C_2$, and $\varphi \in C_3$. Since choice of labelling does not affect the number of unlabelled maps, each $\sigma' \in C_1$ contributes an equal number of labelled hypermaps to the total and we reduce this number by a factor of $|C_1|$ when we fix $\sigma' = \sigma$. As noted earlier in this section, fixing σ does not ensure that we have only counted each isomorphism class of hypermap once, so this gives an upper bound on $\mathcal{N}(\sigma, C_2, C_3)$.

For the lower bound, we consider a hypermap $(\sigma, \alpha, \varphi)$ and ask how many pairs $(\alpha', \varphi') \in C_2 \times C_3$ there could be such that $(\sigma, \alpha, \varphi)$ is isomorphic to $(\sigma, \alpha', \varphi')$. Recalling that $(\sigma, \alpha, \varphi)$ is isomorphic to $(\sigma, \alpha', \varphi')$ if there exists $\rho \in S_n$ such that

$$\rho\sigma\rho^{-1} = \sigma', \quad \rho\alpha\rho^{-1} = \alpha', \quad \rho\varphi\rho^{-1} = \varphi'$$

we can see that the only isomorphisms from $(\sigma, \alpha, \varphi)$ to another hypermap of the form $(\sigma, \alpha', \varphi')$ are the $\rho \in Z(\sigma)$. Therefore each isomorphism class of hypermap contributes at most $|Z(\sigma)|$ to the sum and dividing by $|Z(\sigma)|$ gives a lower bound on $\mathcal{N}(\sigma, C_2, C_3)$. It also shows that when we let $Z(\sigma)$ act by conjugation on the set of hypermaps $(\sigma, \alpha, \varphi)$ for fixed σ, $\alpha \in C_2, \varphi \in C_3$, then the $\mathcal{N}(\sigma, C_2, C_3)$ is the number of orbits in this action, since there is an isomorphism between $(\sigma, \alpha, \varphi)$ and $(\sigma, \alpha', \varphi')$ if and only if they are conjugate by some $\rho \in Z(\sigma)$. □

Unfortunately, this lower bound is not typically sharp because there may be hypermaps $(\sigma, \alpha, \varphi)$ and elements $\rho \in S_n$ such that

$$\rho\sigma\rho^{-1} = \sigma, \quad \rho\alpha\rho^{-1} = \alpha, \quad \rho\varphi\rho^{-1} = \varphi.$$

For example: Consider the 6-bit hypermap $(\sigma, \alpha, \varphi)$ where

$$\sigma = (1\ 2\ 3\ 4\ 5\ 6), \quad \alpha = (1\ 5\ 6\ 4\ 2\ 3), \quad \varphi = (1\ 2\ 3)(4\ 5\ 6).$$

Letting $\rho = (1\ 4)(2\ 5)(3\ 6)$ we observe that

$$\rho^{-1}\sigma\rho = \sigma, \quad \rho^{-1}\alpha\rho = \alpha, \quad \rho^{-1}\varphi\rho = \varphi$$

so there are fewer than $|Z(\sigma)|$ hypermaps isomorphic to $(\sigma, \alpha, \varphi)$ and the lower bound in Theorem A.10 is an underestimate for the number of isomorphism classes of hypermaps where $(\sigma, \alpha', \varphi')$ where α' is a 6-cycle and φ' is the product of 2 disjoint 3-cycles. Note that by Corollary 5.1.1 $(\sigma, \alpha, \varphi)$ and all other hypermaps with the same cycle-types corresponds to elements of $\mathcal{R}_2$.

We would have liked to recast our problem of counting isomorphism classes of hypermaps with fixed cycle-types into a problem of arithmetic in $\mathbb{C}S_n$ and found an analog of Frobenius' formula to solve that problem. It turns out not to be too difficult to restate our problem as an arithmetic problem in $\mathbb{C}S_n$, but the resulting problem differs in a critical way from the problem solved by Frobenius' formula which presents a serious obstacle. Here we set up that problem and see what makes it more challenging.

Theorem A.11. *Let σ, C_2, C_3, and $\mathcal{N}(\sigma, C_2, C_3)$ be as defined in Theorem A.10. Under the transitivity hypothesis, $\mathcal{N}(\sigma, C_2, C_3)$ is equal to the coefficient of the*

identity in

$$\frac{1}{|Z(\sigma)|}\sigma\left(\sum_{i=1}^{j}\sum_{\rho\in Z(\sigma)}\rho^{-1}\alpha_i\rho\right)\left(\sum_{i=1}^{k}\sum_{\rho\in Z(\sigma)}\rho^{-1}\varphi_i\rho\right)$$

where $\{\alpha_1,\ldots,\alpha_j\}$ *are a set of representatives for the orbits of* C_2 *under the conjugation action by* $Z(\sigma)$ *and* $\{\varphi_1,\ldots,\varphi_k\}$ *are a set of representatives for the orbits of* C_3 *under the conjugation action by* $Z(\sigma)$.

Proof. We define $H=\{(\sigma,\alpha,\varphi)|\alpha\in C_2,\varphi\in C_3,\sigma\alpha\varphi=()\}$, the set of labelled hypermaps with first element σ, and with α and φ having the prescribed cycle types. As in Lemma A.1, $|H|$ is given by the coeffcient of the identity in the following product:

$$\sigma\left(\sum_{\alpha\in C_2}\alpha\right)\left(\sum_{\varphi\in C_3}\varphi\right).$$

Since the orbits of H under the conjugation action by $Z(\sigma)$ correspond to the isomorphism classes of hypermaps (σ,α,φ) with $\alpha\in C_2,\varphi\in C_3$, we found a lower bound for this coefficient in Theorem A.10 by dividing the entire result by $|Z(\sigma)|$, the maximum size of any one orbit. The idea here is to add some terms to the product which will cause each isomorphism class of hypermap to contribute $|Z(\sigma)|$ to the sum, rather contributing the size of its orbit.

Fix φ_ℓ as one of the $\{\varphi_1,\ldots,\varphi_k\}$ and consider the coefficient of the identity in the product

$$\sigma\left(\sum_{i=1}^{j}\sum_{\rho\in Z(\sigma)}\rho^{-1}\alpha_i\rho\right)\sum_{\rho\in Z(\sigma)}\rho^{-1}\varphi_\ell\rho.$$

We observe that we can choose the representative of each orbit of C_2 under the

$Z(\sigma)$ action freely, so within each orbit, if there is an element α which makes $(\sigma, \alpha, \varphi_\ell)$ a hypermap, we will choose that α as the representative of its orbit to simplify some notation.

We also observe that for any fixed σ, φ, there is a unique element $\alpha \in S_n$ (not necessarily in C_2) such that the equation $\sigma\alpha\varphi = 1$ holds. Therefore, for each $i \in \{1, 2, \ldots, j\}$, if $\sigma\alpha_j\varphi_\ell = ()$, then taking any $\rho \in Z(\sigma)$ we have

$$\begin{aligned}\sigma(\rho^{-1}\alpha_j\rho)(\rho^{-1}\varphi_\ell\rho) &= \sigma\rho^{-1}\alpha_j\varphi_\ell\rho \\ &= \rho^{-1}(\sigma\alpha_j\varphi_\ell)\rho \\ &= ()\end{aligned}$$

Thus, if $(\sigma, \alpha_i, \varphi_\ell)$ is a hypermap, then the coefficient of the identity in

$$\sigma\left(\sum_{\rho\in Z(\sigma)} \rho^{-1}\alpha_i\rho\right)\left(\sum_{\rho\in Z(\sigma)} \rho^{-1}\alpha_\ell\rho\right)$$

is either 0 or $|Z(\sigma)|$ (since there are $|Z(\sigma)|$ terms in the rightmost summation). Furthermore, since that coefficient can be non-zero for at most one $\alpha_i \in \{\alpha_1, \ldots, \alpha_j\}$ we conclude that for each φ_ℓ the coefficient of the identity in

$$\sigma\left(\sum_{i=1}^{j}\sum_{\rho\in Z(\sigma)} \rho^{-1}\alpha_i\rho\right)\left(\sum_{\rho\in Z(\sigma)} \rho^{-1}\varphi_\ell\rho\right)$$

is $|Z(\sigma)|$ if there is an $\alpha \in C_2$ such that $(\sigma, \alpha, \varphi_\ell)$ is a hypermap, and 0 otherwise. This reasoning holds for each $\varphi_\ell \in \{\varphi_1, \ldots, \varphi_k\}$ so the coefficient of the identity

in

$$\sigma \left(\sum_{i=1}^{j} \sum_{\rho \in Z(\sigma)} \rho^{-1} \alpha_i \rho \right) \left(\sum_{i=1}^{k} \sum_{\rho \in Z(\sigma)} \rho^{-1} \varphi_i \rho \right)$$

is exactly $|Z(\sigma)|$ times $\mathcal{N}(\sigma, C_2, C_3)$. □

At first glance, this formula looks quite similar to the one in Lemma A.1, but in the formula from Lemma A.1 two of the terms in the product were elements of the center of $\mathbb{C}S_n$ and for the third term, σ, we could replace it with an element of the center of $\mathbb{C}S_n$ and appeal to symmetry. It was crucial that the elements of our product were in the center of $\mathbb{C}S_n$ because it ensured the action of each term on the irreducible representations of S_n was by scalar multiplication. In our case, $\sum_{i=1}^{j} \sum_{\rho \in Z(\sigma)} \rho^{-1} \alpha_i \rho$ and $\sum_{i=1}^{k} \sum_{\rho \in Z(\sigma)} \rho^{-1} \varphi_i \rho$ are not central elements of $\mathbb{C}S_n$, so we cannot say as much about the trace of their product in general.

All is not lost, however, in the particular case where σ is a p-cycle in S_p for p a prime, we can compute $\mathcal{N}(\sigma, C_2, C_3)$ exactly.

Theorem A.12. *Let p be prime and σ a p-cycle in S_p. Then for C_2, C_3 conjugacy classes in S_p,*

- *If exactly one of C_2, C_3 is the identity and the other is the set of p-cycles, $\mathcal{N}(\sigma, C_2, C_3) = 1$.*
- *If C_2 and C_3 are both the set of p-cycles, then*

$$\mathcal{N}(\sigma, C_2, C_3) = \frac{(p-2)(p-1)}{p} + \frac{(p-1)!}{p^2} \sum_{V_i \in Irr\, S_p} \frac{\chi_(\sigma)}{\chi_i(Id)}.$$

- *Otherwise*

$$\mathcal{N}(\sigma, C_2, C_3) = \frac{|C_2||C_3|}{p(p!)} \sum_{V_i \in Irr\ S_p} \frac{\chi_(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(Id)}.$$

Note that the third case above is the lower bound from Theorem A.10.

Proof. For the case where exactly one of C_2 or C_3 is the identity and the other is the set of p-cycles, we can compute $\mathcal{N}(\sigma, C_2, C_3)$ simply and directly. Without loss of generality assume that C_2 is the identity and C_3 are the p-cycles. Then $\alpha = ()$, and whenever $(\sigma, (), \varphi)$ is a hypermap we have:

$$\sigma \cdot () \cdot \varphi = () \implies \varphi = \sigma^{-1}$$

and there is a unique choice of φ that makes $(\sigma, (), \varphi)$ a hypermap.

For the other cases, as in Theorem A.10 we note that the number of ways to choose $(\alpha, \varphi) \in C_2 \times C_3$ such that $(\sigma, \alpha, \varphi)$ forms a hypermap is given by

$$\frac{|C_2||C_3|}{p!} \sum_{V_i \in \mathrm{Irr}\ S_p} \frac{\chi_i(\sigma)\chi_i(\alpha)\chi_i(\varphi)}{\chi_i(\mathrm{Id})}.$$

Letting $H = \{(\sigma, \alpha, \varphi) | \alpha \in C_2, \varphi \in C_3, \sigma\alpha\varphi = ()\}$, we saw in the proof of Theorem A.10 that the number of orbits of H under the conjugation action by $Z(\sigma)$ is equal to $\mathcal{N}(\sigma, C_2, C_3)$. Now we procede by determining precisely how many elements of H have orbits of size less than $|Z(\sigma)|$ under the conjugation action and showing that if they do, they are fixed by the action.

First we note that when σ is a p-cycle in S_p, $|Z(\sigma)| = \langle\sigma\rangle = \{(), \sigma, \sigma^2, \ldots, \sigma^{p-1}\}$. Since the conjugacy class of σ in S_p is the set of p-cycles, and there are $(p-1)!$ of them, by the Orbit-Stabilizer Theorem we have

$$|Z(\sigma)| = \frac{|S_p|}{(p-1)!} = p.$$

Since σ certainly commutes with other powers of σ, and σ is of order p, $\langle\sigma\rangle$ must be all of $Z(\sigma)$.

Now, since $|Z(\sigma)| = p$, a prime, all orbits of H under the conjugation action by $Z(\sigma)$ have order dividing p, so each orbit has either size p or size 1. Suppose $(\sigma, \alpha, \varphi)$ is a hypermap which is fixed under conjugation by $Z(\sigma)$. I.e., for all $\rho \in Z(\sigma)$

$$\rho^{-1}\alpha\rho = \alpha, \quad \rho^{-1}\varphi\rho = \varphi.$$

In particular, since $\sigma \in Z(\sigma)$ this means

$$\sigma^{-1}\alpha\sigma = \alpha, \quad \sigma^{-1}\varphi\sigma = \varphi$$

so both α and φ are elements of $Z(\sigma)$. Since $|Z(\sigma)| = p$, all of its non-identity elements must have order p, and the only order-p elements of S_p are the p-cycles, so we immediately conclude that C_2 and C_3 are either the p-cycles or the identity. We already handled the case where precisely one of them is the identity.

Next suppose that both C_2 and C_3 are the p-cycles. The fixed points of H under conjugation by $Z(\sigma)$ are the hypermaps $(\sigma, \alpha, \varphi)$ with $\alpha, \varphi \in \langle\sigma\rangle$ (and not the

identity) so we simply need to count those. There are $p-1$ ways to choose α as a non-identity element of $Z(\sigma)$, namely $\alpha = \sigma^k$ for $1 \leq k \leq p-1$. Since $\sigma\alpha\varphi = ()$ for a hypermap we have:

$$() = \sigma\alpha\varphi = \sigma\sigma^k\varphi \implies \varphi = \sigma^{-(k+1)}.$$

As long as $k+1 \neq p$, this uniquely determines φ is a p-cycle in $Z(\sigma)$, so we see there are $p-2$ choices of $(\alpha, \varphi) \in C_2 \times C_3$ yielding hypermaps $(\sigma, \alpha, \varphi)$ which are fixed under the action of $Z(\sigma)$. All of the other orbits of H under the $Z(\sigma)$ conjugation action have size p, so we have:

$$\begin{aligned}\mathcal{N}(\sigma, C_2, C_3) &= p - 2 + \frac{|H| - (p-2)}{p} \\ &= (p-2)\frac{p-1}{p} + \frac{|H|}{p}\end{aligned}$$

and since $|H|$ is the upper bound from Theorem A.10 we have

$$= \frac{(p-2)(p-1)}{p} + \frac{|C_2||C_3|}{p \cdot p!} \sum_{V_i \in \mathrm{Irr}\ S_p} \frac{\chi_i(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(\mathrm{Id})}$$

Substituting $|C_2| = |C_3| = (p-1)!$ and noting that $\sigma \in C_2 = C_3$, so $\chi_i(\sigma) = \chi_i(C_2) = \chi_i(C_3)$ we have

$$= \frac{(p-2)(p-1)}{p} + \frac{(p-1)!}{p^2} \sum_{V_i \in \mathrm{Irr}\ S_p} \frac{\chi_i(\sigma)^3}{\chi_i(\mathrm{Id})}$$

There is a procedure called the Murnaghan-Nakayama rule to compute irreducible characters of the symmetric groups, which is beyond the scope of this dissertation, but we refer the interested reader to [19]. Application of this rule for computing characters of symmetric groups shows that $\chi_i(\sigma) \in \{0, 1, -1\}$ for any irreducible character χ_i and any prime p, so we can replace $\chi_i(\sigma)^3$ in the summation above with $\chi_i(\sigma)$ giving the desired formula.

If C_2 and C_3 were the identity, the only possible $(\sigma, \alpha, \varphi)$ would be $(\sigma, (), ())$ but $\sigma \cdot () \cdot () = \sigma$, so this is not a hypermap. This means $\mathcal{N}(\sigma, C_2, C_3) = 0$, but it also means that since H is empty, there are no elements of $|H|/p = 0 = \mathcal{N}(\sigma, C_2, C_3)$. Substituting in the upper bound from Theorem A.10 for $|H|$ we obtain the desired value:

$$\mathcal{N}(\sigma, C_2, C_3) = \frac{|H|}{p} = \frac{|C_2||C_3|}{p(p!)} \sum_{V_i \in \text{Irr } S_p} \frac{\chi_i(\sigma)\chi_i(C_2)\chi_i(C_3)}{\chi_i(\text{Id})}.$$

Finally, if at least one of C_2, C_3 is neither the identity element nor the p-cycles, then we already saw that H has no fixed points under the conjugation action by $Z(\sigma)$. Then each orbit has size p and $\mathcal{N}(\sigma, C_2, C_3) = |H|/p$ which completes the proof. □

We cannot end this section without the following corollary of Theorem A.12 which yields a congruency result about sums of reciprocals of binomial coefficients.

Corollary A.12.1. *For any prime p*

$$(p-1)! \sum_{i=0}^{p-1} (-1)^i \binom{p-1}{i}^{-1} \equiv -2p \; mod \, p^2$$

Proof. We consider $\mathcal{N}(\sigma, C_2, C_3)$ where σ is a p-cycle in S_p and $C_2 = C_3$ are the conjugacy class of p-cycles. From Theorem A.12

$$\mathcal{N}(\sigma, C_2, C_3) = \frac{(p-2)(p-1)}{p} + \frac{(p-1)!}{p^2} \sum_{V_i \in \text{Irr } S_p} \frac{\chi_i(\sigma)}{\chi_i(\text{Id})}.$$

Computing the values of $\frac{\chi_i(\sigma)}{\chi_i(\text{Id})}$ for each irreducible representation of S_p (it is particularly easy to compute χ_i for these two conjugacy classes using the Murnaghan-Nakayama rule) gives that

$$\sum_{V_i \in \text{Irr } S_p} \frac{\chi_i(\sigma)}{\chi_i(\text{Id})} = \sum_{i=0}^{p-1} (-1)^i \binom{p-1}{i}^{-1}.$$

Then the formula for $\mathcal{N}(\sigma, C_2, C_3)$ becomes

$$\mathcal{N}(\sigma, C_2, C_3) = \frac{p^3 - 3p^2 + 2p}{p^2} + \frac{(p-1)!}{p^2} \sum_{i=0}^{p-1} (-1)^i \binom{p-1}{i}^{-1}.$$

Noting that $\mathcal{N}(\sigma, C_2, C_3)$ counts the size of a finite set and must therefore be a nonnegative integer (in fact we know from the proof of Theorem A.12 that it is positive) we know that

$$p^2 - 3 + \frac{2p}{p^2} + \frac{(p-1)!}{p^2} \sum_{i=0}^{p-1} (-1)^i \binom{p-1}{i}^{-1}$$

is an integer, which gives the desired result. □

Appendix B Full Code of Computer Program to Compute $\mathcal{R}_g$ and $\mathcal{C}_g$

This appendix provides the code used to run Algorithm 1. It is written in the Julia language and broken up into several parts. The parts are organized as follows (with various helper functions distributed amongst the parts). The algorithm as a whole is run using the file minseps.jl, which also processes the embeddings found to compute the $\mathcal{R}_g$ into isomorphism classes of graphs for the $\mathcal{C}_g$. In search_organization.jl the case breakdown for the various searches is handled, and computations to convert the hypermaps found back into the corresponding combinatorial maps is handled. In hypermap_search.jl the functions to compute all isomorphism classes of hypermaps with the given conjugacy classes is handled, and hypermap_utils.jl provides a few useful functions for interacting with hypermaps. Then combinadic.jl includes iterators for generating combinatorial objects, in particular a method to iterate over elements of a given conugacy class in S_n, perm_utils.jl has some frequently used methods for permutations and permutation groups, and murnaghan_nakayama.jl implements the Murgnahan-Nakayama algorithm for computing characters of symmetric groups need for memory pre-allocation (the current implementation only implements the Murnaghan-Nakayama algorithm for characters which do not evaluate to zero in the cases where memory usage was an issue).

minseps.jl:

```
include("search_organization.jl")

include("perm_utils.jl")

using Graphs
```

```
# Input should be the graphs found by make_minseps functions for genus g,
# along with the sets I_h for h < g
function make_I_g(g_graphlist, existing_graphs)
  I_g = []
  # for each graph in g_graphlist, want to iso test it againt all
  # of existing_graphs, if it's not iso to any, it stays
  for graph in g_graphlist
    if graph == []
      isDupe = 1
    else
      isDupe = 0
      for e_graph in existing_graphs
        # The following line should be tested thoroughly whenever
        # OSCAR.jl or Graphs.jl is updated.
        if Graphs.Experimental.has_isomorph(graph, e_graph) ==1
          isDupe = 1
          break
        end
      end
      if isDupe == 0
        push!(I_g, graph)
      end
    end
  end
  return(I_g)
end


function main(max_g::Int)
  graphmaketime = 0.0
  graphisotime = 0.0
  I_g_list = [[] for i in 0:max_g]
  I_gs = [Graphs.Graph([1;;])]
```

```
  I_g_list[1] = I_gs
  for g in 1:max_g
    genus_g_duals= generate_minseps_genus(g)
    x=time()
    E_g_count = count_embeds(genus_g_duals)
    g_minseps = dual_list_to_minseps(genus_g_duals)
    println(string(g))
    flush(stdout)
    println("process time = ")
    println(string(time()-x))
    println("Size of E_g = ")
    println(string(E_g_count))
    flush(stdout)
    graphsg = minseps_list_to_graphs(g_minseps, g)
    I_g = make_I_g(graphsg, I_gs)
    I_g_list[g+1] = I_g
    I_gs = vcat(I_gs, I_g)
    println("Size of I_g = ")
    println(length(I_g))
    flush(stdout)
  end
end


main(4)
```

search_organization.jl

```
include("hypermap_search.jl")
include("hypermap_utils.jl")
include("perm_utils.jl")
using Graphs
using Combinatorics
using DataStructures
```

```
#  First step: given g,ghat, E, find all valid triples
# lambda_1,lambda_2,lambda_3 such that they are minsep candidates.
# Note at this point choice of one lambda doesn't affect the others,
# so just make 3 lists.
function get_class_candidates(g::Int, ghat::Int, E::Int, i::Int)
  # recall that the number of vertices is 2+g-ghat,
  # and i is the number of black vertices.
  j = 2+g-ghat -i
  psi_candidates = Combinatorics.partitions(E, i)
  phi_candidates = Combinatorics.partitions(E, j)
  # Now we determine the number of faces, F
  F = E - g - ghat
  # Candidate partitions for theta have no 1-cycles, so we
  # construct partitions of E-F, and then add one to each block.
  theta_candidates = [x + ones(Int, F) for x in
          collect(Combinatorics.partitions((E-F),F))]

  candidates = [psi_candidates, phi_candidates, theta_candidates]
  # Now we need to compute some sizes
  n_psi = sum([conj_class_size(part) for part in psi_candidates])
  n_phi = sum([conj_class_size(part) for part in phi_candidates])
  n_theta = sum([conj_class_size(part) for part in theta_candidates])
  if n_psi*length(phi_candidates) >= n_phi*length(psi_candidates)
    if n_phi >= n_theta
      theta_flag = 1
    else
      theta_flag = 0
    end
    return(theta_flag, psi_candidates, theta_candidates, j, 0)
  else
    if n_psi >= n_theta
      theta_flag =1
    else
```

```
      theta_flag =0
    end
    return(theta_flag, phi_candidates, theta_candidates, i, 1)
  end
end

function make_default_perm(in_partition::Vector{Int})
  defperm = Vector{Vector{Int}}()
  Counter = 1
  for i in 1:length(in_partition)
    push!(defperm, [Counter:Counter+in_partition[i]-1;])
    Counter = Counter+in_partition[i]
  end
  return(defperm)
end

# Computes hypermaps corresponding to minsep embeddings with
# least separating genus g and embedding genus ghat
function get_ghat_minseps(g::Int, ghat::Int)
  big_ghat_minseps = Vector{Vector{Vector{Int}}}[]
  for E in (g+ghat+1):(2*(g+ghat))
    x = time()
    push!(big_ghat_minseps, get_ghat_minseps_edges(g, ghat, E))
    println(string(E))
    println("E edges time =")
    println(string(time()-x))
    flush(stdout)
  end
  ghat_minseps = reduce(vcat, big_ghat_minseps)
  return(ghat_minseps)
end

# Find hypermaps corresponding to minimal separating ribbon graphs
```

```
# with embedding genus ghat and e edges. Specialized version of
# get_ghat_minseps  to be more easily split up for long calculations
# on multiple nodes
function get_ghat_minseps_edges(g::Int, ghat::Int, E::Int)
  ghat_minseps_E = Vector{Vector{Int}}[]
  needed_vertices = 2+g-ghat
  for i in 1:div(needed_vertices,2)
    conj_class_nums = get_class_candidates(g, ghat, E, i)
    psi_choices = conj_class_nums[2]
    if conj_class_nums[1] ==1
      theta_choices = conj_class_nums[3]
      for psi_choice in psi_choices
        psi = make_default_perm(psi_choice)
        for theta_choice in theta_choices
          if g-ghat >1
            append!(ghat_minseps_E, [x for x in
                get_phi_candidates_v1(E,theta_choice, ghat, psi, (conj_class_nums[4]))])
          else
            append!(ghat_minseps_E, [x for x in
                get_phi_candidates_v1(E,theta_choice, ghat, psi, (conj_class_nums[4]),1)])
          end
        end
      end
    else
        phi_cycles = conj_class_nums[4]
        for psi_choice in psi_choices
          psi = make_default_perm(psi_choice)
          outs = get_phi_candidates_v2(E, phi_cycles, ghat, psi)
          append!(ghat_minseps_E, [x for x in outs])
        end
    end
  end
  if g-ghat > 1
```

```
    return([x for x in ghat_minseps_E if is_transitive_pair([Perm(x[1]), Perm(x[2])])])
  else
    return(ghat_minseps_E)
  end
  flush(stdout)
end



# Finds hypermaps corresponding to minimal separating embeddings
# with least separating genus g
function generate_minseps_genus(g::Int)
  total_minseps = []
  for ghat in 0:g
    println("g, ghat = ")
    print(string(g))
    println(string(ghat))
    flush(stdout)
    glist = get_ghat_minseps(g,ghat)
    push!(total_minseps, glist)
  end
  return(reduce(vcat,total_minseps))
end


# Function to determine the actual number of embeddings in R_g
# Takes list of hypermap duals and determines how many elements
# of R_g were counted twice due to distinct colorings
function count_embeds(hypermap_list::Vector{Vector{Vector{Int}}})
  tempcount = length(hypermap_list)
  self_color_counts = zeros(Threads.nthreads())
  Threads.@threads for hypermap in hypermap_list
    x = Perm(hypermap[1])
    y = Perm(hypermap[2])
    if length(cycles(x)) == length(cycles(y))
```

```
      if is_self_color_dual(x, y) == 0
        #tempcount= tempcount -0.5
        self_color_counts[Threads.threadid()] += 1
      end
    end
  end
  tempcount = tempcount - 0.5*(sum(self_color_counts))
  return(tempcount)
end

# Converts hypermap duals of minseps to combinatorial maps
# for those minsep embeddings
function dual_list_to_minseps(dual_list::Vector{Vector{Vector{Int}}})
  minseps_list = [get_dual_map(Perm(dual[1]),Perm(dual[2]), Int(sum(length(k) for k
                in cycles(Perm(dual[1]))))) for dual in dual_list]
  return(minseps_list)
end

# Takes as input a list of graphs and returns a list with
# one copy of each isomorphism class in the graph list.
function getIsoClasses(graphlist)
  isoclasses = Vector{SimpleGraph}()
  for graph in graphlist
    isdupe = 0
    Threads.@threads for G in isoclasses
      if Graphs.Experimental.has_isomorph(graph, G)==1
        isdupe = 1
        break
      end
      #extra check to avoid instability with threads
      if isdupe ==1
        break
      end
```

```
    end
    if isdupe == 0
      push!(isoclasses, graph)
    end
  end
  return(isoclasses)
end

# Converts each combinatorial map in a list into a graph for isomorphism testing.
# Graphs.jl doesn't support isomorhpism testing for multigraphs so loops and
# multiple edges need to be subdivided.
function minseps_list_to_graphs(minsep_list::Vector{Vector{Perm{Int}}}, g::Int)
   temp_graphs_list = [graph_from_embedding(ribbon_graph[1],
                        length(cycles(ribbon_graph[2]))) for ribbon_graph in minsep_list]
   sorted_graphs_list =[Vector{SimpleGraph}() for e in 1:4*g]
   for ribbon_graph in minsep_list
    E = length(cycles(ribbon_graph[2]))
    push!(sorted_graphs_list[E], graph_from_embedding(ribbon_graph[1], E))
  end
  # Need to deal with edge countness
  final_graphs = [getIsoClasses(sorted_graphs_list[e]) for e in 1:length(sorted_graphs_list)]
  fgs = reduce(vcat, final_graphs)
  return(fgs)
end
```

hypermap_search.jl

```
include("perm_utils.jl")
include("combinadic.jl")
include("murnaghan_nakayama.jl")
using Oscar
using Combinatorics
using DataStructures
```

```
function perm_components(part::Array{Array{Int,1},1}, n::Int)
  starters = [chunk[1] for chunk in part]
  perm_parts = [chunk[2:length(chunk)] for chunk in part]
  return(starters, perm_parts,n)
end

# Creates a counter of permutations with given orbits
function perm_counter(s::Array{Int,1})
  return([1:factorial(i-1) for i in s])
end

# Creates a permutation from given "cycle starters", elements that should form each cycle,
# size of symmetric group, and position of each cycle in the lex order of cycles
# comprised of the given elements
function make_perm(starters::Array{Int,1}, perm_parts::Array{Array{Int,1},1}, n::Int, index::NTuple)
  phi = [Int(1):n;]
  for i in 1:length(starters)
    tempperm = vcat(starters[i], nthperm(perm_parts[i], index[i]))
    for j in 1:(length(tempperm)-1)
      phi[tempperm[j]] = tempperm[j+1]
      phi[tempperm[end]] = tempperm[1]
    end
  end
  return(Perm(phi))
end

function make_cartesian(v::Vector{Int})
  n = length(v)
  return(CartesianIndices(ntuple(i-> 1:v[i], n)))
end

# Finds all (psi,phi,theta) that form hypermaps of genus g for given psi
# on n bits where phi has k cycles by searching through all possible phi
```

```
function get_phi_candidates_v2(n::Int,k::Int,g::Int, psitemp::Vector{Vector{Int}})
  println("running v2")
  flush(stdout)
  S = symmetric_group(n)
  sigma = cperm(S, psitemp...)
  psi = Perm(Vector{Int}(sigma))
  needed_verts = n-k -length(cycles(psi))+2 - 2*g
  H = centralizer(S,sigma)
  HH = [Perm(Vector{Int}(x)) for x in H[1]]
  parts = [part for part in Combinatorics.partitions([Int(1):Int(n);],k)]
  outlist = [Vector{Int}[] for i in 1:Threads.nthreads()]
  for part in parts
    decomp = perm_components(part,Int(n))
    PP = make_cartesian([factorial(length(part[i])-1) for i in 1:length(part)])
    Threads.@threads for index in PP
      phi = make_perm(decomp[1],decomp[2], decomp[3], Tuple.(index))
      theta = psi*phi
      # really this is theta^(-1)  but here only the cycle type of theta
      # is needed, which is the same as the cycle type of theta^(-1)
      if length(cycles(theta)) == needed_verts
        if 1 in [length(cyc) for cyc in cycles(theta)]
          nothing
        else
          is_min =1
          for g in HH
            if (g^(-1)*phi*g).d < phi.d
              is_min =0
              break
            end
          end
          if is_min ==1
            push!(outlist[Threads.threadid()], phi.d)
          end
```

```
        end
      end
    end
  end
  return([[Vector{Int}(sigma), x] for x in reduce(vcat,outlist)])
end


# Iterates through tload amount elements of a conjugacy class for use
# in get_phi_candidates_v1.  Starts at position in lex-order among
# that conjugacy class based on avgtload and i (i counts a threads
# position) to allow for thread safe iteration.
function find_phis(i::Int, avgtload::Int, tload::Int, part::Vector{Int},
                        cc::Accumulator{Int, Int}, K::Vector{Int},
      p_inv::Perm{Int}, HH::Vector{Perm{Int}}, PP::Vector{UnitRange{Int}},
      n_phi_cycles::Int, n::Int, outlist::Vector{Vector{Int}})


  x = time()
  combo_part = unrank_combo_partition(1+avgtload*(i-1), n, K, [cc[k] for k in K])
  iter_done=0
  for j in 1:tload
    if iter_done == 1
      break
    end
    decomp = perm_components(ct_to_p(combo_part), n)
    for index in Iterators.product(PP...)
      theta = make_perm(decomp[1], decomp[2], decomp[3], index)
      phi = theta^(-1)*p_inv
      if length(cycles(phi)) == n_phi_cycles
        is_min =1
        for g in HH
          if (theta^g).d < theta.d
            is_min=0
            break
```

```
          end
        end
        if is_min ==1
          push!(outlist, phi.d)
        end
      end
    end
  combo_part, iter_done = conj_class_next!(n, combo_part, [cc[k] for k in K])
  end
end


# Finds all (psi,phi,theta) that form hypermaps of genus g for given
# psi on bits where theta is in a given conjugacy class by searching through
# all possible theta.
function get_phi_candidates_v1(n::Int, part::Vector{Int}, g::Int,
                               psitemp::Vector{Vector{Int}}, n_phi_cycles::Int)
  println("running v1 no nmb")
  flush(stdout)
  if sum(part) != n
    println("Invalid Partition")
    return([])
  else
    cc= counter(part)
    K = reverse(sort([k for k in keys(cc)]))
    PP = perm_counter(vcat([[k for i in 1:cc[k]] for k in K]...))
    total_load_denom = 1
    for k in K
      total_load_denom = total_load_denom*factorial(k)^(cc[k])*factorial(cc[k])
    end
    total_load = div(factorial(n), total_load_denom)
    threadload = div(total_load, Threads.nthreads())
    S = symmetric_group(n)
    sigma = cperm(S,psitemp...)
```

```
    p_inv = Perm(Vector{Int}(sigma^(-1)))
    H = centralizer(S,sigma)
    HH = [Perm(Vector{Int}(x)) for x in H[1]]
    outlist = [Vector{Int}[] for i in 1:Threads.nthreads()]
      for i in 1:Threads.nthreads()
        sizehint!(outlist[i], threadload)
      end
      Threads.@threads for i in 1:Threads.nthreads()
        if i<Threads.nthreads()
          find_phis(i, threadload, threadload, part, cc, K, p_inv, HH, PP, n_phi_cycles, n, outlist[i])
        else
          tload = total_load - threadload*(Threads.nthreads()-1)
          find_phis(i, threadload, tload, part, cc, K, p_inv, HH, PP, n_phi_cycles,n,outlist[i])
        end
      end
    return([[Vector{Int}(sigma), x] for x in reduce(vcat,outlist)])
  end
end


# Version of get_phi_candidates_v1 that pre-allocates memory for storage by
# using the bounds from theorem A.10
function get_phi_candidates_v1(n::Int, part::Vector{Int}, g::Int,
                               psitemp::Vector{Vector{Int}}, n_phi_cycles::Int, nm_flag::Int)
  println("running v1 with nmb")
  flush(stdout)
  if sum(part) != n
    println("Invalid Partition")
    return([])
  else
    cc= counter(part)
    K = reverse(sort([k for k in keys(cc)]))
    PP = perm_counter(vcat([[k for i in 1:cc[k]] for k in K]...))
    if n_phi_cycles ==1
```

```
    hintload = div(map_bound(n, part), n)
  elseif n_phi_cycles ==2
    hintload = div(round(sum([map_bound(n, [n-i, i], part) for i in 1:div(n,2)])), n)
  else
    println("use version without mn rule")
    flush(stdout)
    return([])
  end
  total_load_denom = 1
  for k in K
    total_load_denom = total_load_denom*factorial(k)^(cc[k])*factorial(cc[k])
  end
  total_load = div(factorial(n), total_load_denom)
  threadload = div(total_load, Threads.nthreads())
  S = symmetric_group(n)
  sigma = cperm(S,psitemp...)
  p_inv = Perm(Vector{Int}(sigma^(-1)))
  H = centralizer(S,sigma)
  HH = [Perm(Vector{Int}(x)) for x in H[1]]
  outlist = [Vector{Int}[] for i in 1:Threads.nthreads()]
  for i in 1:Threads.nthreads()
    sizehint!(outlist[i], hintload)
  end
  Threads.@threads for i in 1:Threads.nthreads()
    if i<Threads.nthreads()
      find_phis(i, threadload, threadload, part, cc, K, p_inv, HH, PP, n_phi_cycles, n, outlist[i])
    else
      tload = total_load - threadload*(Threads.nthreads()-1)
      find_phis(i, threadload, tload, part, cc, K, p_inv, HH, PP, n_phi_cycles,n,outlist[i])
    end
  end
  return([[Vector{Int}(sigma), x] for x in reduce(vcat,outlist)])
end
```

```
end
```

hypermap_utils.jl

```
include("perm_utils.jl")

using Oscar


# Takes an integer i and permutations phi, psi.

# Returns [n,m] where n is the length of the orbit of i under phi and

# m is the length of the orbit of i under psi

function point_type(i::Int, phi::Perm{Int},psi::Perm{Int})

  return([[length(cycles(phi)[j]) for j in 1:length(cycles(phi)) if i in cycles(phi)[j]][1],

  [length(cycles(psi)[j]) for j in 1:length(cycles(psi)) if i in cycles(psi)[j]][1]])

end


#Takes a pair of integers x and y and permutations phi and psi.

# Returns true if there is a "color swapping isormophism" mapping x to y

function is_color_swap_iso(x::Int,y::Int,sigma, alpha)

  E = sum(conjclass(sigma))

  f_array = [0 for i in 1:E]

  # fill in f_array

  f_array[x] = y

  counter = 0

  while minimum(f_array)==0 && counter <= E

    for i in 1:E

      if f_array[i] != 0

        f_array[sigma[i]] = alpha[f_array[i]]

        f_array[alpha[i]] = sigma[f_array[i]]

      end

    end

    counter = counter +1

  end

  if sort(f_array) != [1:E;]

    return(0)
```

```
  end
  f = Perm(f_array)
    if f*sigma == alpha*f && f*alpha == sigma*f
      return(1)
    end
  return(0)
end

# Determines if a hypermap is "self-color-dual"
function is_self_color_dual(sigma::Perm, alpha::Perm)
  E = sum(conjclass(sigma))
  # First easy check is is cycle structures match
  if conjclass(sigma) != conjclass(alpha)
    return(0)
  end
  # Now we check for isomorphism
  # Possible destinations for 1: set n = length of the cycle in sigma containing 1, m = lengt of
  # cycle in alpha containing 1.  Destinations are all other elements with type (m,n)
  one_targets = [y for y in 1:E if point_type(y,sigma, alpha) == point_type(1, alpha, sigma)]
  for y in one_targets
    if(is_color_swap_iso(1,y,sigma, alpha)) ==1
      return(1)
    end
  end
  return(0)
end

# Constructs combinatorial map for the dual of a hypermap
function get_dual_map(psi::Perm{Int}, phi::Perm{Int}, n::Int)
  phitemp = [n+phi[i] for i in Int(1):n]
  theta = Perm(vcat([psi[i] for i in Int(1):n],phitemp))
  alpha = Perm(vcat([i+n for i in Int(1):n],[i for i in Int(1):n]))
  sigma = theta^(-1)*alpha
```

```
  return([sigma,alpha])
end


function flipVert(vert, e::Int)
  flipped = []
  for i in vert
    if i >e
      j=i-e
    else
      j = i+e
    end
    push!(flipped, j)
  end
  return(flipped)
end


# Creates a graph from a combinatorial map (sigma,alpha) with
# alpha in our standard form: (1, n+1)(2, n+2)...(n, 2n)
# Adds additional edges to make a simple graph to use built
# in graph isomorphism testing method
function graph_from_embedding(sigma::Perm{Int}, e::Int)
  G = Graphs.Graph(length(cycles(sigma)))
  for i in 1:length(cycles(sigma))
    for x in cycles(sigma)[i]
      if x < e+1
        for j in 1:length(cycles(sigma))
          if x+e in cycles(sigma)[j]
            if Graphs.has_edge(G, i, j)
              Graphs.add_vertex!(G)
              Graphs.add_edge!(G, i, Graphs.nv(G))
              Graphs.add_edge!(G, j, Graphs.nv(G))
              break
            else
```

```
              Graphs.add_edge!(G,i,j)
              break
            end
          end
        end
      end
    end
  end
  return(G)
end
```

combinadic.jl

```
# Implements ranking unranking algorithms for k-combinations from sets of size n
# Assumes that
using SplittablesBase
using DataStructures


# Returns the next combination of k elements
# from a set of size n
function combination_next(n::Int,k::Int, v::Vector{Int})
  j=k
  while v[j] == n-k+j
    j=j-1
    if j==0
      return(v)
    end
  end
  v[j] = v[j]+1
  for i in (j+1):k
    v[i] = v[i-1]+1
  end
  return(v)
end
```

```
# length_vec should be the following vector:
# [sum([length(x) for x in v[i:end]]) for i in 1:length(v)]
# Precomputing this reduces allocations by about 50%
function combination_tuple_next(v::Vector{Vector{Int}}, length_vec::Vector{Int})
 for i in length(v):-1:1
   k = length(v[i])
   if v[i] != [1; length_vec[i]+2-length(v[i]):length_vec[i];]
     v[i] = combination_next(length_vec[i],k, v[i])
     return(v)
   end
   v[i] = [1:k;]
 end
 return(v)
end

# Given a vector of k-subsets of [N], [N-k], [N-2k],...
# computes the next such vector in lex order
function regular_combination_next!(v::Vector{Vector{Int}}, k::Int, N::Int)
  n = length(v)*k
  length_vec = [N:-k:(N-(length(v)-1)*k);]
  if v == [[1+N-n;length_vec[i]+2-k:length_vec[i];] for i in 1:length(length_vec)]
    return(v,1)
  end
  for i in length(v):-1:1
    if i == length(v)
      firstval = 1+length_vec[i]-k
    else
      firstval = 1+length_vec[i]-k*(1+length(v)-i)
    end
    if v[i] != [firstval; length_vec[i]+2-k:length_vec[i];]
      v[i] = combination_next(length_vec[i],k, v[i])
      for j in i+1:length(v)
```

```
        v[j] = [v[j-1][1]: v[j-1][1]+k-1;]
      end
      return(v,0)
    end
  end
  return(v,0)
end

# Finds the r-th vector in lex order of set partitions of [n] into
# mults[i] blocks of size K[i]
function unrank_combo_partition(r::Int, n::Int, K::Vector{Int}, mults::Vector{Int})
  if length(K) != length(mults)
    println("K does not match mults")
    return([[0]])
  elseif length(K) ==1
    return(unrank_reg_combo(r,n,K[1]))
  end
  if n != sum([K[i]*mults[i] for i in 1:length(K)])
    println("error not a valid cycle type")
    return([[0]])
  elseif length(K)==1 && mults ==[1]
    return([[1:K[1];]])
  else
    qdenom =1
    for i in 2:length(K)
      qdenom = qdenom * factorial(mults[i])*(factorial(K[i])^(mults[i]))
    end
    Q = div(factorial(n-(K[1]*mults[1])), qdenom)
    if Q ==0
      println("yikes!")
      return([[1]])
    else
    # Here we're going to do a kind of conversion.
```

```
    # We scale down by i-1 do a unrank_reg_combo, then scale back up by i-1
      for i in 1:n-(K[1]*mults[1])
        Rone = div(r-1, Q)
        Rtwo = r-Q*Rone
        w = unrank_reg_combo(Rone+1, n, K[1], mults[1])
        return([w..., unrank_combo_partition(Rtwo, n-K[1]*mults[1], K[2:end], mults[2:end])...])
      end
    end
  end
end

#This version unranks regular combinations, need to adapt to list of disjoint k-subsets
function unrank_reg_combo(r::Int, n::Int, k::Int)
  c = div(n,k)
  if c*k != n
    println("error k does not divide n")
    return([[0]])
  elseif n <=k
    return([[1:k;]])
  else
    Q = div(factorial(n-k), factorial(c-1)*(factorial(k)^(c-1)))
    if Q ==0
      println("yikes!")
      return([[1]])
    else
      Rone = div(r-1, Q)
      Rtwo = r - Q*Rone
      w = combination_unrank(Rone+1,n,k)
      if n-k ==k
        return([w, [1:k;]])
      else
        return([w, unrank_reg_combo(Rtwo, n-k, k)...])
      end
```

```
    end
  end
end

# Computes the r-th regular combination of l disjoint k-subsets
# of [n] with respect to lex order
function unrank_reg_combo(r::Int, n::Int, k::Int, l::Int)
  if l==0
    return(Vector{Int}[])
  elseif k*l > n
    println("error k*l > n")
    return([[0]])
  else
    for i in 1:(n+1-k*l)
      Q = div(factorial(n+1-i-k), factorial(l-1)*factorial(k)^(l-1)*factorial(n+1-i-l*k))
      if Q==0
        println("yikes!")
        return([[1]])
      elseif r > Q*binomial(n-i, k-1)
        r = r - Q*binomial(n-i,k-1)
      else
        Rone = div(r-1,Q)
        Rtwo = r - Q*Rone
        w = [i-1 for j in 1:k] + combination_unrank(Rone+1, n+1-i, k)
        return([w, [[i-1 for j in 1:k] for m in 1:(l-1)]+unrank_reg_combo(Rtwo, n+1-i-k, k, l-1)...])
      end
    end
  end
end

# Given r disjoint l-subsets of [n], computes the position
# in lex order among all such regular combinations
function rank_reg_combo(v::Vector{Vector{Int}}, l::Int, r::Int)
```

```
  N= l*r
  s = 1
  for i in 1:(r-1)
    s = s+(combination_rank(v[i], N-l*(i-1), l)-1)*div(factorial(N-i*l), factorial(r-i)*factorial(l)^(r-i))
  end
  return(s)
end


# Take n and a permutation in S_n and
# computes the next permutation with
# respect to lex-order
function permutation_next(n, v::Vector{Int})
  k=n-1
  while v[k] > v[k+1]
    k = k-1
  end
  if k==0
    return(v)
  end
  j = n
  while v[k] >v[j]
    j = j-1
  end
  v[k],v[j] = v[j],v[k]
  r = n
  s = k+1
  while r >s
    v[r], v[s] = v[s], v[r]
    r = r-1
    s= s+1
  end
  return(v)
end
```

```
# Computes all k-subsets of [n] in co-lex order
# and returns as binary vectors.  Very fast
# but requires storing all elements
function colex_bitstring(n::Int,k::Int)#,v::Vector{Int})
  if k==0
    return([zeros(Int, n)])
  else
    if k < n
      l1 = [vcat([0], C) for C in colex_bitstring(n-1, k)]
    else
      l1 = Vector{Int64}[]
    end
    l2 = [vcat([1], C) for C in colex_bitstring(n-1,k-1)]
    l = vcat(l1, l2)
    return(l)
  end
end


# input should be n (size of symmetric group) sigma a vector of vectors with entries ordered by decreasing
# length , and increasing first element within entries of the same length.  Within each  length block, the
# entries of the vectors should be integers in {1,... k} where k is the sum of the lengths of vectors in
# this block and all following blocks
function conj_class_next!(n::Int, sigma::Vector{Vector{Int}}, lambda::Vector{Int})
  chunk_track=0
  i = length(lambda)
  chunk_end = length(sigma)
  while i>0
    current_chunk = sigma[1+chunk_end - lambda[i] : chunk_end]
    t = regular_combination_next!(current_chunk, length(current_chunk[1]), n
        - sum([length(sigma[i]) for i in 1:(chunk_end-lambda[i])]))[2]
    if t == 0
      sigma[1+chunk_end - lambda[i] : chunk_end] = current_chunk
```

```
      return(sigma, 0)
    else
      sigma[1+chunk_end - lambda[i] : chunk_end] = [[1:length(current_chunk[1]);] for i in 1:lambda[i]]
    end
    chunk_end = chunk_end - lambda[i]
    i = i-1
  end
  return(sigma, 1)
end


# Returns all k-subsets of [n]
function combinations_list(n,k)
  V = colex_bitstring(n,k)
end


# Computes the position of k-subset of [n]
# in lex-order
function combination_rank(s::Vector{Int},n::Int, k::Int)
  return(binomial(n,k) - sum([binomial(n- s[i], k-i+1) for i in 1:k]))
end


# Computes the r-th k-subset of [n] with
# respect to lex-order
function combination_unrank(r::Int, n::Int, k::Int)
  running_sum = 1
  if r > binomial(n,k)
    println("error r> n choose k")
    return([0])
  elseif n == k
    if r ==1
      return [1:n;]
    else
      println("error n=k, r not ")
```

```
    end
  elseif k == 1
    if r <= n
      return [r]
    else
      println("error")
      return [0]
    end
  else
    running_sum = binomial(n-1, k-1)
    i=1
    while running_sum <r
      i += 1
      running_sum += binomial(n-i, k-1)
    end
    running_sum = running_sum - binomial(n-i, k-1)
    out = vcat([i], [j+i for j in combination_unrank(r-running_sum, n-i, k-1)])
  end
  return(out)
end


# Computes the r-th partition of a set of size N into k blocks of size n
function regular_partition_unrank(r::Int, n::Int, k::Int)
  N = n*k
  total_parts = div(factorial(N), (factorial(n)^k)*factorial(k))
  fixed_first_block_num = div(total_parts, binomial(N-1, n-1))
  first_part_no = 1+ floor(r/fixed_first_block_num)
  new_r = r % fixed_first_block_num
end


# Converts a partition of a set into a ‘‘default’’
# permutation.
function ct_to_p(v::Vector{Vector{Int}})
```

```
  n = sum([length(w) for w in v])
  N = [1:n;]
  Q = [zeros(Int, length(w)) for w in v]
  for i in 1:length(v)
    Q[i] = N[v[i]]
    N= N[filter(x -> !(x in v[i]), eachindex(N))]
  end
  return(Q)
end
```

perm_utils.jl

```
using Combinatorics
using DataStructures
using Oscar

# Returns the conjugacy class of a permutation as a list of cycle lengths
function conjclass(x::Perm{Int})
  return(sort([length(i) for i in cycles(x)]))
end

function conj_class_size(part::Vector{Int})
  n = sum(part)
  l = length(part)
  # The size of a the conjugacy class in S_n is equal to the index of the centralizer subgroup
  # of any element of the conjugacy class
  # The order of the centralizer is the product of the cycle lengths times the number of ways to
  # mix up the cycles of the same length
  numb_k_cycles = counter(part)
  C_ord = prod(part)*prod([factorial(big(numb_k_cycles[k])) for k in 1:n])
  return(factorial(big(n))/C_ord)
end

# If two permutations x and y are conjugate, returns a permutation t such that x^t = y.  Otherwise
```

```
# returns 0.
function solve_conjugation(x::Perm{Int}, y::Perm{Int}, n::Int)
  x_cyc_type = sort([length(cycles(x)[i]) for i in 1:length(cycles(x))])
  y_cyc_type = sort([length(cycles(y)[i]) for i in 1:length(cycles(y))])
  if x_cyc_type != y_cyc_type
    return(0)
  else
    VV = vcat(sort([cycles(x)[i] for i in 1:length(cycles(x))], by=length)...)
    WW = vcat(sort([cycles(y)[i] for i in 1:length(cycles(y))], by=length)...)
    tau = zeros(Int64, n)
    for i in 1:n
      tau[VV[i]] = WW[i]
    end
    t = Perm(tau)
    if t^-1 *x*t != y
      println("OH NO!")
    end
    return(t)
  end
end

# Makes a "default" permutation with given cycle type.  Returns it as a list of integers
function make_default_perm(in_partition::Vector{Int})
  defperm = Vector{Vector{Int64}}()
  counter = 1
  for i in 1:length(in_partition)
    push!(defperm, [counter:counter+in_partition[i]-1;])
    counter = counter+in_partition[i]
  end
  return(defperm)
end

# Takes a pair of permutations and returns whether they generate a transitive permutation group
```

```
# MUST NOT BE RUN IN ANY TYPE OF MULTITHREADED ENVIRONMENT - it calls GAP through Oscar and
# will crash Julia if multiple threads are active
function is_transitive_pair(permpair::Vector{Perm{Int}})
  n = sum(conjclass(permpair[1]))
  S = SymmetricGroup(n)
  sigma = cperm(S, [cycles(permpair[1])[i] for i in 1:length(cycles(permpair[1]))])
  alpha = cperm(S, [cycles(permpair[2])[i] for i in 1:length(cycles(permpair[2]))])
  # type adjust sigma and alph to be elements of S
  H = Oscar.permutation_group(n, [sigma, alpha])
  return(is_transitive(H))
end
```

murnaghan_nakayama.jl

```
include("perm_utils.jl")
#Evaluates the irreducible character of S_n corresponding to partition lambda for
#elements of cycle type rho.  Assumes that lambda has the form k^1,1^{n-k}.
function char_eval(n::Int, lambda::Vector{Int}, rho::Vector{Int})
  h = length(lambda)-1
  r = rho[1]
  charval = 0
  #edge cases:
  if r == n
    return((-1)^h)
  elseif length(lambda) ==1
    return(1)
  elseif length(lambda) ==n
    return((-1)^(r-1)*char_eval(n-r, ones(Int, n-r), rho[2:end]))
  end
  # contribution from leaving only horizontal strip
  if r< lambda[1]
    charval = charval + char_eval(n-r, vcat([lambda[1]-r], lambda[2:end]), rho[2:end])
  end
  # contribution from leaving only vertical strip
```

```
  if r<length(lambda)
    charval = charval + (-1)^(r+1)*char_eval(n-r, vcat([lambda[1]], ones(Int, h-r)), rho[2:end])
  end
  return(charval)
end

# Computes the lower bound on the number of possible n-bit
# hypermaps (sigma,alpha,varphi) where sigma and alpha are
# both n-cycles and varphi has type rho
function map_bound(n::Int, rho::Vector{Int})
  boundval = 0
  bigval = conj_class_size(rho)
  for i in 1:n
    lambda = vcat([i], ones(Int, n-i))
    boundval = boundval + bigval*char_eval(n, lambda, rho)/(n*char_eval(n, lambda, ones(Int, n)))
  end
  return(Int(round(boundval)))
end

# Computes the lower bound on the number of possible n-bit
# hypermaps (sigma,alpha,varphi) where sigma is an n-cycle
# alpha has type phi and varphi has type theta
function map_bound(n::Int, phi::Vector{Int}, theta::Vector{Int})
  boundval = 0
  bigval = conj_class_size(phi)*conj_class_size(theta)
  for i in 1:n
    lambda = vcat([i], ones(Int, n-i))
    boundval = boundval + bigval*char_eval(n,lambda, [n])*char_eval(n,lambda, phi) * char_eval(
    n,lambda,theta)/(factorial(n)*char_eval(n,lambda,ones(Int,n)))
  end
  return(Int(round(boundval)))
end
```