REPRESENTING SURFACES USING POLYGONS

A Thesis Presented

by

Ian Stobert

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fulfillment of the Requirements
for the Degree of Master of Science
Specializing in Mathematics

May, 1997

Accepted by the Faculty of the Graduate College, The University of Vermont, in partial fulfillment of the requirements for the degree of Master of Science, specializing in Mathematics.

Thesis Examination Committee:

Dan S. Archdeacon, Ph.D.	Advisor
Charles J. Colbourn, Ph.D.	
Sanjoy K. Baruah, Ph.D.	Chairperson
Delcie R. Durham, Ph.D.	Dean, Graduate College

Date: April 1, 1997

Abstract

A *surface* is a compact, connected, Hausdorff space that is locally homeomorphic to $\mathbb{R}^2$. A *spine* is a topological-minimal embedding of a graph in a surface Σ with exactly one face which is an open two-cell. This face can be thought of as a polygon which represents Σ as an identification map. In this thesis, we give a method for finding all polygonal representations of the orientable surfaces. When the surface is orientable, we fix an orientation and define the spines as *oriented* embeddings.

To find these identification maps we use the geometric duals of the spines. Since spines are graphs with one face, their duals are graphs with only one vertex. Such a graph is called a *bouquet*. To describe an embedding of a bouquet we need only describe the cyclic permutation of edge ends at the vertex. The relative simplicity of cyclic permutations allow us to analyze combinatorially all bouquets up to a certain size using a computer algorithm. Those embeddings with all faces of size 3 or more are the duals of spines.

In the first and second parts of the thesis we give the necessary graph theoretical background. The third part of the thesis describes algorithms used to find the spines of the tori. In the fourth part of the thesis, we give the results obtained from the implementation of the algorithms. We found that there are 160 spines on the double torus and 345,038 spines on the triple torus. The final part of the thesis discusses methods that were used to double check the computer results.

Acknowledgments

I thank Dr. S. Baruah and Dr. C. Colbourn for reviewing my research. Their comments were very helpful in the final editing of my thesis.

I thank Dr. Dan Archdeacon for being a superlative thesis advisor. For over a year he has patiently explained things to me (over and over and over...), guided me through the writing process, and shown me how much fun research can be. Thanks for being a great mentor and for finding such a wonderful problem to work on.

Finally, I thank Sam for being a pillar of support.

Table of Contents

Acknowledgments ii

List of Figures v

1 Introduction **1**

1.1 Graphs 4

1.2 Surfaces and Embeddings 6

1.3 Rotations 10

1.4 Euler's Formula 12

1.5 Geometric Duals 16

2 The Problem **17**

2.1 Minimal Graphs 18

2.2 Statement of the Problem 21

3 Algorithms **22**

3.1 Overview 22

3.2 Bounds on the Size of a Spine 23

3.3 Generating all Maps of the Bouquets 25

3.4 Counting the Faces of a Graph 29

3.5 The Algorithm for Finding Spines 33

3.6 The Spawning Algorithm 34

4 Results **37**

5 Double Checks **39**

6 Conclusion 48

Appendices 51

A Pascal Source Code for Spine Program 51

B Pascal Source Code for Spawning Program 60

C Bouquets on the Double Torus in Matching Form 73

List of Figures

1 Turning a rectangle into a torus. 1

2 Labeled identification map for the torus. 2

3 Hexagonal identification map for the torus. 3

4 Double torus and triple torus. 4

5 Two drawings of K_5. 5

6 A bouquet with 6 edges (B_6). 6

7 Surfaces of genus 1, 2, or 3. 7

8 Identification map for the Möbius strip. 8

9 Two different rotations of the same graph. 12

10 Cutting the circuit to detach the handle. 14

11 A graph and its geometric dual. 16

12 Contracting an edge. 19

13 An example of a matching 26

14 Two isomorphic matchings of maps of bouquets 28

15 Tracing faces: selecting the next edge. 30

16 Tracing the faces of a map. 31

17 The two non-isomorphic maps of B_2 40

18 The two graphs with 2 vertices and 3 edges 41

19 The three non-isomorphic maps for $v = 2$, $e = 3$ 41

20 The spines of the torus 42

21 The nine spines with $v = 6$ on S_2 44

22 A spine of the double torus 46

1 Introduction

Imagine a large rectangular piece of paper. If we glue the top edge to the bottom edge we get a cylinder. We now connect the ends of the cylinder to get a doughnut shaped object called a *torus* (Figure 1).

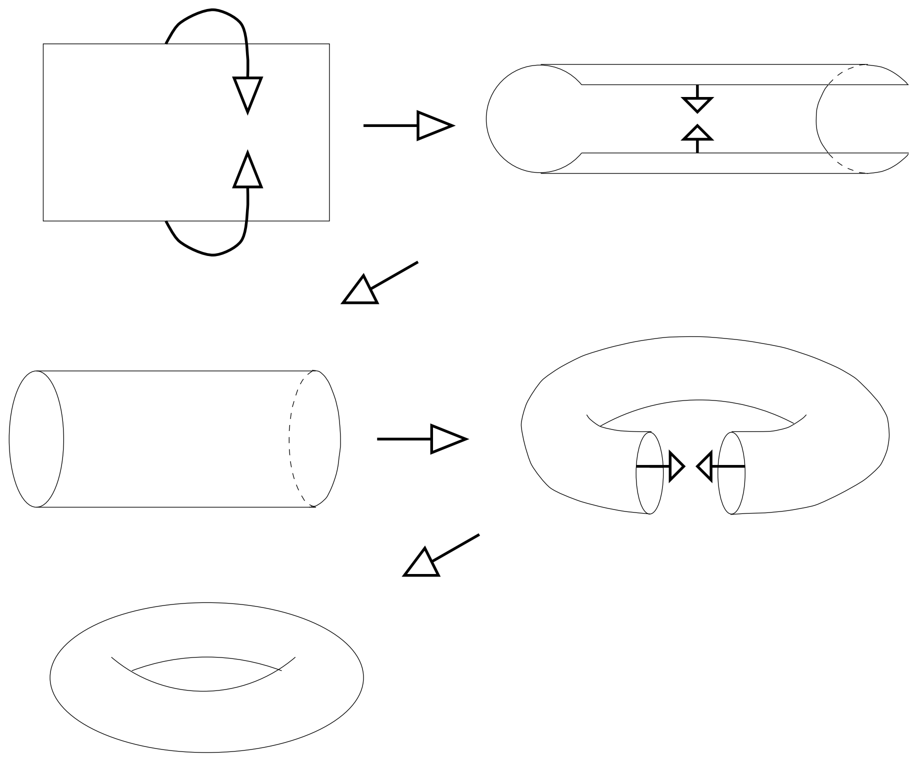

Figure 1: Turning a rectangle into a torus.

We have just transformed a two-dimensional polygon (the rectangle) into a surface embedded in three-dimensional space (the torus). We formed the torus by *identifying* the top and bottom edges and then identifying the left and right edges. We can describe this pictorially by giving the same label to two sides that are to be identified (Figure 2). We also add arrows to the sides of our rectangle with the instruction that when two sides are identified, their arrows must be aligned. This labeled rectangle can now be thought of as a planar representation of the torus. We refer to these labeled polygons as *identification maps*.

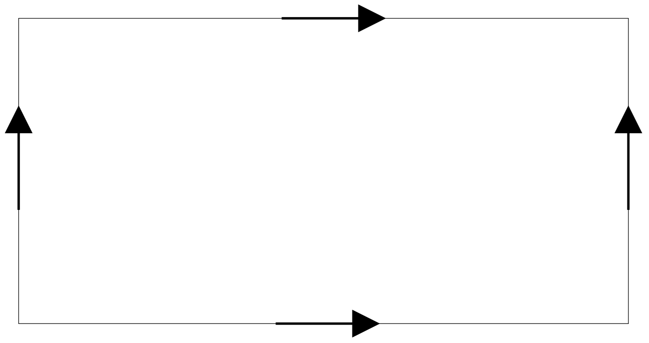

Figure 2: Labeled identification map for the torus.

We may now ask the question: "Are there any other identification maps that can be used to represent the torus?" It turns out that there is only one other identification map for the torus that is not topologically equivalent to the rectangular representation. We explain what is meant by topologically equivalent later in the thesis. In Figure 3 we give the other identification map for the torus.

This second identification map reveals a 3-fold symmetry in the torus that is not obvious with the more commonly used rectangular depiction.

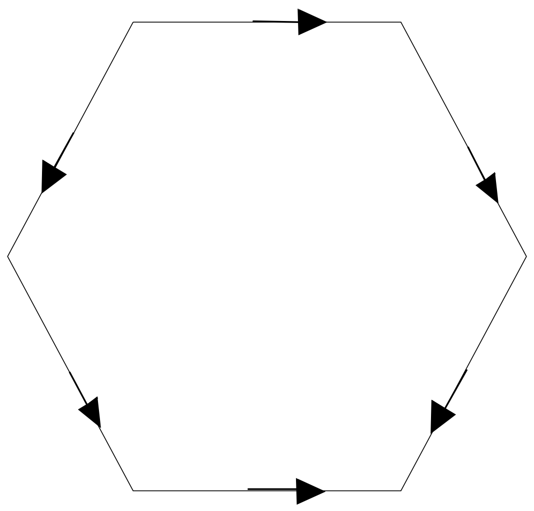

Figure 3: Hexagonal identification map for the torus.

What about other surfaces embedded in three-dimensional space? In this thesis, we are particularly interested in surfaces called the *double torus* and the *triple torus* (Figure 4). We found that there are 160 different identification maps for the double torus and 345,038 for the triple torus.

To find these identification maps, think of how we might obtain an identification map from a surface. Imagine drawing lines that tell us where to cut the surface. The cut surface could then be laid flat and the cut edges would form the sides of the polygon in the identification map. If we place dots on the surface where the lines intersect, then we find that we have drawn a graph. A problem that is essentially topological can now be rephrased in the language of graph theory. The branch of

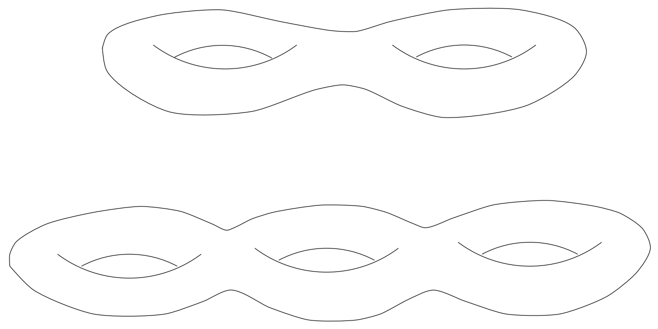

Figure 4: Double torus and triple torus.

graph theory realizing graphs geometrically is called topological graph theory.

The problem of finding all identification maps for a surface is equivalent to the problem of finding all topological-minimal embeddings of graphs with one face. These embeddings were found through an exhaustive search. The numbers involved were quite large, so we used a computer algorithm.

We begin by outlining some of the essential concepts of graph theory and in particular, topological graph theory. In later sections we describe the search algorithms and give a detailed description of our results.

1.1 Graphs

A basic knowledge of graph theory is assumed. For additional background, we recommend the text "Graph Theory with Applications" by Bondy and Murty [2].

We start by defining some important terms in topological graph theory, the branch of graph theory concerned with the drawing of graphs on various topological surfaces. Given a graph G, denote the vertex set by $V(G)$, and the edge set by $E(G)$, where each edge is an unordered pair of vertices. For convenience we let $|V(G)| = v$ and $|E(G)| = e$. Call the vertices that form an edge the *ends* of that edge. If both ends of an edge are the same vertex, then we call the edge a *loop*. If two edges have the same pair of ends, then we say that the edges are *parallel*. A graph with no loops or parallel edges is called a *simple graph*. The *degree* of a vertex is the number of edge ends that are incident to that vertex.

We call a fixed geometrical representation of a graph a *drawing*. Here the vertices are dots and the edges are line segments. In topological graph theory it is very important to distinguish a graph from its many drawings. For example, K_5 is a complete graph on five vertices (every vertex is connected to every other vertex by exactly one edge). In Figure 5 we show two different drawings of K_5.

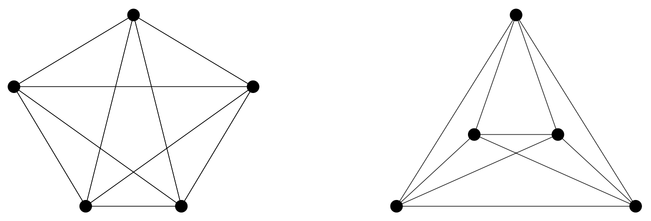

Figure 5: Two drawings of K_5.

If two line segments intersect in a drawing at a point that does not correspond to a vertex, then we call this point a *crossing* of the two edges. Figure 5 gives a drawing of K_5 with 5 crossings and a drawing of K_5 with 1 crossing.

A *bouquet* is a graph with exactly one vertex. The bouquet with n edges or *petals* is denoted B_n (Figure 6).

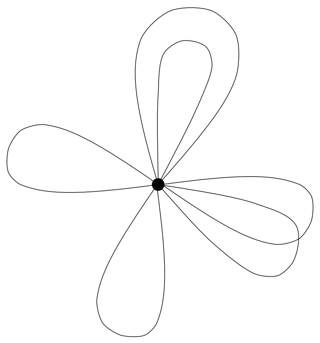
Figure 6: A bouquet with 6 edges (B_6).

1.2 Surfaces and Embeddings

A *two-manifold* is a Hausdorff space X where for all x in X, there exists a neighborhood of x homeomorphic to $\mathbb{R}^2$. A *surface* is a compact, connected two-manifold. Topological graph theory deals with drawing graphs on different surfaces. These surfaces can be grouped into two classes: orientable surfaces and non-orientable surfaces. An orientable surface is obtained by taking a sphere and attaching n handles to it. We denote this surface by S_n.

The sphere is, in a sense, the simplest orientable surface. The next simplest is the *torus*, which most people would think of as a doughnut. You can also think of a torus as a sphere with a handle attached. In many ways, the torus and the sphere with a handle attached seem to be different objects. To a topologist, however, they have the same properties. The most important of these properties is the “hole” in the doughnut or in the sphere handle. The other orientable surfaces are like the torus, but with more holes. A torus with two holes is called a double torus (denoted S_2); similarly the torus with three holes is the triple torus (denoted S_3). In a similar fashion we can make S_n, the n-torus (or in other words the sphere with n handles attached). The number of holes, or handles, on the sphere is called the *genus* of the surface (Figure 7).

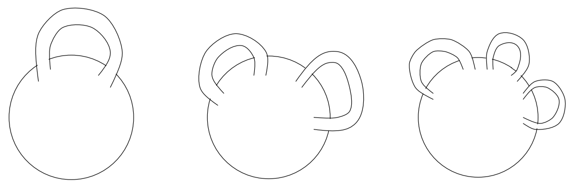

Figure 7: Surfaces of genus 1, 2, or 3.

The other broad class of two-manifolds is the non-orientable surfaces. Unlike orientable surfaces, there is no consistent sense of clockwise or counter-clockwise on a non-orientable surface. On a non-orientable surface we could write a word on the surface and then slide it around continuously so that it would appear as its mirror image somewhere else on the surface! The simplest non-orientable surface is the projective plane. The projective plane can be modeled by a disc with antipodal boundary points identified. Another model of the projective plane is the sphere with antipodal points identified.

Another interesting non-orientable object is the Möbius strip. If we take a long rectangular piece of paper and glue a pair of opposite sides in such a way as to create a twist in the resulting band, then we have a Möbius strip. The resulting object has only one side and only one edge. This edge is homeomorphic to a circle. If we take a sphere and cut a hole in its surface, then we can sew the edge of a Möbius strip into the hole. We call this a *cross-cap*. It is impossible to do this procedure in three-space. We can make a sphere with k cross-caps by cutting k holes in the surface of the sphere and then sewing a cross-cap into each hole. The non-orientable surfaces are described by the number of cross-caps that they have, just as the orientable surfaces are described by genus.

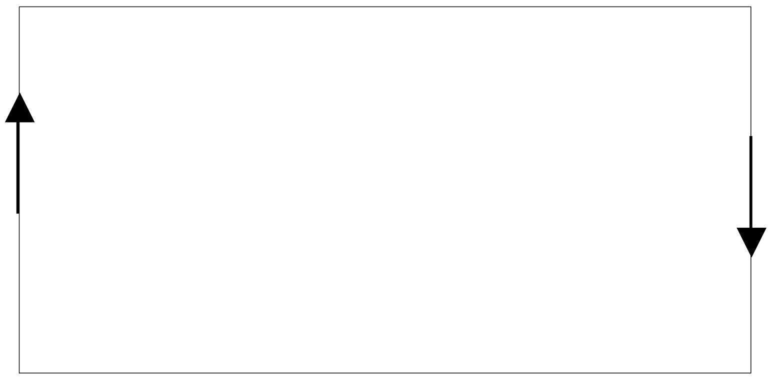

Figure 8: Identification map for the Möbius strip.

Why are graph theorists interested in these surfaces? We would like to understand the properties of surfaces that are locally two-dimensional, in other words, locally like a sheet of paper or any surface where one might draw graphs. It turns out that the different topological properties of surfaces permit a graph to be drawn in one surface in a way that would be impossible on another surface. For example, we were able to draw K_5 in two different ways on the sphere or plane (Figure 5), but no matter how we try, we are always forced to draw it with edges crossing. In contrast, we can draw K_5 on the torus with no crossings.

Now that we have described some surfaces, we are ready to think about how we can draw graphs on the surfaces. Let us first assume that we would like to avoid crossings in our drawings (so edges only meet at their ends). A drawing that avoids such crossings is called an *embedding* of the graph. More technically, an *embedding* of a graph G is a homeomorphism between the geometric realization of G and a subspace of the topological space (in this case, the surface) [1].

If we have a drawing of a graph on an orientable surface that involves crossings then we can add handles to the surface which act as bridges to eliminate these crossings. Consequently, for every finite graph G there exists an n such that G can be embedded in S_n. The *genus of a graph* is the smallest n for which G can be embedded in S_n. If a connected graph G is embedded in some surface S_n, and each component of $S_n - G$ is homeomorphic to an open two-cell, then we call the embedding a *cellular embedding.* We call these open two-cells *faces.*

1.3 Rotations

At this point, we formalize the idea of how we can classify all drawings of a graph into equivalence classes. To do this we define a *rotation* for our graph G.

Label the edge ends of our graph and then describe each edge in our graph as a set of two edge ends. Let $E^2(G)$ denote the set of edge ends in G. If $\{a, b\}$ is a pair of edge ends corresponding to an edge in G, then we have a function defined by the map θ that sends $a \to b$ and $b \to a$. The map θ is a fixed-point-free-involution. We call this map an *end swap.* The edge set of a graph G corresponds to a partition of the edge ends into pairs. Call this partition P_e or $P_e(E^2(G))$.

When we have an orientable surface, it is possible to fix an orientation for the surface. When we fix an orientation on a surface, we can distinguish between an embedding and its mirror image formed by reversing the orientation. With this fixed orientation, we call the embeddings *oriented.* If we have a cellular embedding of G in an orientable surface with a fixed orientation, then we can look at the counter-clockwise ordering of edge ends at each vertex in G. If we write each of these orderings as a cyclic permutation, then the permutation ρ formed by all of these cycles is called a *rotation* of G. The orbits of ρ form a new partition of the edge ends which corresponds to the vertex set of G. Call this partition P_v or $P_v(E^2(G))$.

Now we might ask if we can reverse the process. Is it possible to start with a permutation ρ and derive a cellular embedding in an orientable surface? The answer is yes, and Gross and Tucker [4] ascribe credit for this result (implicitly) to Heffter (1891) and (explicitly) to Edmonds (1960). Moreover, for any two embeddings with the same rotation, there exists an orientation-preserving homeomorphism that maps one into the other. This proves to be very useful, because problems about cellular embeddings can be reduced to combinatorial problems of permutations. We define a *map* to be an ordered pair (G, ρ) where G is a graph and ρ is a rotation. Define the *genus of a map* as a number g such that the map corresponds to a two-cellular embedding in S_g. A finite graph has infinitely many drawings, but only finitely many maps.

We now have the tools to define a graph isomorphism. First we define the equivalence relation on edge ends a and b by:

$$a \sim_i b \Leftrightarrow a, b \text{ are in the same part of the partition } P_i.$$

A bijection $\phi : E^2(G_1) \rightarrow E^2(G_2)$ is a *graph isomorphism* if:

$$i) \qquad a \sim_1 b \Leftrightarrow \phi(a) \sim_1 \phi(b)$$

$$ii) \qquad a \sim_2 b \Leftrightarrow \phi(a) \sim_2 \phi(b)$$

The first condition implies that $\phi\theta = \theta\phi$, that is, ϕ preserves edges. The second condition implies that ϕ preserves vertices.

In Figure 9 we depict two different rotations of the same graph. By considering the counter-clockwise ordering of edge ends in the drawing on the left, we obtain the rotation $\rho = (321)(456)$. In the drawing on the right the rotation is $\rho = (321)(546)$. The first map is a planar embedding of the graph, while the second is not.

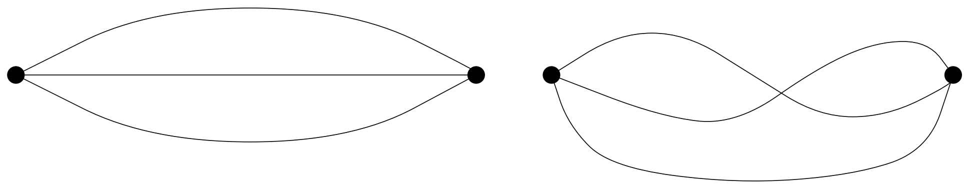

Figure 9: Two different rotations of the same graph.

A *map isomorphism* from (G_1, ρ_1) to (G_2, ρ_2) is a graph isomorphism $\phi : E^2(G_1) \rightarrow E^2(G_2)$ with the added property that

$$\phi\rho(G) = \rho\phi(G).$$

Hence if a and b are adjacent edge ends at a vertex in the rotation, then $\phi(a)$ and $\phi(b)$ are adjacent in $\phi(G)$. Map isomorphisms allow us to decide when two embeddings are "essentially the same".

1.4 Euler's Formula

Given a particular map, how can we find the surface of minimum genus in which it can be embedded? To answer this question there is a useful theorem, originally due to Euler and later refined by Lhuilier (see [4]).

Theorem 1.1 *If we have a connected planar graph G with v vertices, e edges, and f faces, then*

$$v + f - e = 2.$$

Proof: We use induction on e. If $e = 0$, then $v = 1$ and $f = 1$ and the equation is satisfied. Now we assume that the formula is true for all graphs with fewer than e edges and we consider a graph with e edges and v vertices. If G is a tree, then $v = e + 1$ and $f = 1$ and the formula holds. If G is not a tree, then since it is connected it must contain a cycle. Choose one edge from this cycle and remove it. The resulting graph must be connected and it must have v vertices, $(e - 1)$ edges, and $(f - 1)$ faces. We can now apply our induction hypothesis to obtain $v + (f - 1) - (e - 1) = 2$, and if we re-insert the deleted edge we get $v + f - e = 2$ as desired. ■

Here is a generalization of Euler's formula for a surface of genus g.

Theorem 1.2 *If G is a connected graph with v vertices, e edges, and f faces cellularly embedded in a surface of genus g, then*

$$v + f - e = 2 - 2g.$$

Proof: (This proof is modeled after a proof by Wilson [8].) This formula can be proven by induction on g. Imagine the graph G drawn on a surface with g handles attached. Let the number of vertices in G be v_1, the number of edges be e_1 and the number of faces be f_1. There must exist edges that pass over the handle and under the handle (like roads that pass over a bridge and under a bridge). If not, then the handle would be unnecessary and the graph would be embedded in a surface of lower genus. Similarly, there must exist a circuit in G that contains exactly one end of a handle. Call this circuit C.

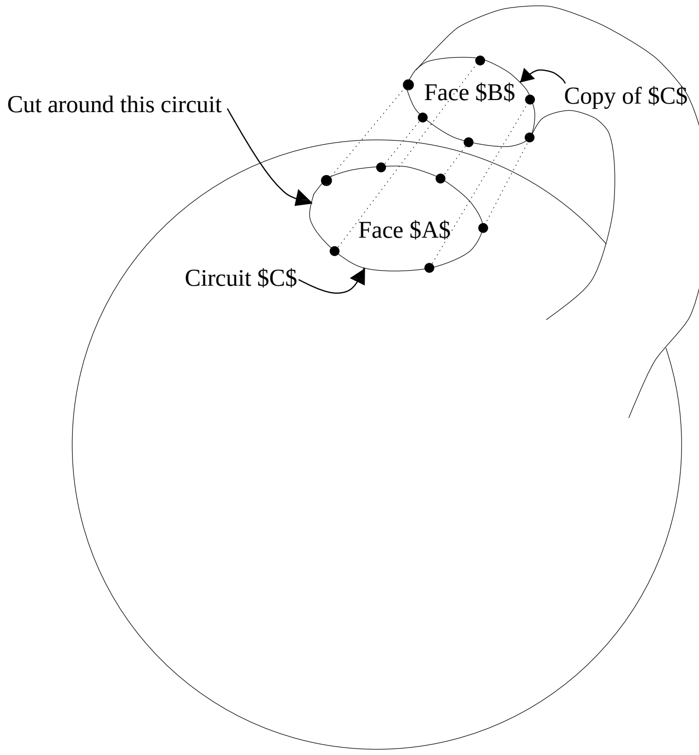

Figure 10: Cutting the circuit to detach the handle.

If we cut the surface along this circuit, then we are detaching one end of a handle (Figure 10). We can create two copies of the subgraph C, one at the "hole" in the sphere and one at the free end of the detached handle. Let the new number of vertices be v_2 and the new number of edges be e_2. Since C is a circuit, the number of vertices and edges being added is the same. Hence $v_2 - e_2 = v_1 - e_1$.

The "holes" in the regions surrounded by these two circuits can be filled in with two new faces (see faces A and B in Figure 10). Let the number of faces in this newly formed, connected graph be f_2 and note that $f_2 = f_1 + 2$. The new surface is now of genus $g-1$, so we can use the inductive hypothesis to conclude that

$$\begin{aligned} & v_2 + f_2 - e_2 = 2 - 2(g-1) \\ \Longrightarrow \quad & v_2 + (f_1 + 2) - e_2 = 2 - 2g + 2 \\ \Longrightarrow \quad & v_1 + f_1 - e_1 = 2 - 2g \end{aligned}$$

which is the desired formula. ■

The number $2-2g$ is called the *Euler characteristic* of the surface. A similar formula for non-orientable surfaces with k cross-caps:

$$v + f - e = 2 - k.$$

For a proof of this theorem see Gross and Tucker [4].

1.5 Geometric Duals

A useful idea is the concept of a *geometric dual* of an embedding. A geometric dual of a graph G is a graph H obtained as follows: for each face of G create an associated vertex in H, and for each edge in G create an edge in H which joins the vertices of H corresponding to the two faces that bordered on the original edge in G. Clearly, if G is embeddable in S_n then so is H. In addition, the number of faces in G is equal to the number of vertices in H and the number of vertices in G is equal to the number of faces in H, while the number of edges remains the same. If G has v vertices, e edges and f faces, then by convention we denote the corresponding values on the dual by $v^\star, e^\star$ and $f^\star$. Note that $e = e^\star$ while $v = f^\star$ and $f = v^\star$.

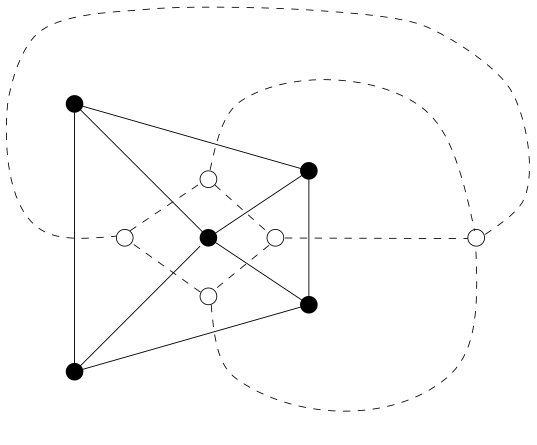

Figure 11: A graph and its geometric dual.

2 The Problem

In the introduction we discussed the relationship between the torus and the labeled rectangle which we call the identification map (Figure 2). The question that naturally arises is "Given a particular surface, what are all of the possible identification maps for that surface?" By finding all of the embeddings of graphs in a surface with exactly one face, we are finding all of the identification maps for that surface. The edges of the graph with one face effectively describe lines along which the surface could be cut. If such a cut is made, then the resulting object is an identification map. Similarly, if one starts with an identification map and forms a surface, then the "seams" in the surface would form the edges of a graph with one face on that surface. Hence the problems of finding all identification maps and finding all embeddings of graphs with one face are equivalent.

There are an infinite number of graphs (and maps) with one face on any orientable surface (and hence an infinite number of identification maps), but many of these graphs can be grouped together by defining some partial orderings on sets of graphs. This is done by describing operations that can be performed on graphs to obtain one graph from another. For example, if a graph H can be obtained from a graph G by deleting an edge then we could say that $H \leq G$. With some of these operations, properties that the graphs hold are preserved, so the "smaller" graph has the same property as the "larger" graph. For convenience, we select the smallest graph among all graphs that can be related by this partial ordering, to represent this family of related graphs. In the next section, we describe these partial orderings.

2.1 Minimal Graphs

It is often useful to find a minimal set of graphs with a certain property. The most famous example of this is Kuratowski's theorem.

Theorem 2.1 *A graph G is planar if and only if G contains no subgraph homeomorphic with K_5 or $K_{3,3}$.*

In this example $\{K_5, K_{3,3}\}$ form a minimal set of graphs which characterize all non-planar graphs. What do we mean by minimal? We define operations that we can perform on graphs to obtain other graphs. One such operation is edge deletion, which means to delete an edge e, leaving $G - e$. Another is vertex deletion which means to deleting a vertex and all of its incident edges. If we can obtain a graph H from a graph G by a series of edge and vertex deletions, then we say that H is a *subgraph* of G. There is a third operation called *edge contraction*. Edge contraction is the result of deleting an edge e and then identifying its endpoints. Figure 12 illustrates the contraction of the edge labeled A. We will use these operations to define a partial ordering of graphs.

We can eliminate all vertices of degree two from a graph by contracting edges as necessary. We say that a graph H is an *elementary subdivision* of G if we can form H from G by deleting an edge of G and replacing it with two edges joined by a new vertex (creating a new vertex of degree two). The inverse of this operation is called *suppression of a degree-two vertex*. This is equivalent to contracting one of the two

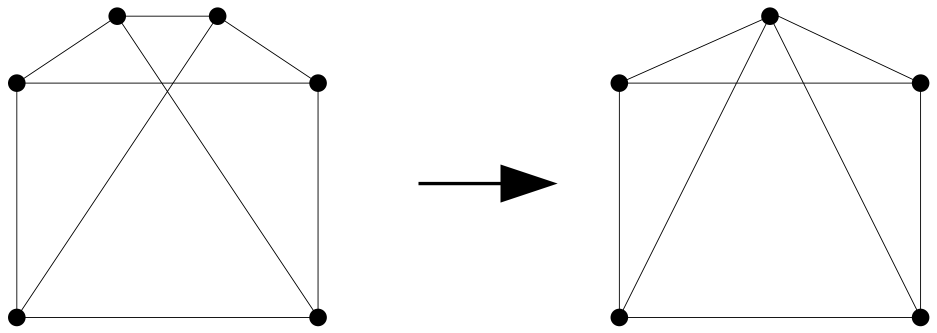
Figure 12: Contracting an edge.

edges incident to a vertex of degree two. Two graphs are said to be *homeomorphic* if we can transform one into the other through a series of elementary subdivisions and suppressions of vertices of degree two. The reason we use the term homeomorphism is that a path containing vertices of degree two has the same topological properties as a single edge.

The *topological order* is defined by $H \leq G$ if and only if H is homeomorphic to a subgraph of G. In other words, H can be obtained from G by deletion of isolated vertices, edge deletion, and suppression of vertices of degree two. Kuratowski's Theorem is equivalent to saying that the only topologically minimal nonplanar graphs are K_5 and $K_{3,3}$.

Another ordering that we can impose on a set of graphs is the *minor order*. To define when $H \leq G$ under this ordering we consider three valid operations: deletion of isolated vertices, deletion of edges, and edge contraction. When H can be formed from G by a series of these operations we say that H is a *minor* of G.

The previous two partial orderings were defined on graphs but we can similarly define partial orderings on maps. We are now ready to define *minor minimal* and *topological minimal* maps. Let $\mathcal{P}_\Sigma$ be the set of maps embedded in the surface Σ with one cellular face. If $M_i \in \mathcal{P}_\Sigma$ and each $M_j \leq M_i$ is not in $\mathcal{P}_\Sigma$, then M_i is a topological-minimal map in $\mathcal{P}_\Sigma$. We can similarly define minor minimal maps in $\mathcal{P}_\Sigma$ by replacing the topological order with the minor order. The minor minimal graphs in $\mathcal{P}_\Sigma$ are maps with a single vertex.

Lemma 2.2 *Topological minimal maps in $\mathcal{P}_\Sigma$ have no vertices of degree 0, 1, or 2.*

Proof: Graphs with vertices of degree 0 or 1 are not minimal under topological ordering because these vertices can be deleted by vertex and edge deletions. All vertices of degree 2 can be eliminated by contracting an adjacent edge. Hence $\delta \geq 3$ in a topological minimal map. ■

2.2 Statement of the Problem

We would like to find an algorithm for generating all oriented topological minimal maps that embed on an orientable surface of genus g with exactly one face. This is equivalent to finding all identification maps for these surfaces. The dual problem is to find every cellular embedding of a bouquet on the surface such that every face has three or more edges on its boundary.

3 Algorithms

3.1 Overview

The geometric dual is very useful in the search for spines. The geometric dual of a spine (a map with one face) is a bouquet (a map with one vertex). The advantage of dealing with bouquets is the fact that a rotation for a bouquet is determined by the rotation at a single vertex. This rotation forms a cyclic permutation, which is convenient to use in a combinatorial search by computer. It is easy to generate all $(2n-1)!$ possible cyclic permutations of edge ends at this vertex when n is not too large. In fact, it is not necessary to test all of these permutations, as there is often a high degree of symmetry in bouquets which reduces the number of rotations that need to be considered.

In this section we outline the algorithms used to find the spines of orientable surfaces. Because of the huge number of cases that need to be checked, a computer program is used to carry out these algorithms. In this section we give the outline of the algorithms used to solve the problem. The details of the computer algorithm can be found in the Pascal source code in Appendix B.

Our first algorithm has three main steps:

Step 1: We determine bounds on the number of edges $e^\star$ in the dual graph, based on Euler's formula and the conditions for a topological minimal map.

Step 2: We find all non-isomorphic maps for the bouquets with the prescribed number of edges.

Step 3: For each of the maps found in the second step we count the number of faces in each map. This value of $f^\star$ allows us to determine the genus of the map. If this is the genus of the surface in question, then we have a map of a bouquet whose dual is a spine.

A more detailed description of this algorithm is given in section 3.5. We now give some of the ideas necessary to carry out the steps in the algorithm.

3.2 Bounds on the Size of a Spine

The first step in the search for spines is determining all possible values for v and e when we make the assumption that $f = 1$. We do this by using Euler's formula.

Assume that the surface is S_g where g is the genus. Euler's formula gives us

$$v + f - e = 2 - 2g.$$

Since we are searching for graphs with just one face this gives

$$e = v + 2g - 1.$$

We are searching for topological minimal graphs, so $\delta(G) \geq 3$ where δ is the minimum degree of G. This implies that $e \geq 3v/2$. Combining these formulas yields

$$\begin{aligned} v + 2g - 1 &\geq \frac{3v}{2} \\ \implies v &\leq 4g - 2. \end{aligned}$$

So the range of possible values for v is given by

$$1 \leq v \leq 4g - 2$$

and the range of possible values for e is given by

$$2g \leq e \leq 6g - 3.$$

For example, on the double torus ($g = 2$) we get $1 \leq v \leq 6$ with $e = v + 3$. This means that the possible values for e on the double torus are $\{4, 5, 6, 7, 8, 9\}$. On the triple torus ($g = 3$) we get $1 \leq v \leq 10$ with $e = v + 5$. The range of possible values for e becomes $\{6, 7, 8, \ldots, 15\}$.

Bouquets, which are geometric duals of the spines, have the same range of possible values for $e^\star$ as the spines, but only one vertex. The formula relating $f^\star$ and $e^\star$ is

$$e^\star = f^\star + 2g - 1$$

Since there are only a finite number of graphs with a bounded number of vertices and edges, we can easily see why there are only a finite number of topological minimal graphs that embed in a surface of genus g.

3.3 Generating all Maps of the Bouquets

We must first develop two notations for describing the rotation of an embedded bouquet.

The *matching description* of a bouquet (B_n) is formed by examining a small neighborhood about the vertex and fixing ρ, labeling the $2n$ edge ends in clockwise order: $(1, 2, 3, \ldots, 2n)$. The matching corresponding to this rotation is a permutation of $\{1, 2, 3, \ldots, 2n\}$ which lists these numbers in n pairs which correspond to the edges of G. In Figure 13 we show a rotation of a graph with matching (1 5 , 2 4 , 3 7 , 6 9 , 8 10). When only some of the edges in the bouquet are specified we get a *partial matching.*

The *neighbor description* of a bouquet is not as easy to transcribe from the rotation, but proves to be much more useful when doing a face count for the map. The neighbor description starts with a labeling of the edge ends as we did with the matching description (Figure 13). Now we create a $2n$-tuple where the i^{th} entry is found by writing down the label of the other end of the edge with i at one end. In the above example, the matching (1 5 , 2 4 , 3 7 , 6 9 , 8 10) would correspond to the neighbor description (5 4 7 2 1 9 3 10 6 8).

If we have an identification map for a surface then it can be represented as a polygon with labeled, directed sides. We can list the labels of the perimeter of this polygon in cyclic order, writing the label in one form (say lower-case) if the direction

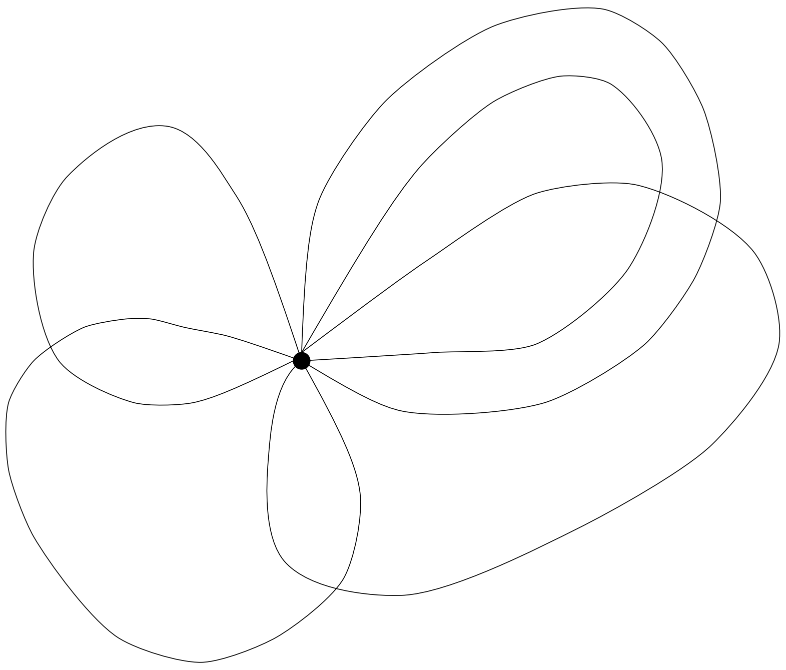

Figure 13: An example of a matching

arrow is in the same direction as our cycle of the perimeter, and writing the label in another form (say upper-case) if the direction arrow is pointing the opposite way. For example, if we consider the identification map for the torus in Figure 2 we get the cycle (a b A B). We call this the *word form* of the identification map.

If we have a bouquet (B_n) with n edges and $2n$ edge ends, then it is possible to count all matchings for B_n. First we must fix the orientation on the surface. This allows us to fix ρ and label the edge ends $\{1, 2, 3, \ldots, 2n\}$ in clockwise order around the vertex. Now let us select any one of these edge ends. Without loss of generality let the selected edge be labeled 1. There are $(2n - 1)$ possible labels for the other end of this edge. Fix the smallest unused symbol, and select one of the remaining $2n-3$ remaining labels for the other end of this edge. Continuing this way, we find the number of matchings is

$$(2n-1)(2n-3)(2n-5)\dots(5)(3)(1) = \frac{(2n)!}{n!2^n}.$$

Clearly, this number is very large for large values of n: for example, when $n = 10$ the number of matchings is 654,729,075. However many of these matchings correspond to isomorphic maps.

The 2^n in the formula comes from the fact that an end swap does not change the rotation. The $(2n)!$ counts the number of ways to order the edge ends. This is why the matchings (1 5 , 4 2 , 3 7 , 9 6 , 8 10) and (1 5 , 2 4 , 3 7 , 6 9 , 8 10) are equivalent. Similarly, permuting the pairs in a matching does not change the bouquet being represented, so the matchings (1 5 , 2 4 , 3 7 , 6 9 , 8 10) and (2 4 , 3 7 , 1 5 , 8 10 , 6 9), are equivalent (this is where the $n!$ comes from in the formula). In order to have a canonical representation for these matchings we use the notion of *lexicographical ordering.* Lexicographical order is very similar to alphabetical ordering, except that digits are compared instead of letters. We henceforth write all matchings in the form $(a_{1,1}\ a_{1,2}\ ,\ a_{2,1}\ a_{2,2}\ ,\ \cdots\ ,\ a_{e,1}\ a_{e,2})$ where $a_{i,1} < a_{i,2}$ for all i, and $a_{i,1} < a_{j,1}$ whenever $i < j$. We call this the *lexicographical form.* By creating and listing all matchings in lexicographic order, we ensure that no matching is listed twice.

There is another way that we can reduce the number of matching to be checked. A cyclic rotation of ρ (this is the rotation that was initially fixed) acts as a permutation on the edge ends of the map of a bouquet, and simply re-labels the edge ends by a cyclic shift. For example, the matchings (1 2 , 3 4 , 5 6 , 7 8) and (2 3 , 4 5 , 6 7 , 8 1) represent isomorphic maps (Figure 14).

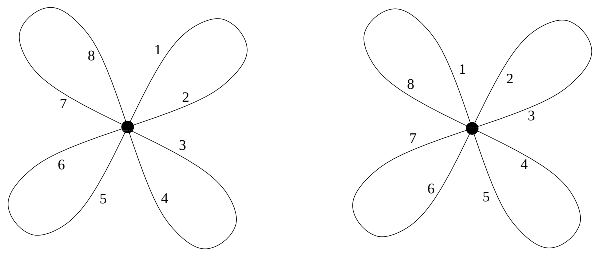

Figure 14: Two isomorphic matchings of maps of bouquets

The matching (2 3 , 4 5 , 6 7 , 8 1) is not in lexicographical form. It lexicographically should be written as (1 8 , 2 3 , 4 5 , 6 7). Hence the matchings (1 8 , 2 3 , 4 5 , 6 7) and (1 2 , 3 4 , 5 6 , 7 8) correspond to the same map. A given map will have up to $2n$ different matchings, depending on the rotational symmetries of the graph. To eliminate this redundancy, we generate all $2n$ matchings that correspond to a given rotation and we compare them. Only the smallest in lexicographical order is selected as the matching to represent the map. We call this representative the *canonical form*.

We wrote a procedure called *test canonical* which generates all of the cyclic shifts of the symbols $\{1, 2, 3, \dots, 2n\}$ (modulo $2n$) and uses them to generate a new matching which is equivalent to the original one up to a rotation of the bouquet. The two matchings are then compared, and this process is repeated for all $(2n - 1)$ possible other rotations of the bouquet. The original matching is in canonical form if it is no larger than any of the other matchings.

3.4 Counting the Faces of a Graph

In order to determine the genus of the surface in which a particular rotation of a graph embeds, it is necessary to count the number of faces in this map. If we have a drawing of this map in the plane then it is relatively easy to trace out each face. We do this by traversing sides of edges and we call this *walking*.

We first describe face-tracing geometrically. To trace out the faces we start at any vertex v_1 and select an edge e_1. We now walk along one side of that edge until we reach the vertex (v_2) at the other end of the edge. We now look for the next edge e_2 listed in the cycle in the permutation ρ corresponding to this vertex. At this stage we may have to choose between two edges that are adjacent to e_1 in ρ. We choose the one that we can now walk along without crossing e_1 (Figure 15). We call this turning point a *corner*. We proceed by walking along the side of e_2 to v_3, and so on.

If we continue this method of traversing along sides of edges we eventually will come back to our starting point. The circuit that we have traversed corresponds to a face in a cellular embedding of this map. If we have not walked along every side of every face, then we select an unvisited corner and we start a new circuit there. After we have walked along every side of each edge, we count the number of circuits completed and this gives the number of faces of the cellular embedding. We are now able to calculate the genus of the map of the graph. For example, consider Figure 16.

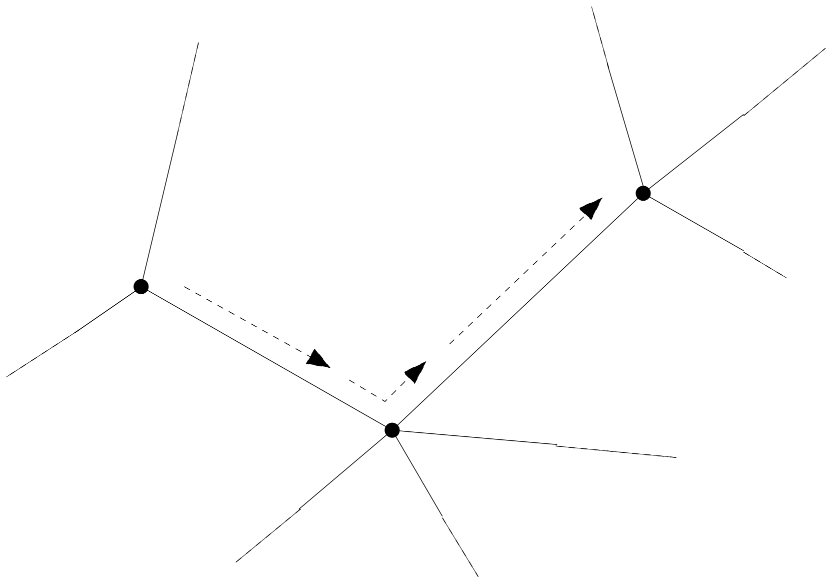

Figure 15: Tracing faces: selecting the next edge.

In this example we have a map of a graph with 4 vertices and 6 edges. In the lower part of the figure we show the same map with two circuits traced out, hence this map has 2 faces. Using Euler's formula ($v + f - e = 2 - 2g$) we find that the genus of this map is 1. Hence this map has a cellular embedding in the torus.

We have just described how one might visually count the faces of a map; however, since we are using a computer algorithm we need find an algorithm to count the number of faces for the map of a bouquet stored in matching form. To do this we first convert the matching to the neighbor form described earlier, which can be thought of as an array where the i^{th} element of the array contains the number j if and only if i and j are a pair of edge ends that form a loop.

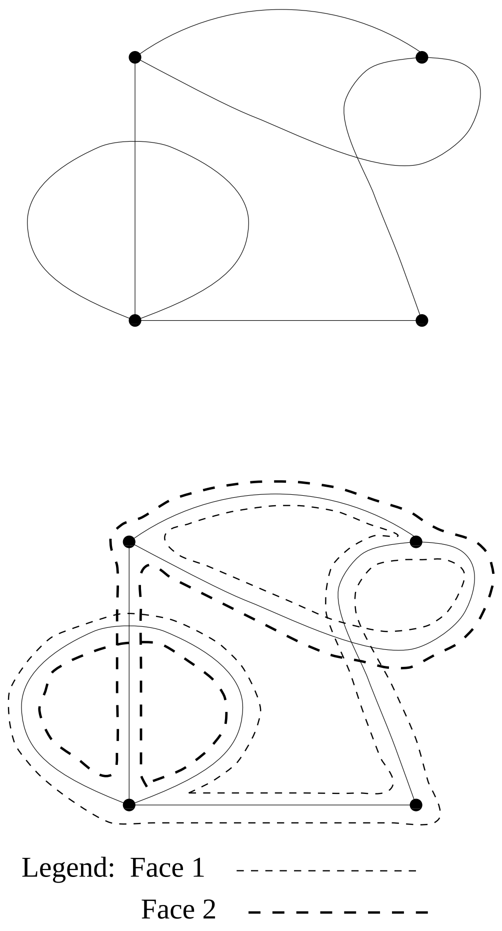

Figure 16: Tracing the faces of a map.

We can describe the faces in an algebraic manner by considering the rotation of a graph. If ρ is a rotation of a graph G and θ is an end swap, then the faces of G are the orbits of $\rho \circ \theta$. We now use the permutations from the neighbor form to count the faces. Here $\theta = N(x)$ is a function that returns the neighbor of x where x is an edge end, and $\rho : x \to x + 1$. To trace out a face in neighbor form, we simply start at an edge end (say x_1) and find its neighbor ($N(x_1)$), and add one to the neighbor to get the next edge end in our face circuit (so $x_2 = N(x_1) + 1$). If $x_i = 2e + 1$ then we set $x_i = 1$. Adding one corresponds to "turning a corner" in our walk along the edges. We now find the $N(x_2)$ and call it x_3. We continue in this fashion with $x_i = N(x_{i-1}) + 1$, finding neighbors and adding 1 each time. Eventually we return to the edge end $x_j = x_1$. If, at this point, we have traversed every edge twice (i.e. $j = 2e$) then our map has only one face. However, if there are some edges that have not been traversed then we select the smallest i such x_i was not visited, and we repeat the process. Each completed circuit corresponds to a face circuit, hence the number of circuits obtained by this algorithm corresponds to the number of faces in the map of the bouquet.

3.5 The Algorithm for Finding Spines

We now have the tools needed to describe the algorithm for finding all spines on the double torus and triple torus:

Step 1: We use Euler's formula to find all possible values of v, e and f for spines on the surface being considered. Next, we find the corresponding values of $v^\star$, $e^\star$ and $f^\star$ for the geometric duals of these spines. These geometric duals are always bouquets.

Step 2: For each possible value of $e^\star$ we recursively generate all matchings in lexicographical order. Each matching is stored in a one dimensional array. It is then tested to see if it corresponds to a spine. If so it is called a *winner*.

Step 3: Each time we generate a new matching we test to see if it is a canonical matching. To do this, we cyclicly shift ρ and find the new matching corresponding to the rotated map. The new matching and the old are then compared, and if the new one is smaller we go to Step 2 and generate the next matching. If the original is less than or equal to the cyclic shifts then we say it is canonical and proceed.

Step 4: We count the faces in the map corresponding to the matching while keeping track of the number of edges on the boundary of each face. We call this number the *face cycle length*. Since we have the requirement that $\delta \geq 3$ for the spines, the corresponding bouquets may not have face cycles of length 1 or 2. If the value of $f^\star$ corresponds to the number found in Step 1, then we have found a winner, a map of a bouquet whose dual is a spine. If $f^\star$ is different from the value found in Step 1, then the map must embed in a surface of a different genus.

The next algorithm refines the first. This algorithm, which we call the spawning algorithm is used to improve significantly the speed of the search for spines on orientable surfaces.

3.6 The Spawning Algorithm

In this section we describe a modification of our algorithm for finding spines called the *spawning algorithm.* This algorithm is markedly faster that the original algorithm because it dramatically reduces the number of rotations of the bouquets to be considered.

The principle behind the spawning algorithm is that "winning" rotations with $e+1$ edges can be built from "winning" rotations with e edges. We call this process *spawning.* In order to generate all winning bouquets on the double torus, we simply need to generate all of the winners with 4 edges (the smallest possible value for e) and spawn the rest. The original algorithm for finding spines can generate the winners for small values of e very quickly. Since the subsequent number of rotations to be tested is much smaller, we get a great improvement in the speed of our algorithm.

Spawning is based on the fact that the addition of an edge to a winning rotation with e edges results in a rotation with one more edge and either one more face or one less face. If it does result in a rotation with one more face, then Euler's formula is still satisfied and the resulting rotation will be a winner. Moreover, every winning rotation on $e+1$ edges arises in this way.

In order to spawn winners from winners, we simply take the winning rotations and add an edge in all possible ways, and then check if the resulting rotation is also a winner. The addition of an edge corresponds to the addition of two numbers to the matching form of the winners.

Since our winning rotations involve no face cycles of length 1 or 2, we need only insert the new edge in ways that do not create these restricted configurations. Face cycles of length 1 are simply loops, which are represented in matching form by a pair of numbers that differ by one $(*\,*\,,\ a\ (a+1)\ ,\ *\,*\,, \ldots\ *\,*\,,\ *\,*)$.Face cycles of length 2 are represented by matchings of the form $(*\,*\,,\ a\ b\ ,\ (a+1)\ (b-1)\ , \ldots\ *\,*\,,\ *\,*)$. The spawning program takes winning matchings and adds two more symbols, subject to these restrictions.

However, there are some problems with spawning that need to be addressed. A new winner (with $e+1$ edges) may be spawned from more than one winner (with e edges), and a winner (with e edges) may spawn the same winner (with $e+1$ edges) by adding an edge in two different ways. We deal with these problems by keeping track of how a map is spawned. If a map M_2 is spawned from a map M_1 then we say that M_1 is the *parent* of M_2. If M_3 can be obtained from M_2 by the deletion of an edge in M_2 (the inverse operation of spawning), then we say that M_3 is an *uncle* of M_2. In geneological terms, our problem is that a child may have many parents and a parent may have several children that are identical twins.

The spawning program uses a depth-first search to generate winners, with each new winner spawning more winners until the maximum length of matching is obtained. Each time that we spawn a new map, we do a face count. If the new map is not a winner, then we discard it, return to the parent, and spawn the next map.

However, if the new map is a winner, then we generate all possible uncles for this map. We now compare the uncles to the parent. If the parent is smaller, in lexicographical order, than all of the uncles then we proceed. If not then we again discard the child. This takes care of the problem of a winner being spawned from many sources. The problem of a winner spawning several isomorphic children is dealt with by temporarily storing all of the new children spawned in an array. This is only a temporary storage of data, but it allows us to determine if an isomorphic map has already been spawned at this level of the recursion.

The spawning program differs very little from the original spine generation program. The main difference is a procedure called SPAWNER which generates all of the spawned matching and calls all of the other procedures. Here are a few of the different procedures:

MAKE-CANONICAL: a procedure that converts a matching into canonical form. This is done by cyclicly shifting the matching in all possible ways and keeping the smallest form.

MAKE-UNCLE: a procedure that removes each edge (pair) from a partial matching.

MAKE-KID: a procedure that adds edges (pairs) to a partial matching, except edges that have ends with labels that differ by 1.

4 Results

We found that there are 160 spines on the double torus and 345,038 spines on the triple torus. In the following table we show how the spines are distributed in terms of numbers of edges for each of the first few orientable surfaces.

Surface ⇒	S_1	S_2	S_3
1 edge			
2 edges	1		
3 edges	1		
4 edges		4	
5 edges		21	
6 edges		45	131
7 edges		52	1841
8 edges		29	10883
9 edges		9	35448
10 edges			71145
11 edges			92225
12 edges			77737
13 edges			41308
14 edges			12594
15 edges			1726
total number	2	160	345038

Table 1: Number of spines with a fixed number of edges for the orientable surfaces of genus 1,2, and 3.

This algorithm works well for both the double torus and triple torus, but a "combinatorial explosion" makes it difficult to extend the result to surfaces of higher genus. As we see in the above table, the number of spines grows very quickly with genus.

The program was run on a Silicon Graphics Challenge-S (180 MHZ IP22 Processor) and generating the matchings for the triple torus, using the spawning program, took over eighteen hours.

5 Double Checks

Although our results depend heavily on computer calculations, we have found reasonable methods for verifying some of our results. We use a method that doesn't require any computer calculations, at least for the double torus. We also run the spawning algorithm on the double torus to verify that the two algorithms generate the same results when used for the same surface.

Our algorithm relies on the fact that we can bound the number of edges and vertices of potential spines using Euler's formula. We first generate all possibly non-simple candidate graphs with minimum degree at least 3 on these number of vertices and edges. Once we have a list of candidate graphs we can form all possible rotations for these graphs and then for each we can do a "face count" to find the surface in which the map has a 2-cellular embedding.

The easiest way to describe the details of this method is by way of an example. We proceed by manually finding all spines in the torus.

As we have done before, we can use Euler's formula to put bounds on v and e. Since the surface is S_1 the formula is $v + f - e = 0$ and since we are searching for graphs with just one face this formula becomes $e = v + 1$. Since we are searching for all topological minimal graphs with one face we know that $\delta(G) \geq 3$ where δ is the minimum degree of G. This implies that $e \geq 3v/2$. Combining these formulas yields

$$v + 1 \geq \frac{3v}{2} \implies v \leq 2,$$

so the two possibilities are: $v = 1$, $e = 2$ and $v = 2$, $e = 3$.

We try to find all possible maps with either $v = 1$, $e = 2$ or $v = 2$, $e = 3$ remembering that our graphs must be connected with $\delta \geq 3$. If $v = 1$, then there are only two rotations that yield maps that are not isomorphic. We can describe these two maps with their rotation systems or by their drawings. Their rotation systems are (1 2 , 3 4) and (1 3 , 2 4). Here are their corresponding drawings:

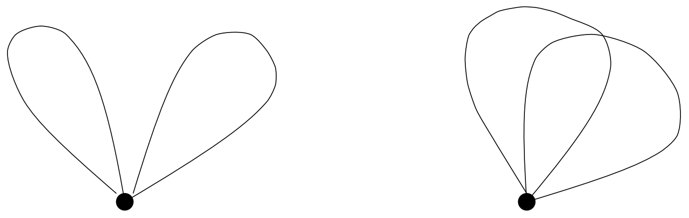

Figure 17: The two non-isomorphic maps of B_2

For $v = 2$, $e = 3$ the analysis is a little harder. We still have the restriction that our graphs must be connected with $\delta \geq 3$. If $v = 1$ we end up with two possible graphs:

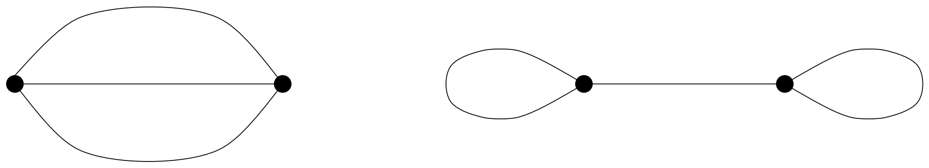

Figure 18: The two graphs with 2 vertices and 3 edges

However we still have not found all maps for these two graphs. In fact there are three (after accounting for isomorphisms of maps):

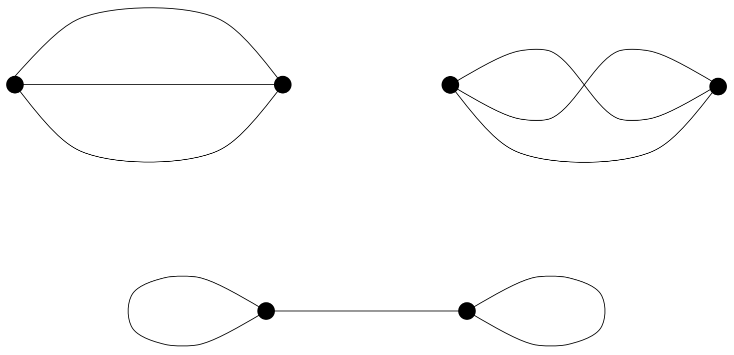

Figure 19: The three non-isomorphic maps for $v = 2$, $e = 3$

All that remains to be done is a face count. A face count is a procedure by which we determine the genus of surface in which G can be embedded. In this case the maps that embed in the torus are precisely those with 1 face.

Doing a face count on the five candidate maps for the torus yields two *winners* (graphs with one face). They are shown in Figure 20, both as graphs in the plane and as they are embedded in the torus. These correspond to the two identification maps given in the introduction (Figure 2 and Figure 3).

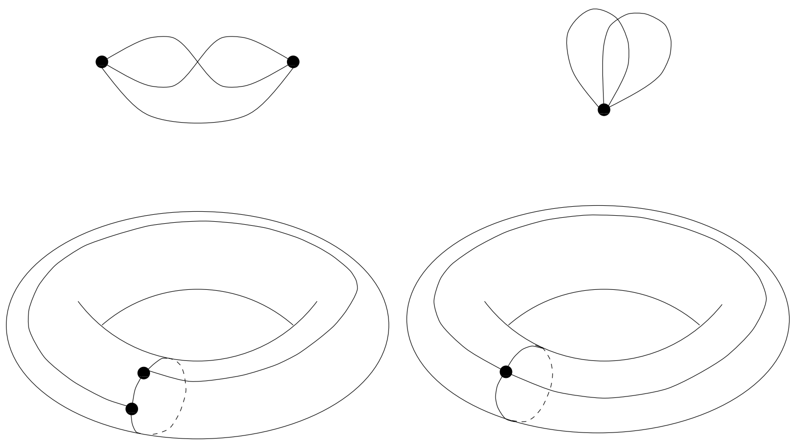

Figure 20: The spines of the torus

We have now demonstrated the manual search method for the torus and we can discuss how the same method can be applied to the double torus (S_2). Recall that the range of possible values for v is given by

$$1 \leq v \leq 4g - 2$$

and the range of possible values for e is given by

$$2g \leq e \leq 6g - 3$$

So on a surface of genus g we get $v \in \{1, 2, ..., 4g - 2\}$ and $e \in \{2g, 2g + 1, ..., 6g - 3\}$. Applying this result to the double torus gives

$$v \in \{1, 2, 3, 4, 5, 6\} \text{ and } e = v + 3.$$

The spines are connected and each one contains a subgraph which is a simple connected graph. If we start with a complete list of all simple, connected graphs with at most 6 vertices, then we can generate a finite list of all graphs that are potential spines. Fortunately, Harary has generated a list of all connected simple graphs on up to six vertices [6]. Although we are not searching for only simple graphs, all of the graphs that we seek contain simple graphs that are on Harary's list. We may consider all of the graphs on Harary's list and add in either loops or multiple edges in all possible ways and then check the condition that $\delta \geq 3$.

We managed to reduce Harary's list of graphs to 52 candidate graphs by selecting graphs that could possibly satisfy the $\delta \geq 3$ condition. For each of these graphs, we had to find all possible maps for each and do a face count to see if there were any maps that embed in the double torus. This task was very tedious and time-consuming and was abandoned for the much more efficient computer search method. However, we do present a portion of this search for the case $v = 6$, $e = 9$ because it provides a valuable double check for the computer algorithm.

Recall that there are $18!/(9!2^9) = 34,459,425$ different matchings for the bouquets with 9 edges. Our computer search came up with 9 winners, that is, 9 non-isomorphic maps of bouquets whose duals are spines. By using the manual search method we came up with precisely the same 9 spines. These spines are shown in Figure 21. We found these 9 spines by doing the following procedure.

First, we used Harary's list to find all connected graphs on six vertices with fewer

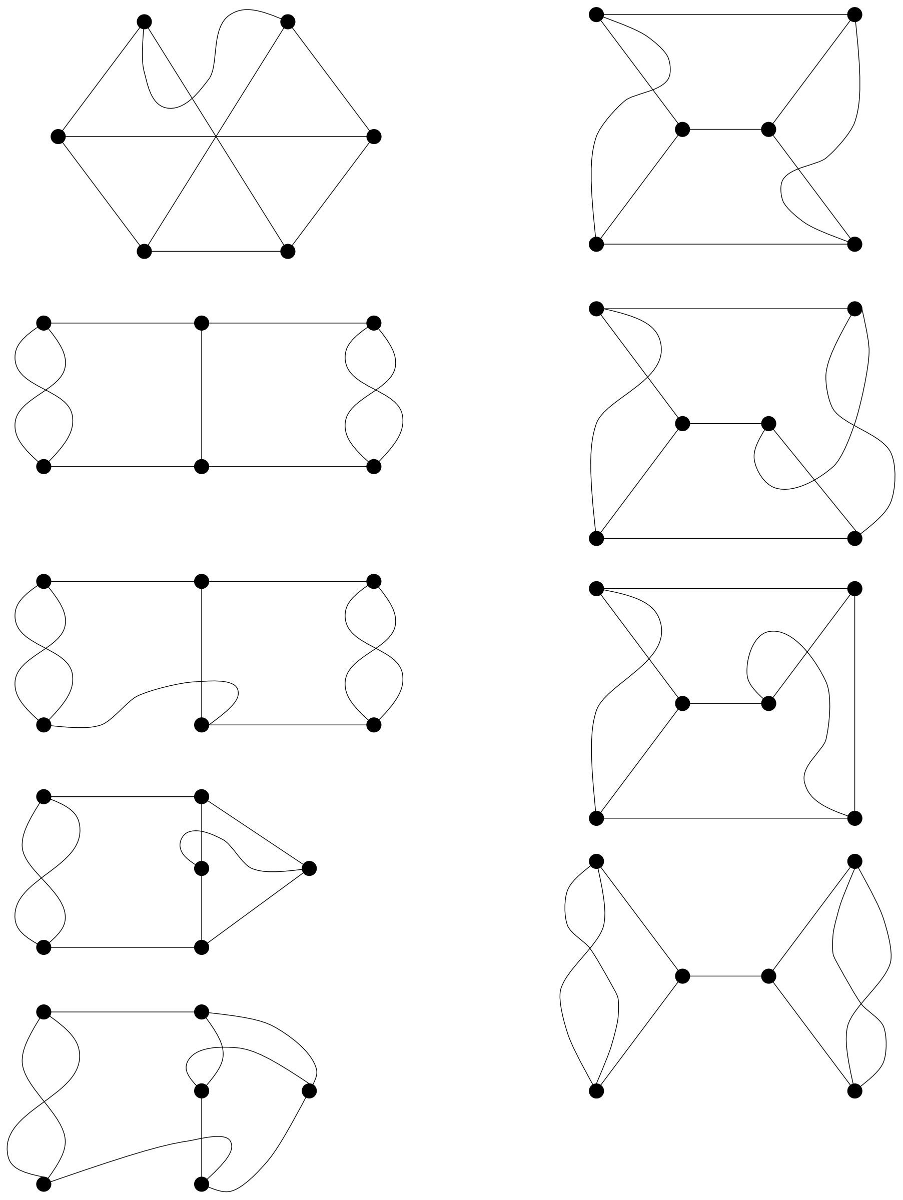

Figure 21: The nine spines with $v = 6$ on S_2

than 10 edges. These graphs had from 5 to 9 edges. For those with 9 edges we could easily determine which ones satisfied the condition that $\delta \geq 3$. However, for those with less than 9 edges, we had to examine all possible ways to add in loops and multiple edges in order to satisfy the $\delta \geq 3$ condition. This task was simplified by the fact that most of these graphs could never satisfy the condition that $\delta \geq 3$, even with the additional edges. After this step we were left with only 6 graphs. The next step was to generate all non-isomorphic rotations for each of these graphs. The high degree of symmetry in these graphs greatly reduced the number of cases that needed to be checked. Finally, for each of the remaining candidate maps, a face count was done. Since we were dealing with spines, we were looking for maps with just one face. It was often fairly easy to determine that a graph had more than one face by visual inspection of the map. The 6 remaining graphs yielded 9 non-isomorphic maps.

To show how to convert from a drawing of a spine to the matching form we give the following example (Figure 22):

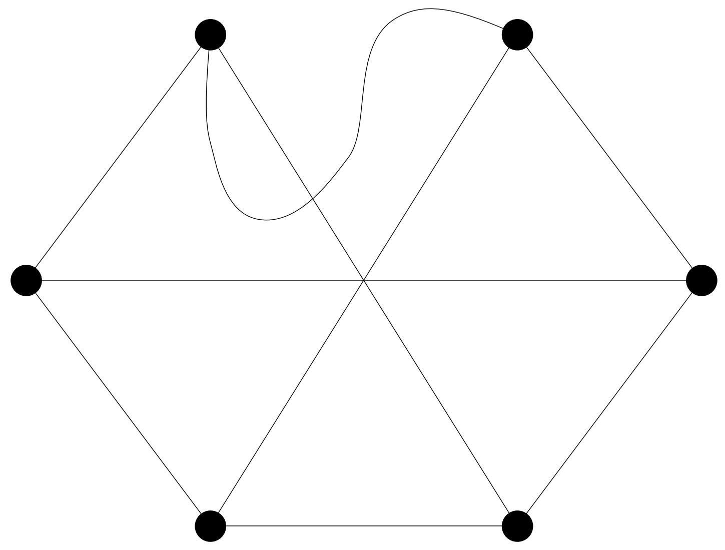

Figure 22: A spine of the double torus

With this labeling, we can trace out a face cycle of length 18, starting on the lower side of the edge labeled 1. We get the face cycle:

$$(1, 7, 3, 9, 5, 8, 2, 3, 4, 5, 6, 1, 8, 4, 7, 6, 9, 2)$$

The edge labeled 1 appears in the 1^{st} and 12^{th} position in the cycle. This corresponds to the partial matching (1 12). Similarly, the symbol 7 appears in the 2^{nd} and 15^{th} positions, giving the partial matching (2 15). In the same manner, we can generate all 9 partial matchings to form the matching:

$$(1\ 12\ ,\ 2\ 15\ ,\ 3\ 8\ ,\ 4\ 17\ ,\ 5\ 10\ ,\ 6\ 13\ ,\ 7\ 18\ ,\ 9\ 14\ ,\ 11\ 16).$$

However, this matching is not in canonical form. A little work shows that the canonical form is achieved by a cyclic shift achieved by subtracting 4 from all of the symbols (where $0 = 18, -1 = 17, -2 = 16$, etc.) Hence we obtain the matching:

$$(1\ 6\ ,\ 2\ 9\ ,\ 3\ 14\ ,\ 4\ 17\ ,\ 5\ 10\ ,\ 7\ 12\ ,\ 8\ 15\ ,\ 11\ 16\ ,\ 13\ 18)$$

which is the last of the 160 matchings obtained from the computer algorithms.

In addition to the double check method described above, we ran the spawning algorithm on for the torus and obtained the same two graphs that we obtained by hand.

6 Conclusion

The algorithms described in this thesis have only been used to find the spines of the double torus and triple torus. These algorithms can also be used to find spines on other orientable surfaces. However, the number of spines grows very quickly with genus. This prevents us from easily counting the number of spines on surfaces of high genus.

The problem of finding all spines on the non-orientable surfaces seems to be very similar to the problem in the orientable case. We are attempting to modify the original spine generating algorithms so that we can find the spines on the surfaces with a small number of cross-caps.

A possible application for these spines would be in the problem of finding crossing numbers for pairs of graphs on surfaces of genus at least 1. The *joint crossing number* of a pair of graph in a surface Σ is the minimum number of crossings that we can attain between edges of G_1 and G_2 among all simultaneous embeddings of G_1 and G_2 in Σ. In general, the finding of joint crossing numbers seems to be very hard. Knowing the spines of a surface would allow us to find a lower bound on the joint crossing number for a pair of graphs embedded in the surface. This is because every graph embedded in this surface must have a subgraph homeomorphic to one of these spines. This would be a time-consuming process. For example, on the double torus we would need to look at all $160^2 = 25600$ ways of combining two of the spines from this list, and then calculate the crossing number for this pair of graphs.

Another direction for future research could be to find a closed form expression that would give the precise number of spines for any surface, or generating functions or recurrence relations that determine these numbers.

References

[1] D. Archdeacon, Topological Graph Theory: A Survey, *Congressus Numerantium* **115** (1996) 5-54.

[2] J.A. Bondy and U.S.R. Murty, "Graph Theory with Applications", American Elsevier, New York (1976).

[3] R. Gould, "Graph Theory", Benjamin/Cummings, Menlo Park (1988).

[4] J.L. Gross and T.W. Tucker, "Topological Graph Theory", Wiley, New York (1987).

[5] J.L. Gross, D.P.Robbins, T.W. Tucker, Genus Distribution for Bouquets of Circles, *J. of Combin. Th. Ser. B* **47** (1989) 292-306.

[6] F. Harary, "Graph Theory", Addison-Wesley, Reading, Mass. (1969).

[7] G. Ringel, "Map Color Theorem", Springer-Verlag, Heidelberg (1974).

[8] R.J. Wilson, "Introduction to Graph Theory", Longman, London (1972).

A Pascal Source Code for Spine Program

Here is the Pascal source code for the program that generates duals of spines on the double torus.

```
{ Program for finding all graphs that embed in
  the double torus with one face. We will call these
  graphs spines.

  By: Ian Stobert
  Written: Sept. 12, 1996
  Last modified: March 15, 1997
  ------------------------------------------------}

program Find_Spine;

const
    {Set the genus of the surface in question,
     in this case the double torus}
    genus=2;

type
    holder = array[1..50] of integer;

var
    match,b,temp: holder;
    depth,i,edges: integer;
    numcanon,bigcount: integer;

{ ***********************************************}
{ *************** Procedure Pack *****************}

{ This procedure takes a matching and re-orders it so that
  if there is a pair (a,b) that corresponds to an edge in the matching,
  then the smaller number of the two will always be listed first. The
  pairs are re-ordered in the matching so that the pair (a,b) will come
  before the pair (c,d) if and only if  min(a,b) < min(c,d)}
```

```
procedure pack (edges: integer;var temp:holder);

type
    holder = array[1..50] of integer;

var

   newtemp :holder;
   place,count,e,f:integer;

begin
   for e := 1 to (2*edges) do temp[e] := temp[e]+1;
   for f := 1 to (edges) do
   begin
      newtemp[temp[(2*f)-1]] := temp[2*f];
      newtemp[temp[2*f]] := temp[(2*f)-1];
   end;
   count := 1;
   place := 1;
   repeat
   if newtemp[place] > place then
       begin
       temp[(2*count)-1] := place - 1;
       temp[2*count]   := newtemp[place] - 1;
       count := count + 1;
       end;
   place:= place + 1;
   until count = edges + 1;
end;

{ ****************** Procedure Compare ***************}

{ Compares two matchings to see which is lower in
  lexicographical order. If the first matching is
  bigger than the second then the boolean variable
  "bigger" is set to true. Otherwise "bigger" is false.}

procedure compare (match,temp:holder;edges:integer;
                   var bigger:boolean);
```

```
var
    done:boolean;
    place:integer;

begin
     done := false;
     place:= 1;
     repeat
         if match[place] > temp[place] then
            begin
            bigger := true;
            done := true;
            end;
         if match[place] < temp[place] then
            begin
            bigger := false;
            done := true;
            end;
         place := place + 1;
     until done or (place = (2*edges + 1));
     if place = (2*edges + 1) then bigger:= false;
end;


{ ********* Procedure TestCanonical **************

 {This procedure takes a matching (which corresponds to a
  rotation on the bouquet) and rotates it by all possible
  values (by adding a constant to all values modulo the number of
  edge ends). The newly formed matchings (called temp)
  are then compared with the original to see which is lower
  in lexicographical ordering. If the original is less than
  or equal to all of "rotated versions" then
  we say that it is canonical and we set the boolean
  variable "canonical"="true"}


procedure testcanon (match,temp:holder;edges:integer;
                     var canonical:boolean);

var
    bigger:boolean;
    i,place: integer;

begin
    {default canonical = true}
    bigger := false;

    for i:= 1 to (2*edges-1) do
         begin
         for place:=1 to (2*edges) do
            temp[place]:=(match[place]+i)mod(2*edges);

         {now lexicographically order temp}
         pack (edges,temp);

         {returns bigger = true if match > temp}
         compare (match,temp,edges,bigger);
         if bigger = true then canonical := false;
    end;
end;




{ **************** Procedure Countface *******************}

{ This procedure takes a canonical matching and checks to see
  if it embeds in the surface in question with the right number of
  faces (By right number of faces we mean the number of faces needed
  for this rotation to have a geometric dual that is a spine).
  This is done by creating the neighbour form of the rotation
  and then tracing out all of the faces, which are then counted.
  We put the number of faces into Euler's formula and
  calculate the genus of the rotation. If the genus is correct
  (i.e. matches the genus prescribed in the main program) then
  we have a "winner", a rotation of a bouquet that has a two-cellular
  embedding in the surface in question. This corresponds to one of the
  spines that we are seeking. If a winner is found, it is written to
  the output in matching form.}

procedure countface(match: holder; edges:integer);

type
    holder = array[1..50] of integer;

var

   newtemp,storelgth :holder;
   number_of_faces,store,f,g,h,s,t,cycle_length:integer;
   done,winner : boolean;

begin
   for h := 1 to (2*edges) do match[h] := match[h] + 1;
   for f := 1 to (edges) do

   {create the neighbour form of the rotation from the matching form}
   begin
      storelgth [2*f-1] := -1;
      storelgth [2*f] := -1;
      newtemp[match[(2*f)-1]] := match[2*f];
      newtemp[match[2*f]] := match[(2*f)-1];
   end;

   number_of_faces:= 0;

   {trace out the faces counting their length}
   for g := 1 to (2*edges) do
   begin
       done := false;
       store := g;
       cycle_length := 0;
       repeat
         cycle_length := cycle_length + 1;
         store :=  ((newtemp[store]) mod (2*edges) +1);
         if store < g then done := true;
         if store = g then
            begin
```

```
               number_of_faces := number_of_faces + 1;
               storelgth [number_of_faces] := cycle_length;
               done := true;
             end;
          until done = true
   end;

   winner := false;
   if number_of_faces = (edges+1-(2*genus)) then
       begin
       winner := true;
           for s := 1 to number_of_faces do
           begin
           {check to make sure that all face cycles were of length
            at least 3}
           if storelgth[s] < 3 then winner := false;
           end;
       if winner = true then
          begin
          for t := 1 to 2*edges do write (match[t]:3);
          writeln;
          end;
       end;
end;

{ ********** Get-Edge procedure **********************************}

{ This procedure builds matchings by adding edges.
  Once it builds a matching it sends it to the procedure
  called CHECKCANONICAL to see if it is in canonical form.
  If it passes that check then the matching is sent to a procedure
  called COUNTFACE to see if this particular matching corresponds
  to a rotation which has a cellular embedding in the surface in
  question. This procedure is a recursive loop.}

procedure getedge (match,temp,b:holder;depth,edges:integer);

type
   holder = array[1..50] of integer;
```

```
var
   lim,x,y,z,m,f,start: integer;
   c : holder;
   canonical,quit :boolean;

begin        {procedure getedge}

    {The next part limits the scope of the search on first matching}
    if depth = 1 then
       begin
       lim := edges;
       start := 2;
       end
    else
       begin
       lim:= (2*edges) - (2*depth) + 1;
       start := 1;
       end;

    {This is the main loop in the procedure}

    for x := start to lim do
    begin   {main loop}
        for z:=1 to (2*edges) do c[z]:=b[z]; {temporarily stores b-array}

        {now assign new edge}

        match[2*depth]:=b[x];
        for m:= x to (2*edges) do b[m] := b[m+1];
        match[2*depth+1]:=b[1];
        for y:= 1 to (2*edges) do b[y] := b[y+1];

        {if we have a new matching we do the next part}
{note: "depth" measure the depth of the recursion
               which relates to the number of edges selected}

        if (2*depth)+1 = 2*edges - 1 then
           begin  {check canonical}
              match[2*edges] := b[1];
              canonical := true;
```

```
            testcanon (match,temp,edges,canonical);
            if canonical = true then
                begin  {countface}
                countface (match,edges); {Call the procedure
                                          that checks to see
                                          if the bouquet is
                                          a winner}
                numcanon := numcanon + 1;
                end;   {countface}
            bigcount:=bigcount+1;
         end    {check canonical}


      else      {if we don't have a new matching
                 we go a layer further into the recursive loop}

         begin
            quit := false;
            if depth > 1 then
               begin
               if (match[2]-match[1]) >
               (match[2*depth]-match[2*depth-1]) then quit := true;
               end;
            if quit = false then
                begin
                depth := depth + 1;
                getedge (match,temp,b,depth,edges);
                depth := depth - 1;
                end;
         end;



      for f:=1 to (2*edges) do b[f]:=c[f];
   end;   {main loop}
end;     {procedure getedge}


{ *****************************************************}
{ ******************** Main Program *******************}
{ *****************************************************}

begin
     {Use the bounds derived from Euler's Formula
      to put bounds on the number of edges}
     for edges := (2*genus) to (6*genus-3) do

     begin

     {this loop initializes the variables}
     i := 0;
     repeat
        i := i + 1;
        match[i] := -1;
        b[i] := i;
        temp[i] := -1;
     until i = 2*edges + 2;
     bigcount:=0;
     numcanon:=0;
     match[1] := 0;
     depth := 1;

     {now call the procedure GETEDGE which contains a recursive
      loop which generates all matchings on the bouquet with the
      prescribed number of edges. All other procedures are called
      from GETEDGE}
     getedge (match,temp,b,depth,edges);

     end;
end.
```

B Pascal Source Code for Spawning Program

Here is the source code for the spawning program.

```
{ Program for finding all graphs that embed on the
  g-torus with one face. We will call these
  graphs spines.

  By: Ian Stobert
  Written: Nov. 5, 1996
  Last modified: March 15, 1997
  -------------------------------------------------}

program Spawner;

type
    holder = array[1..50] of Integer;

var
    a,b,temp,countwin: holder;
    depth,i,j,edges, genus : Integer;
    numcanon,bigcount: integer;
    dumb :char;

{ ********** Procedure Pack ********}
procedure pack (edges: integer;var temp:holder);

{ Re-orders a matching into ordered pairs.}

type
    holder = array[1..50] of Integer;

var

   neighbour :holder;
   place,count,f:integer;

begin
   for f := 1 to (edges) do
   begin
      neighbour[temp[(2*f)-1]] := temp[2*f];
      neighbour[temp[2*f]] := temp[(2*f)-1];
   end;
   count := 1;
   place := 1;
   repeat
   if neighbour[place] > place then
       begin
       temp[(2*count)-1] := place;
       temp[2*count]   := neighbour[place];
       count := count + 1;
       end;
   place:= place + 1;
   until count = edges + 1;
end;

{ ********** Procedure Compare *****}

procedure compare (a,temp:holder;edges:integer;
                   var bigger,same:boolean);

{ Compares two matchings to see which is lower in
  lexicographical order}

{returns bigger = true if the first is bigger}
{returns same = true if a tie}

var
    done:boolean;
    place:integer;

begin
     same := false;
     done := false;
     place:= 1;
     repeat
         if a[place] > temp[place] then
            begin
            bigger := true;
            done := true;
            end;
         if a[place] < temp[place] then
            begin
            bigger := false;
            done := true;
            end;
         place := place + 1;
     until done or (place = (2*edges + 1));
     if place = (2*edges + 1) then bigger:= false;
     if place = (2*edges + 1) then same := true;
end;


{ ********** Procedure TestCanonical ***************

 {This procedure take a matching and checks to see if it
  is in its lowest possible lexicographical form}

procedure testcanon (a,temp:holder;edges:integer;
                     var canonical:boolean);



var
    bigger,same:boolean;
    i,j,place: integer;

begin
   bigger := false;
   for i:= 1 to (2*edges-1) do
     begin
     for place:=1 to (2*edges) do
         temp[place]:=(a[place]+i)mod(2*edges);
     for j := 1 to (2*edges) do
         if temp[j]=0 then temp[j]:=temp[j]+(2*edges);

     pack (edges,temp);
     {lexicographically orders temp}

     compare (a,temp,edges,bigger,same);
     {returns bigger = true if a > temp}

     if bigger = true then canonical := false;
     end;
end;

{ ********** Procedure MAKECanonical ***************

 {This procedure take a matching and puts it in its
 lowest possible lexicographical form}

procedure makecanon (var a:holder;
                     temp:holder;edges:integer;
                     var canonical:boolean);

var
    bigger,same:boolean;
    i,j,t,place,numchange: integer;

begin
 bigger := false;
 repeat
    numchange :=0;
    for i:= 1 to (2*edges-1) do
    begin
       for place:=1 to (2*edges) do
           temp[place]:=(a[place]+i)mod(2*edges);
       for j := 1 to (2*edges) do
           if temp[j]=0 then temp[j]:=temp[j]+(2*edges);

       pack (edges,temp);
       {lexicographically orders temp}

       compare (a,temp,edges,bigger,same);
       {returns bigger=true if a>temp}

       if bigger = true then begin
          canonical := false;
          numchange := numchange + 1;
          for t:= 1 to (2*edges) do a[t] := temp[t];
       end {if};
    end;
 until numchange=0;
end;


{ ********** Procedure Make_Kid *************}

procedure make_kid (parent:holder;j,k,edges  :integer;
                    var kid                  :holder);


{This procedure takes a matching called parent and inserts edge
 (j,k) to generate kid. Outputs a packed kid with symbols 1..2n.}

var
     h,i   :integer;

begin
     for h:= 1 to (2*edges) do kid[h] := parent[h];
     for i:= 1 to (2*edges) do begin
         if (parent[i] >= j) then kid[i]:=kid[i]+1;
         if (parent[i] >= (k-1)) then kid[i]:=kid[i]+1;
     end;
     kid[2*edges+1] := j;
     kid[2*edges+2] := k;
     pack (edges+1,kid);
end;


{ ************* Procedure Make_Uncle ************** }

procedure make_uncle (var kid:holder;
                      var uncle:holder;
                      p:integer;
                      edges:integer);

{ This procedure takes kid and deletes the p-th matching (edge)
  to form uncle}
```

```
var
   a,b,i,j  :integer;

begin
     a:=kid[2*p-1];
     b:=kid[2*p];
     for i:= 1 to (2*edges-2) do begin
         if i < (2*p-1) then uncle[i] := kid[i];
         if i >= (2*p-1) then uncle[i] := kid[i+2];
     end;
         {now re-adjust the digits of uncle}
     for j:= 1 to (2*edges-2) do begin
         if (uncle[j] > a) and (uncle[j] < b)
             then uncle[j] := uncle[j] - 1;
         if (uncle[j] > b) then uncle[j] := uncle[j] - 2;
     end;
end;


{ ********** Procedure CountfaceTWOK ********}

procedure countfacetwo(a: holder;
                       edges,genus:integer;
                       var countwin:holder;
                       var foundwin :boolean);

type
    holder = array[1..50] of Integer;

var

   newtemp,storelgth :holder;
   count,store,f,g,s,p,cntcylgth:integer;
   done,winner : boolean;

begin
   for p := 1 to (2*edges+2) do newtemp[p] := 0;
   for f := 1 to (edges) do
   begin
      storelgth [2*f-1] := -1;
      storelgth [2*f] := -1;
      newtemp[a[(2*f)-1]] := a[2*f];
      newtemp[a[2*f]] := a[(2*f)-1];
   end;
   count:= 0;
   for g := 1 to (2*edges) do
   begin
       done := false;
       store := g;
       cntcylgth := 0;
       repeat
         cntcylgth := cntcylgth + 1;
         store :=  ((newtemp[store]) mod (2*edges) +1);
         if store < g then done := true;
         if store = g then
            begin
              count := count + 1;
              storelgth [count] := cntcylgth;
              done := true;
            end;
       until done = true
   end;

   winner := false;
   if count = (edges + 1 - (2*genus))  then
       begin
       winner := true;
           for s := 1 to count do
           begin
           if storelgth[s] < 3 then winner := false;
           end;
       if winner = true then
          begin
          foundwin := true;
          end;
       end;
end;


{ ********** Procedure Spawn ************}
```

```
procedure spawn(a: holder; edges,genus :integer;
                var countwin : holder);

type
    listofholder = array[1..2000] of holder;

var
    store                                              :listofholder;
    kid,uncle,parent,temp                              :holder;
    r,n,m,j,k,i,q,s,p                                  :integer;
    kid_edges,uncle_edges                              :integer;
    foundwin,done,same,bigger,canonical,go_on          :boolean;

begin
   for r:= 1 to (2*edges) do begin
      parent[r] := a[r];
      temp[r] := 0;
   end {for_r};
   for n:= 1 to (edges*edges*2) do begin
      for m := 1 to 50 do begin
         store [n,m] := -1;
      end;
   end;
   uncle_edges :=edges;
   kid_edges:=edges+1;
   for j := 1 to (2*edges) do begin
      for k:= (j + 2) to (2*edges+2) do begin
      make_kid(parent,j,k,edges,kid);
      makecanon(kid,temp,kid_edges,canonical);
      foundwin:=false;
      countfacetwo(kid,kid_edges,genus,countwin,foundwin);
        if foundwin then begin
            go_on := true;

            {this section compares a parent
             with the uncles}

            for p:= 1 to kid_edges do begin
                 make_uncle(kid,uncle,p,kid_edges);
```

```
              makecanon(uncle,temp,uncle_edges,canonical);
              bigger := false;
              foundwin := false;
              countfacetwo(uncle,uncle_edges,genus,countwin,foundwin);
              if foundwin
                 then compare(parent,uncle,uncle_edges,bigger,same);
              if bigger then go_on := false;
         end;
         if go_on then begin
            done := false;
            i:=0;
            repeat
               i:=i+1;
               same:=false;
               compare (kid,store[i],kid_edges,bigger,same);
               if (store[i,1]=-1) then begin
                   store[i]:=kid;
                   done:=true;
               end {if};
               if same then done := true;
            until done;
         end {go_on};
      end {foundwin};
   end {for_k};
end {for_j};
q:=1;
while (store[q,1] <> -1) do begin
   countwin[kid_edges] := countwin[kid_edges] +1;


   {this version only prints winners with the maximum
    and minimum number of edges. The other winners
    are simply counted}

   if kid_edges=(6*genus-3) then begin
      for s := 1 to (2*kid_edges) do begin
           write (store[q,s]:3);
       end;
   writeln;
   end;
```

```
      if  (kid_edges < (6*genus-3)) then begin
       spawn(store[q],kid_edges,genus,countwin);
      end {if};
      q := q+1;
   end {while};
end;

{ ********** Procedure Countface ********}

procedure countface(a: holder;
                    edges,genus:integer;
                    var countwin:holder);

type
    holder = array[1..50] of Integer;

var

   newtemp,storelgth :holder;
   count,store,f,g,s,t,cntcylgth:integer;
   done,winner : boolean;

begin
   for f := 1 to (edges) do
   begin
      storelgth [2*f-1] := -1;
      storelgth [2*f] := -1;
      newtemp[a[(2*f)-1]] := a[2*f];
      newtemp[a[2*f]] := a[(2*f)-1];
   end;
   count:= 0;
   for g := 1 to (2*edges) do
   begin
       done := false;
       store := g;
       cntcylgth := 0;
       repeat
         cntcylgth := cntcylgth + 1;
         store :=  ((newtemp[store]) mod (2*edges) +1);
         if store < g then done := true;
         if store = g then
            begin
              count := count + 1;
              storelgth [count] := cntcylgth;
              done := true;
            end;
         until done = true
   end;

   winner := false;
   if count = (edges + 1 - (2*genus))  then
       begin
       winner := true;
           for s := 1 to count do
           begin
           if storelgth[s] < 3 then winner := false;
           end;
       if winner = true then
          begin
          for t := 1 to 2*edges do write (a[t]:3);
  writeln;
          spawn (a,edges,genus,countwin);
          countwin[edges] := countwin[edges] + 1;
          end;
       end;
end;

{ ********** Get-Edge procedure ****}

procedure getedge (a,temp,b:holder;
                  depth,edges,genus:integer;
                  var countwin:holder);

type
   holder = array[1..50] of Integer;

var
   lim,x,y,z,m  : integer;
   c            : holder;
   canonical    : boolean;
```

```
begin
    lim:= (2*edges) - (2*depth) + 1;
    for x := 1 to lim do
    begin
        for z:=1 to (2*edges) do c[z]:=b[z];
        a[2*depth]:=b[x];
        for m:= x to lim do b[m] := b[m+1];
        a[2*depth+1]:=b[1];
        for y:= 1 to lim do b[y] := b[y+1];
        if (2*depth)+1 = 2*edges - 1 then
           begin
              a[2*edges] := b[1];
              canonical := true;
              testcanon (a,temp,edges,canonical);
              if canonical = true then
              begin
                 countface (a,edges,genus,countwin);
                 numcanon := numcanon + 1;
              end;
              bigcount:=bigcount+1;
           end
        else
           begin
               depth := depth + 1;
               getedge (a,temp,b,depth,edges,genus,countwin);
               depth := depth - 1;
           end;
        for m:=1 to (2*edges) do b[m]:=c[m];
    end;
end;


{ **********************************}
{ ********** Main Program **********}
{ **********************************}

begin
     genus := 2;
     edges := 4;
     i := 0;
     repeat
countwin[i]:=0;
        i := i + 1;
        a[i] := -1;
        b[i] := i+1;
        temp[i] := -1;
     until i = 2*(6*genus-3) + 2;
     bigcount:=0;
     numcanon:=0;
     a[1] := 1;
     depth := 1;

     getedge (a,temp,b,depth,edges,genus,countwin);

     writeln;
     writeln ('The distribution of the winners was as follows');
     writeln;
     for j := (2*genus) to (6*genus-3) do begin
          write ('for n = ',j:3,' there were : ',countwin[j]);
        writeln;
     end {if_j};
end.

{ ****** end of program ******* }
```

C Bouquets on the Double Torus in Matching Form

In this appendix we give a list of the matchings of the embedded bouquets that correspond to the 160 spines of the double torus. To find the identification map corresponding to one of these matchings, do the following:

1) Draw a vertex with $2e$ line segments radiating outward, like spokes of a wheel (e is the number of edges). Label them in cyclic order with the symbols $(1, 2, 3, \ldots, 2e)$.

2) Take the pairs in the matching and connect the corresponding line segments. This should create e loops; now we have a map of a bouquet.

3) Trace out the face cycles of the bouquet, keeping track of all edge ends visited (in order). Each face of the bouquet should thus generate a cyclic permutation of the symbols $(1, 2, 3, \ldots, 2e)$ with each symbol appearing up to two times in the permutation. Each cycle should start and finish with the same symbol.

4) When finished Step 3, there should be y cycles corresponding to the y faces of the embedded bouquet or the y vertices of the spine. The cycles from Step 3 give the rotations at each of the vertices in the spine. From these rotations it is easy to find a drawing of the spine.

4 Edges:

1 3 , 2 4 , 5 7 , 6 8
1 3 , 2 5 , 4 7 , 6 8
1 3 , 2 6 , 4 7 , 5 8
1 5 , 2 6 , 3 7 , 4 8

5 Edges:

1 3 , 2 4 , 5 7 , 6 9 , 8 10
1 3 , 2 4 , 5 8 , 6 9 , 7 10
1 3 , 2 4 , 5 10 , 6 8 , 7 9
1 3 , 2 5 , 4 7 , 6 9 , 8 10
1 3 , 2 5 , 4 8 , 6 9 , 7 10
1 3 , 2 5 , 4 9 , 6 8 , 7 10
1 3 , 2 6 , 4 7 , 5 9 , 8 10
1 3 , 2 6 , 4 8 , 5 9 , 7 10
1 3 , 2 6 , 4 8 , 5 10 , 7 9
1 3 , 2 6 , 4 9 , 5 7 , 8 10
1 3 , 2 7 , 4 8 , 5 9 , 6 10
1 3 , 2 7 , 4 9 , 5 10 , 6 8
1 3 , 2 8 , 4 7 , 5 9 , 6 10
1 3 , 2 9 , 4 7 , 5 8 , 6 10
1 3 , 2 9 , 4 7 , 5 10 , 6 8
1 4 , 2 5 , 3 8 , 6 9 , 7 10
1 4 , 2 6 , 3 8 , 5 9 , 7 10
1 4 , 2 6 , 3 9 , 5 8 , 7 10
1 4 , 2 7 , 3 8 , 5 9 , 6 10
1 4 , 2 9 , 3 6 , 5 8 , 7 10
1 6 , 2 7 , 3 8 , 4 9 , 5 10

6 Edges:

1 3 , 2 4 , 5 8 , 6 10 , 7 11 , 9 12
1 3 , 2 4 , 5 12 , 6 8 , 7 10 , 9 11
1 3 , 2 4 , 5 12 , 6 9 , 7 10 , 8 11
1 3 , 2 5 , 4 6 , 7 9 , 8 11 , 10 12
1 3 , 2 5 , 4 6 , 7 10 , 8 11 , 9 12
1 3 , 2 5 , 4 8 , 6 10 , 7 11 , 9 12
1 3 , 2 5 , 4 9 , 6 8 , 7 11 , 10 12
1 3 , 2 5 , 4 10 , 6 9 , 7 11 , 8 12

1 3 , 2 6 , 4 8 , 5 10 , 7 11 , 9 12
1 3 , 2 6 , 4 8 , 5 11 , 7 10 , 9 12
1 3 , 2 6 , 4 9 , 5 7 , 8 11 , 10 12
1 3 , 2 6 , 4 9 , 5 11 , 7 10 , 8 12
1 3 , 2 6 , 4 10 , 5 7 , 8 11 , 9 12
1 3 , 2 7 , 4 8 , 5 10 , 6 11 , 9 12
1 3 , 2 7 , 4 9 , 5 10 , 6 12 , 8 11
1 3 , 2 7 , 4 9 , 5 11 , 6 8 , 10 12
1 3 , 2 7 , 4 10 , 5 11 , 6 8 , 9 12
1 3 , 2 7 , 4 10 , 5 12 , 6 8 , 9 11
1 3 , 2 8 , 4 6 , 5 11 , 7 9 , 10 12
1 3 , 2 8 , 4 7 , 5 10 , 6 11 , 9 12
1 3 , 2 8 , 4 10 , 5 11 , 6 12 , 7 9
1 3 , 2 9 , 4 7 , 5 10 , 6 11 , 8 12
1 3 , 2 9 , 4 10 , 5 8 , 6 11 , 7 12
1 3 , 2 10 , 4 7 , 5 9 , 6 11 , 8 12
1 3 , 2 10 , 4 7 , 5 11 , 6 9 , 8 12
1 3 , 2 10 , 4 8 , 5 11 , 6 9 , 7 12
1 3 , 2 11 , 4 7 , 5 9 , 6 10 , 8 12
1 3 , 2 11 , 4 8 , 5 9 , 6 12 , 7 10
1 4 , 2 5 , 3 6 , 7 10 , 8 11 , 9 12
1 4 , 2 5 , 3 8 , 6 10 , 7 11 , 9 12
1 4 , 2 5 , 3 10 , 6 9 , 7 11 , 8 12
1 4 , 2 6 , 3 9 , 5 10 , 7 11 , 8 12
1 4 , 2 6 , 3 10 , 5 8 , 7 11 , 9 12
1 4 , 2 6 , 3 11 , 5 8 , 7 10 , 9 12
1 4 , 2 6 , 3 11 , 5 9 , 7 10 , 8 12
1 4 , 2 7 , 3 8 , 5 10 , 6 11 , 9 12
1 4 , 2 7 , 3 10 , 5 8 , 6 11 , 9 12
1 4 , 2 7 , 3 10 , 5 9 , 6 11 , 8 12
1 4 , 2 8 , 3 9 , 5 10 , 6 11 , 7 12
1 4 , 2 8 , 3 9 , 5 11 , 6 12 , 7 10
1 4 , 2 8 , 3 11 , 5 9 , 6 10 , 7 12
1 4 , 2 9 , 3 6 , 5 10 , 7 11 , 8 12
1 4 , 2 10 , 3 7 , 5 9 , 6 11 , 8 12
1 4 , 2 11 , 3 6 , 5 8 , 7 10 , 9 12
1 5 , 2 7 , 3 9 , 4 11 , 6 10 , 8 12

7 Edges:

1 3 , 2 4 , 5 14 , 6 9 , 7 11 , 8 12 , 10 13
1 3 , 2 5 , 4 6 , 7 10 , 8 12 , 9 13 , 11 14
1 3 , 2 5 , 4 6 , 7 14 , 8 10 , 9 12 , 11 13
1 3 , 2 5 , 4 6 , 7 14 , 8 11 , 9 12 , 10 13
1 3 , 2 5 , 4 10 , 6 9 , 7 12 , 8 13 , 11 14
1 3 , 2 6 , 4 9 , 5 12 , 7 11 , 8 13 , 10 14
1 3 , 2 6 , 4 10 , 5 7 , 8 12 , 9 13 , 11 14
1 3 , 2 6 , 4 11 , 5 7 , 8 10 , 9 13 , 12 14
1 3 , 2 6 , 4 12 , 5 7 , 8 11 , 9 13 , 10 14
1 3 , 2 7 , 4 9 , 5 11 , 6 13 , 8 12 , 10 14
1 3 , 2 7 , 4 10 , 5 12 , 6 8 , 9 13 , 11 14
1 3 , 2 7 , 4 10 , 5 13 , 6 8 , 9 12 , 11 14
1 3 , 2 7 , 4 11 , 5 13 , 6 8 , 9 12 , 10 14
1 3 , 2 8 , 4 10 , 5 12 , 6 13 , 7 9 , 11 14
1 3 , 2 8 , 4 11 , 5 12 , 6 14 , 7 9 , 10 13
1 3 , 2 9 , 4 6 , 5 12 , 7 14 , 8 10 , 11 13
1 3 , 2 9 , 4 7 , 5 11 , 6 12 , 8 14 , 10 13
1 3 , 2 9 , 4 7 , 5 12 , 6 13 , 8 10 , 11 14
1 3 , 2 9 , 4 10 , 5 8 , 6 12 , 7 13 , 11 14
1 3 , 2 11 , 4 8 , 5 12 , 6 10 , 7 13 , 9 14
1 3 , 2 12 , 4 8 , 5 10 , 6 13 , 7 11 , 9 14
1 3 , 2 13 , 4 7 , 5 10 , 6 11 , 8 14 , 9 12
1 4 , 2 5 , 3 6 , 7 10 , 8 12 , 9 13 , 11 14
1 4 , 2 5 , 3 6 , 7 14 , 8 11 , 9 12 , 10 13
1 4 , 2 5 , 3 10 , 6 9 , 7 12 , 8 13 , 11 14
1 4 , 2 6 , 3 7 , 5 10 , 8 12 , 9 13 , 11 14
1 4 , 2 6 , 3 7 , 5 12 , 8 11 , 9 13 , 10 14
1 4 , 2 6 , 3 9 , 5 10 , 7 12 , 8 13 , 11 14
1 4 , 2 6 , 3 11 , 5 12 , 7 10 , 8 13 , 9 14
1 4 , 2 6 , 3 12 , 5 9 , 7 11 , 8 13 , 10 14
1 4 , 2 7 , 3 8 , 5 10 , 6 12 , 9 13 , 11 14
1 4 , 2 7 , 3 8 , 5 10 , 6 13 , 9 12 , 11 14
1 4 , 2 7 , 3 11 , 5 9 , 6 12 , 8 13 , 10 14
1 4 , 2 7 , 3 12 , 5 9 , 6 13 , 8 11 , 10 14
1 4 , 2 7 , 3 12 , 5 10 , 6 13 , 8 11 , 9 14
1 4 , 2 7 , 3 13 , 5 9 , 6 11 , 8 12 , 10 14
1 4 , 2 8 , 3 9 , 5 10 , 6 12 , 7 13 , 11 14

1 4 , 2 8 , 3 9 , 5 11 , 6 12 , 7 14 , 10 13
1 4 , 2 8 , 3 9 , 5 12 , 6 13 , 7 10 , 11 14
1 4 , 2 8 , 3 12 , 5 9 , 6 11 , 7 13 , 10 14
1 4 , 2 8 , 3 13 , 5 10 , 6 11 , 7 14 , 9 12
1 4 , 2 9 , 3 6 , 5 10 , 7 12 , 8 13 , 11 14
1 4 , 2 9 , 3 10 , 5 12 , 6 13 , 7 14 , 8 11
1 4 , 2 11 , 3 7 , 5 9 , 6 12 , 8 13 , 10 14
1 4 , 2 12 , 3 6 , 5 9 , 7 11 , 8 13 , 10 14
1 4 , 2 12 , 3 7 , 5 9 , 6 13 , 8 11 , 10 14
1 4 , 2 12 , 3 8 , 5 9 , 6 11 , 7 13 , 10 14
1 4 , 2 12 , 3 13 , 5 9 , 6 10 , 7 14 , 8 11
1 5 , 2 6 , 3 9 , 4 12 , 7 11 , 8 13 , 10 14
1 5 , 2 7 , 3 10 , 4 13 , 6 11 , 8 12 , 9 14
1 5 , 2 8 , 3 12 , 4 9 , 6 11 , 7 13 , 10 14
1 5 , 2 9 , 3 13 , 4 10 , 6 11 , 7 12 , 8 14

8 Edges:

1 3 , 2 5 , 4 6 , 7 16 , 8 11 , 9 13 , 10 14 , 12 15
1 3 , 2 6 , 4 12 , 5 7 , 8 11 , 9 14 , 10 15 , 13 16
1 3 , 2 7 , 4 11 , 5 14 , 6 8 , 9 13 , 10 15 , 12 16
1 3 , 2 8 , 4 11 , 5 13 , 6 15 , 7 9 , 10 14 , 12 16
1 3 , 2 10 , 4 7 , 5 13 , 6 14 , 8 16 , 9 11 , 12 15
1 3 , 2 10 , 4 12 , 5 8 , 6 14 , 7 15 , 9 11 , 13 16
1 4 , 2 5 , 3 6 , 7 16 , 8 11 , 9 13 , 10 14 , 12 15
1 4 , 2 6 , 3 7 , 5 8 , 9 12 , 10 14 , 11 15 , 13 16
1 4 , 2 6 , 3 7 , 5 12 , 8 11 , 9 14 , 10 15 , 13 16
1 4 , 2 6 , 3 11 , 5 12 , 7 10 , 8 14 , 9 15 , 13 16
1 4 , 2 7 , 3 8 , 5 11 , 6 14 , 9 13 , 10 15 , 12 16
1 4 , 2 7 , 3 8 , 5 12 , 6 9 , 10 14 , 11 15 , 13 16
1 4 , 2 7 , 3 13 , 5 10 , 6 14 , 8 12 , 9 15 , 11 16
1 4 , 2 8 , 3 9 , 5 11 , 6 13 , 7 15 , 10 14 , 12 16
1 4 , 2 8 , 3 9 , 5 12 , 6 14 , 7 10 , 11 15 , 13 16
1 4 , 2 8 , 3 9 , 5 12 , 6 15 , 7 10 , 11 14 , 13 16
1 4 , 2 8 , 3 14 , 5 10 , 6 12 , 7 15 , 9 13 , 11 16
1 4 , 2 9 , 3 10 , 5 12 , 6 14 , 7 15 , 8 11 , 13 16
1 4 , 2 9 , 3 10 , 5 13 , 6 14 , 7 16 , 8 11 , 12 15
1 4 , 2 10 , 3 11 , 5 8 , 6 14 , 7 15 , 9 12 , 13 16
1 4 , 2 12 , 3 13 , 5 9 , 6 14 , 7 11 , 8 15 , 10 16

1 4 , 2 13 , 3 7 , 5 10 , 6 14 , 8 12 , 9 15 , 11 16
1 4 , 2 13 , 3 14 , 5 9 , 6 11 , 7 15 , 8 12 , 10 16
1 4 , 2 14 , 3 8 , 5 10 , 6 12 , 7 15 , 9 13 , 11 16
1 5 , 2 7 , 3 10 , 4 14 , 6 11 , 8 13 , 9 15 , 12 16
1 5 , 2 7 , 3 14 , 4 8 , 6 11 , 9 13 , 10 15 , 12 16
1 5 , 2 8 , 3 12 , 4 15 , 6 10 , 7 13 , 9 14 , 11 16
1 5 , 2 8 , 3 15 , 4 9 , 6 11 , 7 13 , 10 14 , 12 16
1 5 , 2 9 , 3 14 , 4 10 , 6 11 , 7 13 , 8 15 , 12 16

9 Edges:

1 4 , 2 6 , 3 7 , 5 8 , 9 18 , 10 13 , 11 15 , 12 16 , 14 17
1 4 , 2 7 , 3 8 , 5 14 , 6 9 , 10 13 , 11 16 , 12 17 , 15 18
1 4 , 2 8 , 3 9 , 5 13 , 6 16 , 7 10 , 11 15 , 12 17 , 14 18
1 4 , 2 9 , 3 10 , 5 13 , 6 15 , 7 17 , 8 11 , 12 16 , 14 18
1 4 , 2 11 , 3 12 , 5 8 , 6 15 , 7 16 , 9 18 , 10 13 , 14 17
1 5 , 2 8 , 3 15 , 4 9 , 6 12 , 7 16 , 10 14 , 11 17 , 13 18
1 5 , 2 9 , 3 16 , 4 10 , 6 12 , 7 14 , 8 17 , 11 15 , 13 18
1 5 , 2 15 , 3 9 , 4 16 , 6 11 , 7 13 , 8 17 , 10 14 , 12 18
1 6 , 2 9 , 3 14 , 4 17 , 5 10 , 7 12 , 8 15 , 11 16 , 13 18