Generation of Graphs Embedded on the Torus

by

Matthew Adam Skala
B.Sc., University of Victoria, 1999

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. Wendy Myrvold, Supervisor (Dept. of Computer Science)

Dr. Ulrike Stege, Departmental Member (Dept. of Computer Science)

Dr. Gary MacGillivray, Outside Member (Dept. of Mathematics and Statistics)

Dr. Richard Anstee, External Examiner (Dept. of Mathematics, University of British Columbia)

Supervisor: Dr. Wendy Myrvold

ABSTRACT

An algorithm is presented and proved to generate one representative from every isomorphism class of embeddings on the torus of graphs with certain properties. Implementation issues, possible applications, and experimental results from an implementation of the algorithm are described, including the determination of all biconnected topological obstructions to torus embeddability containing ten or fewer vertices.

Examiners:

Dr. Wendy Myrvold, Supervisor (Dept. of Computer Science)

Dr. Ulrike Stege, Departmental Member (Dept. of Computer Science)

Dr. Gary MacGillivray, Outside Member (Dept. of Mathematics and Statistics)

Dr. Richard Anstee, External Examiner (Dept. of Mathematics, University of British Columbia)

Contents

List of Tables

List of Figures

Acknowledgments

The author's work was supported by an NSERC Postgraduate Scholarship (PGS A) from May 2000 onwards, and by a University of Victoria Fellowship prior to that. Thanks also to the author's academic supervisor, Wendy Myrvold, for all her help and support; to Staszek Radziszowski for computer processing resources; and to Meredith Tanner for some words of wisdom.

Chapter 1

Introduction

Graphs describe patterns of connections between things, in an abstract and powerful way. We can deal with graphs mathematically as purely abstract entities, without invoking any concept of space. But as soon as we try to visualise a graph, we have to place it in a physical space, and immediately we encounter topological questions. One of the simplest topological questions we can ask about a graph is whether or not we can draw the graph on a given surface without any of the edges crossing. That is the central question considered in this work.

Embedding problems appear in many real-world situations. For instance, if we use a graph to represent a network of rail lines between cities, we may wish to know whether we can lay out the tracks to maintain the pattern of connections without needing any bridges. A similar situation occurs on a smaller physical scale in the design of electronic circuits. There, each chip may contain many components, and each board may contain many chips, and in both cases there is a pattern of connections between them which must be maintained. In these kinds of situations we may be allowed to use some limited number of crossings between connections, but such crossings are expensive and may not always be available.

Graphs embedded on surfaces are of interest in more purely theoretical situations also. Some things we would like to do with graphs are easier to do when the graphs

are embedded. For instance, the graph isomorphism problem, which is not known to be polynomial-time in the general case [36], can be solved in linear time for graphs embedded on the plane [18, 23]. Since graphs that embed on specific surfaces appear especially desirable both for physical applications and in more abstract situations, it becomes natural to ask how we can find such graphs. Perhaps we could even hope for exhaustive lists of them.

The plane is naturally the first surface on which we might want to embed graphs, and many results are known on planar graphs. The graph isomorphism problem is easier for planar graphs than for general graphs, as mentioned above. Several algorithms are known for testing whether a graph is planar [10, 11, 14, 15]. Some work has also been done on generation of planar graphs [12]. The projective plane is interesting as the simplest non-orientable surface. Graphs known to be projective planar can have their orientable genus computed in polynomial time [16]. Some algorithms are known for embedding graphs on the projective plane [29, 32, 35] and for generating limited classes of projective planar graphs [6, 7].

In this work we consider graphs embedded on the torus. More specifically, we generate all embeddings of diamond-free target graphs (defined in Section 2.3) on the torus. The torus appears to be the next logical step after examination of the plane and projective plane, and this work began with the question of generating randomly-chosen test cases for the "practical torus embedding" code of Neufeld and Myrvold [34]. We expanded the project to cover exhaustive generation of target embeddings. It then gave a method for obtaining torus obstructions (see Section 7.2) without needing a separate torus embedding algorithm.

Algorithms to embed graphs on the torus have been studied by Juvan, Marinček, and Mohar [24] as well as by Neufeld and Myrvold [34, 33]. There is an algorithm by Filotti [17] for embedding cubic graphs on the torus. Some general results for embedding on arbitrary surfaces, for instance the linear-time embedding algorithm of Mohar [30], could be applied to the torus. Unfortunately, that algorithm has not been implemented and appears difficult to implement practically. The generation

results of Barnette [7] for 4-connected graphs can also be applied to the torus as well as the projective plane. In our generation work, we have chosen a set of target graphs intended to make the resulting lists as useful as possible for study of the embedding problem, while still being easy to generate.

We have also chosen to generate embeddings of toroidal graphs rather than merely the graphs themselves. This choice appears to make the generation algorithm easier, but it also allows us to study how many embeddings exist for each graph. The equivalent question on the plane has been studied by Chiba, Nishizeki, Abe, and Ozawa [14], and Cai gives a simplified algorithm for counting planar embeddings [13]. Because we generate embeddings exhaustively, we can find how many torus embeddings any given graph has simply by counting them in the output.

The next chapter contains definitions of terms and notation used in this work. We then discuss in Chapter 3 the operations we perform on embeddings, and prove that our algorithm can generate all target embeddings. In Chapter 4 we discuss the effect of a subgraph called a "diamond" (defined in Section 2.1) and special diamond-related considerations for our algorithm. After that, we discuss some implementation issues in Chapter 5, and present experimental results in Chapter 6. We conclude in Chapter 7 with proposed applications and future work.

Chapter 2

Definitions and notation

Before discussing our results, we define some terms and notation used throughout the work. First we describe basic concepts of graphs and graph theory in Section 2.1. In Section 2.2 we introduce the concept of a surface and discuss graphs embedded on surfaces. In Sections 2.3 and 2.4 we describe the class of embeddings we generate, and the moves and starting points used to generate them. Then in Section 2.5, we describe the generation algorithm.

2.1 Graphs

A *graph* G consists of a finite set V of *vertices* and a finite set E of *edges* where each edge in E is associated with an unordered pair (u, v) of elements of V; the edge (u, v) is *incident to* or has as *endpoints* the vertices u and v. We disallow *multiple edges* (more than one edge with the same endpoints), and *loops* (edges of the form (u, u)).

The number of edges incident to a vertex is the *degree* of the vertex. The vertices u and v are *adjacent* if there is an edge (u, v) in the graph, and the vertices adjacent to a vertex u are called the *neighbours* of u.

Two graphs G_1 and G_2 are called *isomorphic* if there is a bijection ϕ from the

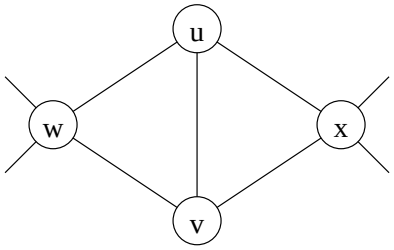


Figure 2.1: A diamond.

vertices of G_1 to the vertices of G_2 such that (u, v) is an edge in G_1 if and only if $(\phi(u), \phi(v))$ is an edge in G_2.

To *subdivide* an edge (u, v) in a graph means to introduce a new vertex w, add edges (u, w) and (w, v), and remove the edge (u, v). Two graphs are *homeomorphic* if there is a graph G such that they each can be obtained from G by relabelling vertices and subdividing edges.

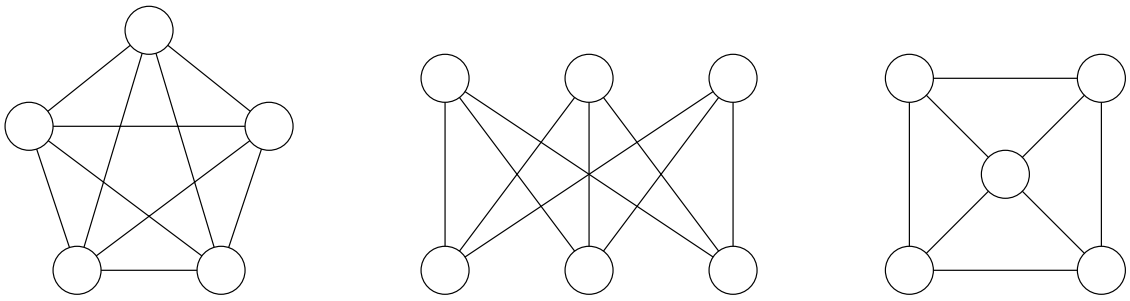
A graph G is called a *subgraph* of a graph H if the edge and vertex sets of G are subsets of the edge and vertex sets, respectively, of H. Let V be the vertex set of G, a subgraph of H. If G contains every edge in H whose endpoints are both in V, then G is called the *subgraph of H induced by V*.

If two adjacent degree three vertices u and v share the same other two neighbours, in other words the neighbours of u are $\{v, w, x\}$ and the neighbours of v are $\{u, w, x\}$, then the resulting subgraph, shown in Figure 2.1, is called a *diamond* and the edge between u and v is a *diamond edge*. There may or may not be an edge between w and x. A graph is *diamond-free* if it contains no diamond edges.

Consider a graph with vertices $\{v_1, v_2, \ldots, v_n\}$ and edges

$$\{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)\}$$

for some n greater than or equal to three. A graph isomorphic to this one is called a *cycle* of length n. Similarly, a graph with n vertices, all pairwise adjacent to each other, but no multiple edges or loops, is called the *complete graph* on n vertices and denoted by K_n. Observe that K_3 is a cycle of length three, which we will



(a) K_5 (b) $K_{3,3}$ (c) A five-vertex wheel

Figure 2.2: Some representative graphs.

call a *triangle*. We also refer to $K_{3,3}$, obtained by taking two disjoint sets X and Y of three vertices each, using their union as the vertex set, and adding an edge from each vertex in X to each vertex in Y. The graphs K_5 and $K_{3,3}$ are shown in Figure 2.2.

If we start with a cycle of at least three vertices and add one more vertex with edges from the new vertex to each vertex of the cycle, the resulting graph is called a *wheel*. A five-vertex wheel is shown in Figure 2.2. Observe that the smallest wheel is isomorphic to K_4.

A sequence of distinct vertices $\langle v_1, v_2, \ldots, v_k \rangle$, where each pair of consecutive vertices is adjacent, is called a *path* with endpoints v_1 and v_k. Two paths are called *internally vertex disjoint* if they have no vertices in common except possibly the endpoints. A graph G is *connected* if for every pair of vertices a and b in G, G contains a path from a to b. A graph G is *biconnected* if it is connected, and the graph obtained by deleting any one vertex is still connected. More generally, G is *k-connected* if G has greater than k vertices and we can remove any set of fewer than k vertices and the edges incident to them and always have the remaining graph be connected.

A *k-connected component* of a graph G is a subgraph H of G such that H is

k-connected but is not a proper subgraph of any other k-connected subgraph of G. A set of k vertices whose removal increases the number of connected components is called a *k-cut*, and the single vertex in a 1-cut is called a *cut vertex*. Note that our k-cuts can be described more precisely as k-vertex cuts; it is also possible to define a k-edge cut of edges which can be removed to disconnect a graph, but we do not use that concept in this work.

To *contract* an edge (u, v) means to remove the edge (u, v), then identify u and v. The reverse operation of contracting an edge is called *splitting* a vertex. We say that a graph G is a *minor* of a graph H if G can be obtained from H by the *minor operations* of removing edges, removing vertices of degree zero, and contracting edges.

2.2 Surfaces and embeddings

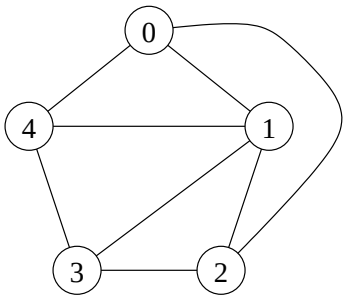
Although we will not discuss the topology of surfaces extensively, except as it applies directly to this graph-theoretic work, we will define a *surface* as a topological space in which any two distinct points have disjoint neighbourhoods, and every point has a neighbourhood topologically equivalent to a two-dimensional open disc. Intuitively, a surface is a space that looks like a plane, when examined within a small enough neighbourhood.

The classification of surfaces is well known, and described in detail in introductory textbooks on topology, such as that by Kinsey [25]. Surfaces are uniquely determined by the properties of *genus* and *orientability*. The genus may be any nonnegative integer, and if the surface has genus greater than zero, it may be *orientable* or *non-orientable*. The *plane*, equivalent to the sphere, is the only surface of genus zero, and is orientable. Genus may be thought of intuitively as describing the number of handles or bridges on the surface, and orientability as describing whether or not the surface has a well-defined sense of clockwise.

After the plane the remaining orientable surfaces are called the *torus*, with genus

one, and the *k-handled torus* for each k greater than one, with genus k. We call the non-orientable surface with genus one the *projective plane*, and with genus two the *Klein bottle*.

In this work we deal with *combinatorial embeddings*, which represent drawings of graphs on orientable surfaces. A combinatorial embedding consists of a list, for each vertex in the graph, of the neighbours of that vertex in clockwise order. An example of a combinatorial embedding is shown in Figure 2.3. The adjacency lists are cyclic, in that we can start at any neighbour; the lists $\langle u, v, w, x\rangle$ and $\langle v, w, x, u\rangle$ are equivalent. Reversing a list would violate the clockwise ordering and is not allowed. Two combinatorial embeddings are *isomorphic* if one can be obtained from the other by relabelling vertices, choosing a starting point for each adjacency list, and possibly reversing all adjacency lists at once (which can be imagined as mirror-reversing the embedding).



$0 : \langle 1, 4, 2\rangle$
$1 : \langle 0, 2, 3, 4\rangle$
$2 : \langle 1, 0, 3\rangle$
$3 : \langle 4, 1, 2\rangle$
$4 : \langle 0, 1, 3\rangle$

Figure 2.3: A drawing of a graph on the plane, and a corresponding combinatorial embedding.

Drawing a graph on a surface divides the surface into regions called *faces*, and a simple algorithm applied to a combinatorial embedding can count the faces and find the sequence of vertices around each face [3, Section 2.5]. If every face contains three vertices, then the embedding is called a *triangulation*. From the combinatorial embedding of a connected graph with n vertices, m edges, and f faces, we can calculate the *genus* of the embedding g with the formula $g = (m - n - f + 2)/2$ [22].

A combinatorial embedding describes a drawing of a graph on the orientable surface with the corresponding genus. The existence of combinatorial embeddings allows us to define *embeddability*: a graph G is said to be embeddable on an orientable surface S if there exists an embedding of G with genus no greater than the genus of S. The *orientable genus* of a graph is the least genus of an orientable surface on which the graph is embeddable. Graphs with orientable genus zero are called *planar* and with genus one *toroidal*.

2.3 Target embeddings

A *move* is an operation we can perform on a combinatorial embedding to produce another combinatorial embedding with more edges. The generation algorithm (described in Section 2.5) starts from a set of embeddings called seeds (defined below) and applies moves from a fixed set of types to generate *target embeddings*. We describe our choices for the target embeddings here, and move types and seeds in the next section.

Definition 2.3.1 *A* target graph *is a graph G such that:*

- *G has orientable genus one;*
- *G has no vertices of degree less than three; and*
- *G is biconnected.*

A *target embedding* is a combinatorial embedding of a target graph on the torus. The complete algorithm as we eventually implemented it uses an additional restriction, generating diamond-free target embeddings, because we consider diamond-free embeddings more topologically interesting. We define target embeddings as above to simplify the discussion in Chapter 3. We then build the more specific results applicable to the final form of our software, in Chapter 4.

The *seeds* for a given set M of moves are those target embeddings that cannot be generated from other target embeddings by moves in M. Since a move increases the number of edges in an embedding, it follows that any target embedding may be generated from some seed by a sequence of moves in M.

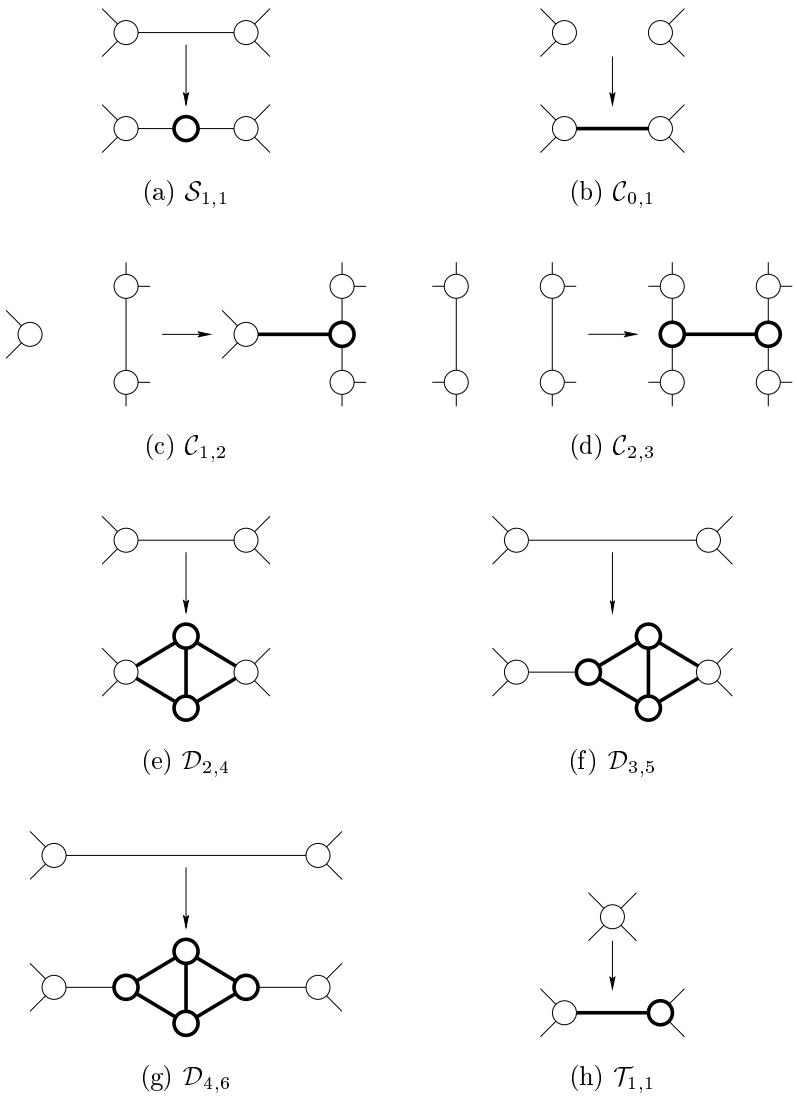
2.4 Moves and sets of moves

Here we define all the types of moves we consider in this work. All these moves maintain the genus of an embedding and affect at most a constant number of vertices and edges, and all our moves increase the number of edges in the embedding. With one exception, all the moves also preserve the other conditions on target embeddings: biconnectivity and no vertices of degree less than three. The expection is that that the $\mathcal{S}_{1,1}$ move introduces a degree two vertex. Although moves formally take place on embeddings, we often discuss the corresponding graph operations using the same symbols.

We denote types of moves with symbols like $\mathcal{M}_{n,m}$, where $\mathcal{M}$ is a mnemonic letter representing the general kind of move, n is the number of vertices added by this move, and m is the number of edges added by this move, which is always at least one. We also define *reverse moves*, as the inverses of the forward moves. Reverse moves do not always preserve the constraints preserved by the corresponding forward moves; for instance, removing an edge can reduce the connectivity of the graph in an embedding, whereas adding an edge can never reduce the connectivity.

An $\mathcal{S}_{1,1}$ move consists of subdividing an edge (u, v) into two edges by adding a new vertex. The edge (u, v) is removed, a new vertex w is introduced, and edges (u, w) and (w, v) are added. Note that the new vertex has degree two, so the resulting embedding is not a target embedding. The $\mathcal{S}_{1,1}$ move type is used in defining other moves that do preserve the target properties.

A $\mathcal{C}_{0,1}$ move consists of adding an edge in a face of the embedding, between two vertices not already adjacent to each other. To make a $\mathcal{C}_{1,2}$ move, we first subdivide



(a) $\mathcal{S}_{1,1}$ (b) $\mathcal{C}_{0,1}$

(c) $\mathcal{C}_{1,2}$ (d) $\mathcal{C}_{2,3}$

(e) $\mathcal{D}_{2,4}$ (f) $\mathcal{D}_{3,5}$

(g) $\mathcal{D}_{4,6}$ (h) $\mathcal{T}_{1,1}$

Figure 2.4: Some examples of moves

an edge as with an $\mathcal{S}_{1,1}$ move, then add a new edge from the new vertex to some other vertex on a face containing the divided edge. The move type $\mathcal{C}_{2,3}$ consists of using two $\mathcal{S}_{1,1}$ moves to subdivide two edges on the same face, then adding an edge through the face, between the two new vertices.

Move types $\mathcal{D}_{2,4}$, $\mathcal{D}_{3,5}$, and $\mathcal{D}_{4,6}$ each consist of removing an edge of the embedding and replacing it with a subgraph including a diamond, as shown in Figures 2.4(e), 2.4(f), and 2.4(g). If the edge being removed appears twice on the same face, or if it is incident to a vertex that appears more than once on the same face, then there may be two or more inequivalent ways to embed the newly-added subgraph while maintaining the rest of the embedding. This issue is discussed in detail in Chapter 4.

A $\mathcal{T}_{1,1}$ move consists of splitting a vertex into two adjacent vertices, in such a way that at least one of the new vertices has degree three. The restriction to creating a degree three vertex may seem mysterious, but the only occasions where we have a reason to make a vertex-splitting move are those where we are creating a degree three vertex anyway. Making the restriction explicit allows us to simplify the computer software based on these results.

We say that a set of move types M is *sufficient* with a given set of seeds if every target embedding can be generated from one of the seeds by a sequence of moves from M. Obviously, if M is sufficient then every superset of M is also sufficient, and if M is not sufficient, then no subset of S is sufficient. We call a set of moves *minimal* if it is sufficient but has no proper subset that is sufficient.

2.5 The generation algorithm

Although many of the details involve concepts that have yet to be discussed, we present the overall generation algorithm here, to motivate the details presented in subsequent chapters. We follow a general algorithm of orderly generation similar to that described by McKay [28]. Our goal is to generate one representative for each

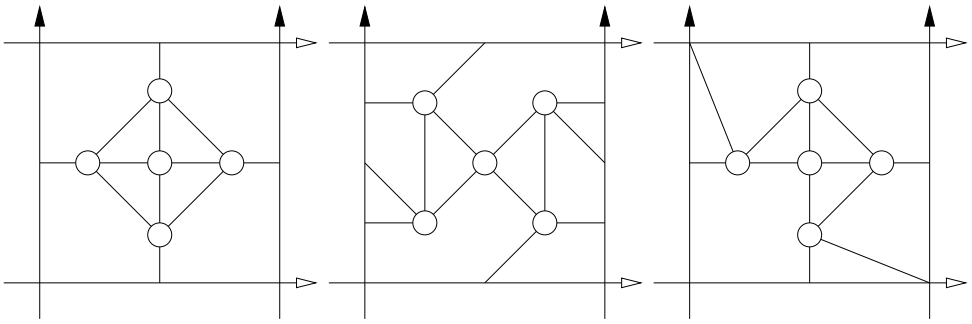
isomorphism class of diamond-free target embeddings, up to a chosen number of vertices and/or edges.

We use the move set $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ and the set of seeds consisting of all embeddings on the torus of K_5 or $K_{3,3}$. There are eight embeddings (up to isomorphism) in that set, enumerated by Argyle [5], and they are shown in Figure 2.5. In Chapter 3 we show that these moves and seeds are sufficient to generate all target embeddings, and in Chapter 4 we show that they continue to be sufficient when we introduce a limit of at most one diamond in each embedding.

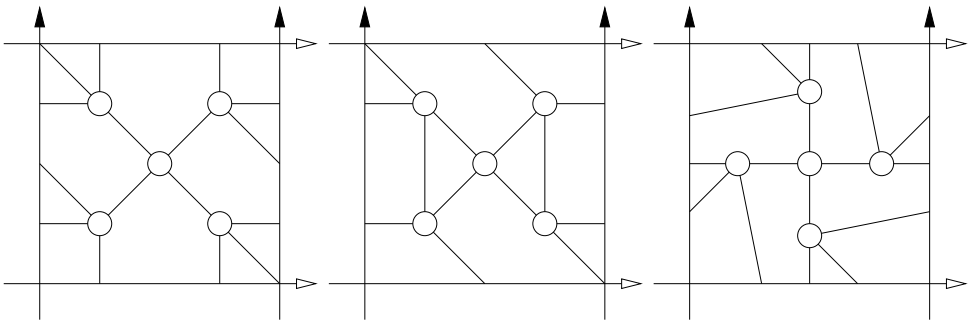
Any target embedding C either is one of the seeds, or has a *parent* P which is another target embedding with fewer edges than C, such that C can be obtained from P by a $\mathcal{C}_{0,1}$, $\mathcal{D}_{2,4}$, or $\mathcal{T}_{1,1}$ move. If P is the parent of C, then C is a *child* of P. The existence of parents is proved in Chapter 3, along with some discussion of other sets of moves and seeds we considered using.

In order to generate all diamond-free target embeddings, we sometimes need to examine target embeddings containing one diamond, as discussed in Chapter 4. It is not clear which target embeddings with one diamond are necessary to generate all diamond-free target embeddings. Rather than spending computation time in a complicated test for whether a diamond is really necessary, we examine all target embeddings with at most one diamond. We do not, however, need to consider embeddings containing more than one diamond. For any target embedding with at most one diamond, other than a seed, we can find a parent with fewer edges and at most one diamond (proved in Theorem 4.2.1); therefore we can eliminate all target embeddings with more than one diamond.

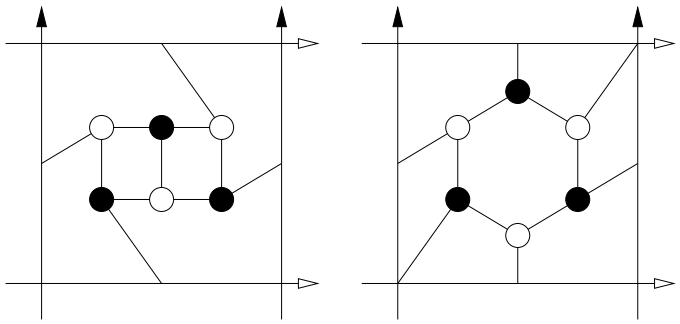
There are a few more details necessary to make sure we generate exactly one representative for each isomorphism class of diamond-free target embeddings. A *canonical form* for an embedding is an object representing the embedding such that two embeddings are isomorphic if and only if they have the same canonical form. The particular canonical form we use is discussed in Section 5.1. When we consider an embedding as a possible parent for a given child, we generate a copy



(a) K_5 (b) K_5 (c) K_5



(d) K_5 (e) K_5 (f) K_5



(g) $K_{3,3}$ (h) $K_{3,3}$

Figure 2.5: The eight seed embeddings for our algorithm.

of the child's parent by a reverse move, and use the canonical form to compare the possible parent with the parent. We also define *move labels* in Section 5.2, which are sequences of vertex labels expressing where in a parent embedding we can make a move to get to the child embedding. Sometimes two different moves can generate the same child from the same parent; move labels, along with the automorphism group calculated during the canonical labelling, allow us recognize that situation and generate the child exactly once.

The following pseudocode describes our algorithm for finding a parent, with all the applicable restrictions. Note that the parent-finding algorithm returns two things: the parent P itself, and a canonical move label for a move to make on P to give the child C. Since every C we will pass into this code has some parent meeting the conditions we test, the algorithm must return some parent and move label. We never attempt to find the parent for a seed. It is important that PARENT(C_1) and PARENT(C_2) be the same for any isomorphic C_1 and C_2; in our implementation, we achieve that by calling PARENT only with canonically labelled input embeddings.

```
PARENT(C) :
for each reverse move M we can apply to C, in some deterministic order
    apply M to C to obtain P
    if P is a target embedding containing at most one diamond
        canonically label P, finding its automorphisms as a side effect
        R ← move label for the inverse of M
        R ← least image of R under any automorphism of P
        return (P, R)
    end if
end for
```

Note that because every target embedding with at most one diamond has a parent which is a target embedding with at most one diamond (except seeds, which are never used as inputs to PARENT), PARENT must return some parent before the

loop terminates. The subroutine PARENT is written with care to make its return value a deterministic function of the canonically-labelled input. As a result, even if the same child could be reached from the same parent by two inequivalent moves, the move from parent to child will only match the move label R once, and so we will still generate the child exactly once.

We use PARENT in a recursive algorithm to do the actual generation. The following pseudocode describes an algorithm called GENERATE, which takes a target embedding P with at most one diamond as an argument, and writes out all diamond-free target embeddings descended from P, including P if it is itself diamond-free, up to some preset limit on the number of edges and vertices. To generate an exhaustive list, we simply call GENERATE once with each of the eight seeds.

```
GENERATE(P) :
if P has more vertices or edges than the preset limits
    return
end if
if P is diamond-free
    output P
end if
for each move label M describing a move from {C_{0,1}, D_{2,4}, T_{1,1}} that
    we can apply to P
    if M is the lexically least image of itself under any automorphism of P
        find C by applying the move to P described by M
        canonically label C, finding its automorphisms as a side effect
        if C contains at most one diamond and (P, M) = PARENT(C)
            GENERATE(C)
        end if
    end if
end for
```

Examination of this pseudocode will reveal a few potential inefficiencies; for instance, if P already contains one less edge than the limit, then there is no point even considering $\mathcal{D}_{2,4}$ moves which would create children too large to output. In our C language implementation of the algorithm we address many of these kinds of issues; the description here is intended to explain the algorithm as clearly as possible rather than provide an exhaustive guide to the features of the implementation.

Chapter 3

Choosing a set of moves

The question of which moves to use was central to the design of our generation algorithm, and in the course of our work we tried several different sets. In this chapter we first describe the set we finally settled on, consisting of the three move types $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$. In Section 3.1 we prove this set to be sufficient to generate all target embeddings, then in Section 3.2 we prove it minimal and discuss why it is an especially attractive minimal move set. Finally, in Section 3.3, we describe some of the other choices we considered, because the process of evolution from the original concept to this three-move set may be of interest.

In this chapter we discuss target graphs: biconnected graphs with genus one and all vertices of degree at least three. We discuss target graphs first, rather than beginning with the restricted class we eventually considered, so as to simplify the proofs. We use the term target graphs for these graphs, rather than for the more restricted graphs output by our software, so as not to have to make exceptions throughout this chapter. Making this definition does require us to make exceptions later, when we discuss diamond-free target graphs, but in those contexts we also need to discuss target graphs containing at most one diamond, target graphs containing exactly one diamond, and so on. In this chapter, it is useful to be able to ignore diamonds as far as possible.

3.1 A three-move sufficient set

Before beginning the proofs relating to existence of parents, we note that all the properties defining a target embedding are actually properties of the underlying target graph. The only way an embedding of a target graph could fail to be a target embedding would be if the genus of the embedding (not the graph) were not equal to one. None of the moves defined in Section 2.4 can increase the genus of an embedding, and although a reverse move in general can decrease the genus of an embedding, it cannot do so if the embedding was already genus one and the graph remains toroidal. An embedding of a toroidal graph cannot have genus zero. Therefore, we will generally talk about the existence of target graph parents for target graph children, rather than discussing embeddings. If a target graph C has a reverse move to a target graph P, then any torus embedding of C has a reverse move to some torus embedding of P, so the results apply equally well to target embeddings.

Fundamentally, what are the moves we need in our set? Since the graph minor hierarchy is central to the embedding problem, and the set of embeddable graphs on a surface is easily characterized in terms of forbidden minors [37], it seems natural that we should use moves resembling the reverse minor operations: splitting a vertex, or adding an edge. As long as we consider only connected graphs, there is no need to insert degree zero vertices.

Adding an edge seems simple enough, and corresponds to our $\mathcal{C}_{0,1}$ move. Splitting a vertex is a more complicated operation. It would be nice to restrict it in some way, to make computer implementation easier. It would also be nice to disallow splits that create degree two vertices, since degree two vertices clearly make no difference to the topological properties of the graph. Note that $\mathcal{S}_{1,1}$ can be imagined as splitting one of the neighbours to create a degree two vertex, instead of our usual description of it as subdividing an edge. In our three-move set, we restrict the split operation to always create a degree three vertex.

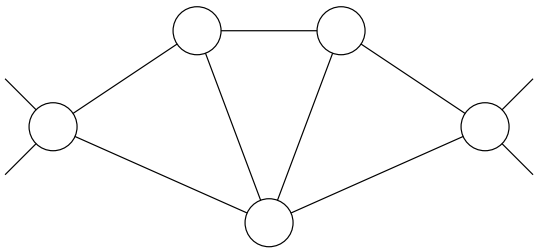


Figure 3.1: A super-diamond.

But with the split operation so restricted, we face the question of how to create diamonds, which are the subject of Chapter 4. We could disallow diamonds, as we disallowed degree two vertices, but then we would have no obvious way to create an infinite number of larger "banana-like" kinds of subgraphs, including the *super-diamond* of five vertices and seven edges, shown in Figure 3.1. We could perhaps disallow all such structures, by requiring that target embeddings be 3-connected, but then we might be faced with testing for 3-connectedness frequently in the software, as well as possible theoretical complications. Our decision was to require only biconnectedness from target embeddings, and have a special move, the $\mathcal{D}_{2,4}$ move, for creating diamonds.

The above intuitive description argues for why each of the moves in our three-move set may be necessary, but does little to justify the claim that they are sufficient to generate all target embeddings. Indeed, the sufficiency of this move set is far from obvious. The following obvious theorem is the beginning of our formal argument for sufficiency of the three-move set; we then complete the proof, and explore other features of this move set and its ability to generate target embeddings.

Theorem 3.1.1 *If a graph C can be obtained from a graph P by a $\mathcal{D}_{2,4}$ move, then P and C have the same orientable genus.*

Proof. If P is embeddable on a surface S, and (u, v) is the edge we can replace with a diamond to obtain C, then we can start with an embedding of P on S and introduce two new vertices w and x. We replace v with the subsequence $\langle w, x \rangle$ in

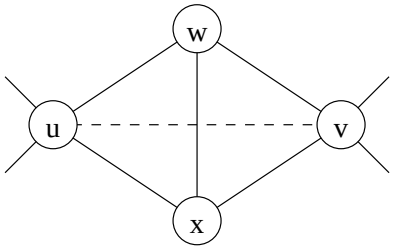


Figure 3.2: Adding a diamond without changing the genus.

the clockwise adjacency list of u, and replace u with the subsequence $\langle x, w\rangle$ in the clockwise adjacency list of v. We also give w the clockwise adjacency list $\langle u, v, x\rangle$ and x the list $\langle u, w, v\rangle$. The result of these operations is illustrated in Figure 3.2; the dashed line shows the position of the original edge (u, v). The resulting graph is C. We have added four edges, two faces, and two vertices; by the formula in the definition of genus for embeddings, we have not changed the genus of the embedding. So if P is embeddable on S, then so is C.

Although there may also be other ways to add a diamond to the embedding of P (see Section 4.3), it suffices that we can make the replacement in this one way and maintain the genus of the embedding. This proof concerns the genus of graphs, and the existence of any embedding of C on the torus shows that the genus of C is at most one.

Conversely, if C is embeddable on a surface S, we can start with an embedding of C on S and reverse the $\mathcal{D}_{2,4}$ move, removing four edges, two vertices, and up to two faces. The number of faces removed may be less than two because we may sometimes remove an edge that appeared twice on a face. Thus, the genus of the resulting embedding of P may be less than the genus of the embedding of C, but is certainly no greater; so P is also embeddable on P. Recall that "P embeddable on S" is true if there is an embedding of P with genus less than or equal to the genus of S.

Therefore for any surface S, P is embeddable on S if and only if C is embeddable on S; the graphs are embeddable on the same surfaces, and have the same orientable

genus. □

By the result known as Kuratowski's Theorem [26, 19, cited in [3]], any graph that is not planar must contain a subgraph homeomorphic to K_5 or $K_{3,3}$; we call that subgraph the *Kuratowski subgraph*. Thus, any target embedding must contain a subembedding of a graph homeomorphic to K_5 or $K_{3,3}$. We make use of that property in proving the existence of parents for target embeddings. Given a target embedding E, we can always find a subembedding of E which is an embedding of a graph homeomorphic to K_5 or $K_{3,3}$. We colour the chosen subembedding *red*, and define the *red-degree* of a vertex in E to be the number of red edges incident to that vertex. The vertices in the red subgraph with red-degree not equal to two are called *main vertices* [29]. Note that no vertex can have red-degree one, because then the red subgraph could not be homeomorphic to K_5 or $K_{3,3}$. We may change the colouring later, but will always preserve the property that the red subgraph is homeomorphic to K_5 or $K_{3,3}$, and therefore nonplanar. The following lemma is useful in manipulating the red colouring.

Lemma 3.1.2 *If a graph G with a red-coloured subgraph homeomorphic to K_5 or $K_{3,3}$ contains a triangle with vertices $\{u, v, w\}$ and edges $e = (u, v)$, $f = (v, w)$, and $g = (w, u)$, and no edges incident to u are red except possibly e and g, then the set of red edges in the triangle must be $\{e, g\}$, $\{f\}$, or the empty set, and we can freely exchange the two nonempty possibilities while keeping the red subgraph homeomorphic to K_5 or $K_{3,3}$.*

Proof. Since no other edges incident to u are red and the red-degree of u cannot be one, e and g must be both red or both not red. All three edges in the triangle cannot be red because then u would have red-degree two and by eliminating u the red subgraph would contain a multiple edge and not be homeomorphic to K_5 or $K_{3,3}$. That leaves only the listed possibilities for the set of red edges in the triangle.

If H_1 is the red subgraph when f is the only red edge in the triangle, and H_2 is the red subgraph when e and g are red but f is not, then the graph obtained by

starting from H_1 and subdividing f is isomorphic to H_2. Recall that no other edges incident to u can be red. Then H_1 is homeomorphic to H_2 and so if one of them is homeomorphic to K_5 or $K_{3,3}$, the other must also be. $\square$

The red colouring allows us to prove results about edges that can safely be removed or contracted without making the graph planar, because as long as we do not disturb the red subgraph too much, the graph must remain nonplanar. There will be times when we change the colouring, to make certain edges red or not, but in all cases, we preserve the property that the red subgraph is homeomorphic to K_5 or $K_{3,3}$.

Lemma 3.1.3 *Let C be a target graph containing a red-coloured subgraph homeomorphic to K_5 or $K_{3,3}$, as described above. If we contract an edge e in C with a degree three endpoint u by a reverse $\mathcal{T}_{1,1}$ move, where e may or may not be red but the other two edges incident to u are not both red, and assuming that the reverse $\mathcal{T}_{1,1}$ move does not create a multiple edge, then the resulting graph P is not planar.*

Proof. Since C contains a red subgraph homeomorphic to K_5 or $K_{3,3}$, the graph P can only be a planar graph if the edge contraction makes the red subgraph planar. We can also change the colouring as described in Lemma 3.1.2 without changing the homeomorphism of the red subgraph to K_5 or $K_{3,3}$. Contracting an edge can make the red subgraph not homeomorphic to K_5 or $K_{3,3}$ in only two ways: by identifying two main vertices, or by identifying a vertex u that is not a main vertex with a neighbour v that is red, but where the edge (u, v) is not red.

Because the two edges other than e incident to u are not both red and u has degree three, the red-degree of u can be at most two. It cannot be one because no vertex has red-degree one, so it must be zero or two. Then u is not a main vertex of the red subgraph. If the red-degree of u is zero then contracting the edge cannot make the red subgraph planar because all the red vertices and edges are unchanged by the operation. If the red-degree of u is two, that also means that e is red, and

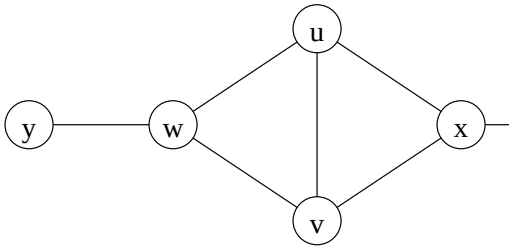


Figure 3.3: Vertex names for the proof of Theorem 3.1.4.

then we are contracting a red edge with an endpoint that is not a main vertex, and so the red subgraph is still homeomorphic to K_5 or $K_{3,3}$. $\square$

With Lemma 3.1.3 providing a sufficient condition under which we can make a reverse $\mathcal{T}_{1,1}$ move and preserve nonplanarity, we are ready to begin finding possible parents for target graphs. The following theorem shows that if we can generate all diamond-free target embeddings, then we can generate all target embeddings with one or more diamonds. This theorem does not place any restrictions on the number of diamonds in the parent.

Theorem 3.1.4 *Any target graph C that contains a diamond can be obtained from a target graph P with fewer edges by some move in the set $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$.*

Proof. Let (u, v) be the diamond edge, and w and x be the two neighbours shared by u and v, as shown in Figure 3.3. If there is an edge (w, x) then we can remove that edge with a reverse $\mathcal{C}_{0,1}$ move. The graph P has fewer edges than C. The graphs P and C must have the same genus because by Theorem 3.1.1 we can replace the diamond with an edge in each one, without affecting the genus, to obtain two graphs that are the same except for the presence of a multiple edge and so have the same genus.

The graph P is biconnected because there are still two internally vertex disjoint paths between w and x, namely $\langle w, u, x\rangle$ and $\langle w, v, x\rangle$. Finally, P has all vertices of degree at least three. Only w and x have their degrees reduced by the reverse move. If w had its degree reduced to two, then either x was a cut vertex of C, or

C was K_4; similarly, if x had its degree reduced to two, then either C was K_4 or w was a cut vertex of C.

If there is no edge (w, x) in C, we consider the degrees of w and x. If one of them (we say without loss of generality w) has degree three, we call its third neighbour (besides u and v) y. We will contract the edge between w and y with a reverse $\mathcal{T}_{1,1}$ move to get a target graph P with fewer edges than C. The contraction of (w, y) cannot create a multiple edge because then y and w would have to be part of a triangle, and the third vertex would have to be u or v. But we already know three distinct neighbours for each of those already; the only way $\{w, y, u\}$ or $\{w, y, v\}$ could be a triangle would be if x and y were the same vertex, in which case there would be an edge (w, x), and that possibility was considered above. Since both endpoints have minimum degree three, the edge contraction cannot reduce the degree of any vertex. Contracting (w, y) could only reduce the connectivity of the graph if $\{w, y\}$ were a two-vertex cut, and then each of them would also be a cut vertex, contradicting the biconnectedness of C. Contracting an edge cannot increase the genus of a graph. Thus, it only remains to show that contracting (w, y) does not reduce the genus of the graph.

Suppose we eliminate the diamond with a reverse $\mathcal{D}_{2,4}$ move to create a graph H, which contains at least one degree two vertex (namely w) and so is not a target graph, but is nonplanar by Theorem 3.1.1. We can colour a red subgraph of H homeomorphic to K_5 or $K_{3,3}$ as above, then replace the diamond to obtain a colouring of C. If the edge (w, x) was red in H then we colour red the edges (w, u) and (u, x); other than that, all the edges introduced when we replace the diamond remain uncoloured. Then (w, u) and (w, v) are not both red, so by Lemma 3.1.3 the graph P is nonplanar and therefore P is a target graph.

The only remaining case for the theorem occurs if w and x both have degree greater than three, with no edge (w, x). In that case, we can perform a reverse $\mathcal{D}_{2,4}$ move to eliminate the diamond, giving a graph P. A reverse $\mathcal{D}_{2,4}$ move must leave the genus of the graph unchanged, by Theorem 3.1.1. The reverse move does not

create a multiple edge because there is no edge (w, x) in C. Because w and x have degree greater than three in C and each has its degree reduced by one, their degrees in P are still are least three. The graph P must be biconnected because if there were a pair of vertices y and z which had two paths between them and distinct at the endpoints in C but not in P, then both those paths must have passed through w and x. Then x or y would have to be a cut vertex in C, unless they each had no other neighbours besides u and v; either choice contradicts the definition of C as a target graph. Therefore P has genus one, is biconnected, and has all vertices of degree at least three, and so is a target graph. □

When looking for a parent of a diamond-free target embedding, our reverse move must necessarily be a reverse $\mathcal{C}_{0,1}$ or $\mathcal{T}_{1,1}$ move, because the reverse $\mathcal{D}_{2,4}$ move can only be applied to a target embedding containing a diamond, and sometimes not even then. It is easy to find edges that can be removed while preserving nonplanarity; any non-red edge will do. Finding edges we can contract while preserving nonplanarity is also easy.

We can only apply a reverse $\mathcal{C}_{0,1}$ move when both endpoints of the edge we removed have degree greater than three, or else the resulting embedding would have vertices of degree two and would not be a target embedding. We can only apply a reverse $\mathcal{T}_{1,1}$ move to contract an edge with at least one endpoint of degree three, and only when the edge is not part of a triangle, to avoid creating multiple edges. With either reverse move, we must preserve the biconnectedness of the graph. The following lemma gives a sufficient condition for changes in a graph to preserve biconnectedness.

Lemma 3.1.5 *Let G_1 be a biconnected graph. Let H_1 be a biconnected subgraph of G_1 joined to the rest of G_1 by exactly two distinct vertices u and v; that is, u and v are the vertices of H_1 adjacent to vertices of G_1 not in H_1. Let H_2 be a biconnected subgraph of H_1 containing u and v, and let G_2 be the graph formed from G_1 by replacing H_1 with H_2. Then G_2 is biconnected.*

Proof. Let w and x be any two distinct vertices in G_2. If both w and x are in H_2, then there must be a cycle including these two vertices in H_2 and so in G_2, by the biconnectedness of H_2. If each of w and x either is not in H_2, or is one of u and v, then we can find a cycle including both of them in G_1. If that cycle includes any edges of H_1, then its intersection with H_1 must consist of a path from u to v. Then we find a path from u to v in H_2, and replace the path from u to v in H_1 with the path from u to v in H_2, to give us a cycle including w and x in G_2.

The remaining case is where one of w and x is not in H_2, and the other is in H_2 and is not u or v. Say without loss of generality that w is in H_2 and is not u or v. Then we find a cycle C_1 including w and x in G_1. The intersection of C_1 with H_1 must consist of a path between u and v. We find a cycle C_2 in H_2 that includes both u and w. If v is in C_2 then we can split C_2 into two internally vertex disjoint paths from u to v, choose one that includes w, and use that to replace the part of C_1 that passed through H_1, giving a cycle in G_2 that includes both w and x.

If v is not in C_2, we find two internally vertex disjoint paths from w to v through H_2. Let P_1 be one of those that does not contain u. Let y be the last vertex in P_1 that is in C_2; since w is in the path and in C_2, y must exist. We split C_2 into two internally vertex disjoint paths from u to y, choose one that includes w, and take the union of that with the segment of P_1 from y to v, to find a path P_2 from u to v contained in H_2 and containing w. We replace the intersection of C_1 and H_1 with P_2, to give a cycle in G_2 containing w and x. See Figure 3.4.

Therefore, for all distinct w and x in G_2 we can find a cycle in G_2 containing w and x; we can split that cycle into two internally vertex disjoint paths from w to x, and so G_2 is biconnected. □

The next lemma shows that under some conditions which target graphs happen to satisfy, we can find an edge whose removal leaves the graph biconnected.

Lemma 3.1.6 *Any biconnected graph G containing at least three vertices and at most two vertices of degree two, must contain some edge whose removal leaves the*

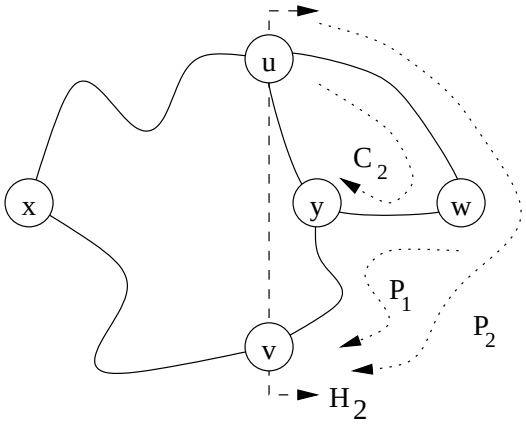


Figure 3.4: Finding a path from u to v containing w, in H_2.

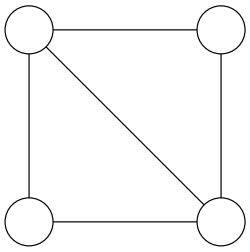


Figure 3.5: K_4 minus an edge.

graph biconnected.

Proof. By examination of all smaller graphs (there are only a few), K_4 minus one edge, shown in Figure 3.5, is the unique biconnected graph with at least three vertices, at most two vertices of degree two, and the smallest possible number of edges. We can remove the edge between that graph's two vertices of degree three, and leave a biconnected graph.

If G has more than five edges, we find a cycle F in G containing all the degree two vertices, if any. This must be possible because there are at most two degree two vertices, G is biconnected, and in a biconnected graph we can find a cycle containing any pair of vertices. Since G has at least three vertices, they cannot all be degree two, and any vertex can only have zero or two incident edges in F, so there must be an edge (u, v) in G that is not in F, and its endpoints u and v must

each have degree at least three in G. If removal of (u, v) leaves the remaining graph $G - (u, v)$ biconnected, then we are done.

Otherwise, there must be at least one vertex w of G which is a cut vertex in $G - (u, v)$. Removal of any cut vertex w must split $G - (u, v)$ into exactly two connected components, because otherwise w would be a cut vertex in G also. Then u and v must be in two different biconnected components of $G - (u, v)$, and all the edges of F must be in one biconnected component of $G - (u, v)$ because F is itself a biconnected subgraph; recall that (u, v) was chosen not to be in F. Then one of u and v, say without loss of generality u, must be in a biconnected component of $G - (u, v)$ which contains no edges in F and is attached to the rest of $G - (u, v)$ only at one cut vertex; we call that biconnected component H and that cut vertex x. Note that x need not be the same as w because w is any cut vertex of $G - (u, v)$ whereas x is the particular cut vertex joining H to the rest of $G - (u, v)$.

The subgraph H cannot consist only of u because then u would be a cut vertex of G, and H cannot consist only of u and one other vertex with an edge between them, because then u would have degree two in G. So H must contain at least three vertices. The subgraph H cannot include any vertex that had degree two in G, because H has no edges in F, and all edges incident to vertices with degree two in G were edges in F. The only vertices in H which have smaller degree in H than they had in G, are u and x; H includes all edges from G incident to any of the other vertices in H. So the vertices u and x are the only ones that could have degree two in H.

Then H is a biconnected graph with at least three vertices, at most two vertices of degree two, and fewer edges than G because it does not contain the edge (u, v). We can look recursively for an edge e to remove from H that will leave $H - e$ biconnected. Then by Lemma 3.1.5, removing e from G leaves $G - e$ biconnected. □

Preserving nonplanarity as well as biconnectedness is only a little more difficult.

In the following proof, we use a similar technique to split the graph into two pieces, but instead of finding a cycle and using it to remove part of the graph, we remove the red subgraph from consideration at the first stage of the recursion. After that, we know that any remaining edges can be removed while keeping the graph nonplanar, and so we simply apply Lemma 3.1.6. Note that Lemma 3.1.7 does not necessarily provide a useful reverse move from C, because the edge selected could have a degree three endpoint, resulting in a degree two vertex in P. However, in that case the edge still provides a useful starting point for the search for reverse moves in the sufficiency theorem.

Lemma 3.1.7 *If C is a target graph with a red-coloured subgraph homeomorphic to K_5 or $K_{3,3}$, then either C is K_5 or $K_{3,3}$, or C contains an edge e that is not red and can be removed to give a graph P which is a target graph except for possibly containing degree two vertices.*

Proof. If C is a target graph other than K_5 or $K_{3,3}$, then it must contain an edge e that is not red. Otherwise, it would contain at least one degree two vertex. If we remove e from C, the genus of the resulting graph $C - e$ must be the same as the genus of C, because removing an edge cannot increase the genus and the red subgraph is preserved. Removing e cannot create a multiple edge in $C - e$. If $C - e$ is biconnected, then it satisfies all conditions for P; otherwise, we will find a different edge to remove.

Suppose removing the edge e would render $C-e$ not biconnected, by creating one or more cut vertices. Removal of any one cut vertex u splits $C-e$ into two connected components. Removal of u cannot split $C - e$ into more than two components, because then (as shown in Figure 3.6), u would be a cut vertex in C also. The graph $C - e$ then contains at least two biconnected components. Because the genus of a graph is the sum of the genera of its biconnected components [8], exactly one biconnected component of $C - e$ is nonplanar. Since the two endpoints of e are in different biconnected components of $C - e$, one of them must be in a planar

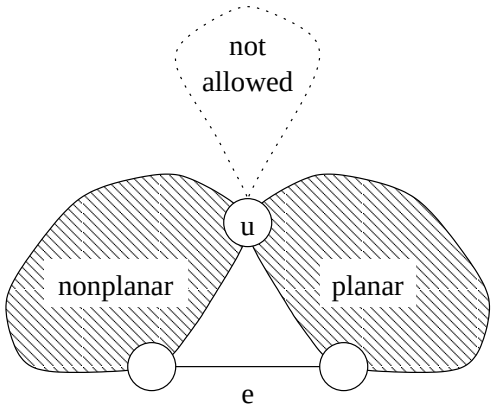


Figure 3.6: Removal of the cut vertex u must split $C-e$ into exactly two connected components: one planar, and one nonplanar.

biconnected component of $C - e$. Let H be a planar biconnected component of $C - e$ containing an endpoint of e, and v be the endpoint of e contained in H.

Just as in Lemma 3.1.6 above, the biconnected component H cannot consist only of v because then v would be a cut vertex of C, and H cannot consist of v and one other vertex with an edge between them, because then v would have degree two in C. When considered as a subgraph of C, H is connected to the rest of C only though the vertices v and w, where w is some cut vertex of $C-e$, as shown in Figure 3.7. The vertices v and w must each have degree at least two in H because otherwise H would not be biconnected. Any other vertices in H have the same degree in H that they had in C, necessarily at least three because C is a target graph.

So H is a biconnected graph with at least three vertices and at most two vertices of degree two. By Lemma 3.1.6 we can remove an edge of H and leave $H - e$ biconnected. Then by Lemma 3.1.5, P is biconnected. Since H contains no red edges, that edge must not be red, and its removal does not change the genus of C. Therefore P is a target graph except for possibly containing degree two vertices. □

If the edge e of Lemma 3.1.7 has a degree three endpoint then we cannot use a

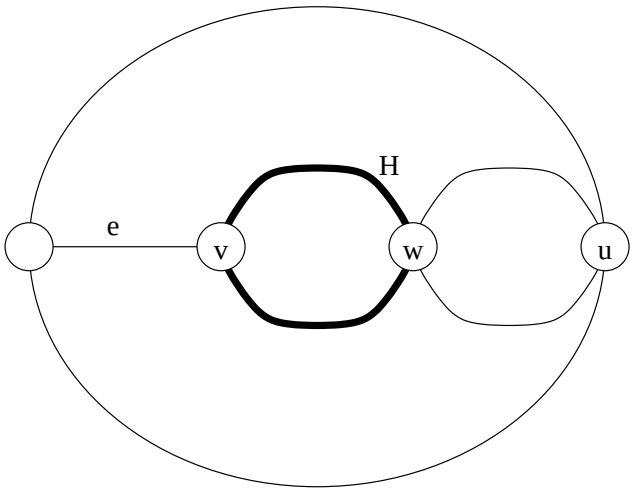


Figure 3.7: The biconnected component H.

reverse $\mathcal{C}_{0,1}$ move to remove it. It could be that the edge from Lemma 3.1.7 is on a triangle, or that any degree three endpoint of this edge also has both other incident edges red, so Lemma 3.1.3 might not allow us to contract it with a reverse $\mathcal{T}_{1,1}$ move either. The following theorem shows that even in such cases, we can always find some reverse move to use.

Theorem 3.1.8 *Any target graph C either is K_5, $K_{3,3}$, or can be obtained by making a move in $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ on a target graph P with fewer edges.*

Proof. If C contains a diamond, Theorem 3.1.4 provides a reverse move to P and we are finished. Otherwise, we find a subgraph of C homeomorphic to K_5 or $K_{3,3}$, and colour that subgraph red. By Lemma 3.1.7, there is an edge e which is not red, such that removal of e would leave the graph biconnected and toroidal.

If both endpoints of e have degree greater than three, then we can remove e with a reverse $\mathcal{C}_{0,1}$ move; the only remaining condition for P to be a target graph is for all vertices to have degree at least three, and removing an edge between two vertices of degree greater than three preserves that. If one endpoint of e has degree three, we call it u. The other endpoint, which we call v, may also have degree three. The other two neighbours of u we call w and x. Now, how many distinct triangles

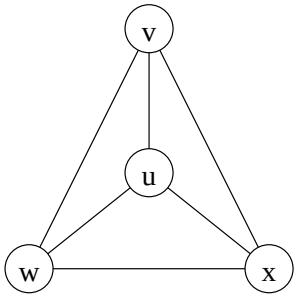


Figure 3.8: Vertex names when u is on three triangles.

in C can contain u? The answer is at most three because any triangle containing u is uniquely determined by u and two of its neighbours; there are only three ways to choose two of the three neighbours of u.

If u is not in any triangle in C: then we can contract one of the edges incident to u, other than e, with a reverse $\mathcal{T}_{1,1}$ move to find P. Since e is not red, by Lemma 3.1.3 the resulting graph P has genus greater than one. The graph P must be biconnected, every vertex in P must have degree at least three, and the genus of P must be at most one, because the reverse $\mathcal{T}_{1,1}$ move always preserves those properties. Since no edge incident to u was on a triangle, contracting one of them cannot create multiple edges. Therefore, P is a target graph.

If u is in three distinct triangles in C: then we have the situation shown in Figure 3.8, where the vertices u, v, w, and x induce a subgraph isomorphic to K_4. If any neighbour of u had degree three, then C would contain a diamond, and we already handled that case. So all of v, w, and x have degree at least four.

If any edge between two neighbours of u is not red, we say without loss of generality that it is (w, x) and eliminate it with a reverse $\mathcal{C}_{0,1}$ move to find P. Since the edge is not red and has both endpoints of degree greater than three, we know that we can remove it while preserving the genus and keeping the minimum degree at least three. The graph P is also biconnected, because there are still two internally vertex disjoint paths between w and x, namely $\langle w, u, x\rangle$ and $\langle w, v, x\rangle$. Therefore P is a target graph.

If all edges between neighbours of u are red, then because the edge (u, v) is not red, by Lemma 3.1.2 none of the edges incident to u is red and we can recolour to make (w, x) not red and (w, u) and (u, x) red. Then, as above, we remove the edge (w, x) with a reverse $\mathcal{C}_{0,1}$ move to obtain P, which is a target graph.

If u is in exactly one triangle in C: that triangle must include two edges incident to u. If any non-red edge incident to u is on the triangle, then we can contract along the one edge incident to u that is not on the triangle, with a reverse $\mathcal{T}_{1,1}$ move, to obtain P. By Lemma 3.1.3 this preserves the genus of the graph. Since the edge being contracted is not on a triangle, the reverse move creates no multiple edges. And a reverse $\mathcal{T}_{1,1}$ move can never decrease the degree of a vertex or reduce the connectivity of a graph, so the remaining conditions are satisfied, and P is a target graph.

If u is in one triangle and both edges incident to u in the triangle are red, then the triangle consists of u, w, and x. By Lemma 3.1.2, the edge (w, x) is not red, and we can re-colour so that (w, x) is red and none of the edges incident to u are red. Then we contract along (u, v) with a reverse $\mathcal{T}_{1,1}$ move, as above, and obtain a target graph P.

If u is in exactly two triangles in C: one neighbour of u must be in both of those triangles also; we call this neighbour y. If y is degree three, then (u, y) is a diamond edge, and we handled the case of graphs containing diamonds already. If (u, y) is red, then there is exactly one vertex z adjacent to both u and y such that (u, z) is red; then by Lemma 3.1.2, the edge (y, z) is not red, but we can recolour so that (y, z) is red and no edge incident to u is red.

Now we have a vertex u of degree three with a neighbour y of degree greater than three, and the edge (u, y) is not red and is on two triangular faces. Let u_0 be u and let u_1 be one of the other neighbours of u. Since (u_0, y) is part of two triangles, u_1 must also be a neighbour of y. If the edge (y, u_1) is red, we can recolour to make it not red while keeping the red subgraph nonplanar, by colouring (u_0, u_1) red if it was not already red, and colouring (u_0, y) red. This has the effect of splitting a

vertex in the red subgraph, which always preserves nonplanarity. The resulting red subgraph is nonplanar but may not be homeomorphic to K_5 or $K_{3,3}$; if necessary, we can uncolour additional edges to leave a red subgraph homeomorphic to K_5 or $K_{3,3}$, and (y, u_1) not red.

We then replace u_0 by u_1 and repeat, to choose vertices $u_2, u_3, \ldots, u_k$: for i starting at one, while u_i is degree three and (u_i, y) is on two triangles, we choose u_{i+1} to be the third neighbour of u_i, other than u_{i-1} and y. The vertex u_k that terminates the repetition must be adjacent to y and either u_k has degree greater than three or (u_k, y) is on just one triangle. Furthermore the edge (u_k, y) must not be red because we were recolouring to ensure that at every step of the way. There must be such a vertex u_k or else the vertices u_i would form a cycle with every vertex also adjacent to y; and then either y would be a cut vertex of C, or C would be a wheel and therefore planar. The following pseudocode restates the iteration algorithm:

FindUk(C, u, y) :
$u_0 \leftarrow u$
$u_1 \leftarrow$ a neighbour of u other than y
if (u_1, y) is red
    uncolour (u_1, y)
    colour (y, u_0) and (u_0, u_1) red
    uncolour additional edges as needed to make red subgraph homeomorphic
        to K_5 or $K_{3,3}$
end if
$i \leftarrow 1$
while *degree*$(u_i) = 3$ and (u_i, y) is on two triangles in C
    if (u_{i+1}, y) is red
        uncolour (u_{i+1}, y)
        colour (y, u_i) and (u_i, u_{i+1}) red

uncolour additional edges as needed to make red subgraph homeomorphic to K_5 or $K_{3,3}$

end if

$i \leftarrow i + 1$

end while

$k \leftarrow i$

return u_k

When we finish we have a vertex u_k adjacent to y, with the edge (u_k, y) not red, and either that edge is on just one triangle in C or the degree of u_k is greater than three. If the degree of u_k is greater than three, then we can remove (u_k, y) with a reverse $\mathcal{C}_{0,1}$ move to obtain P. Because the edge we remove is not red and has both endpoints of degree greater than three, P is nonplanar and has no vertices of degree less than three.

Choose a neighbour u_{k+1} of u_k, other than u_{k-1} and y. Since C is biconnected, we can find a path from y to u_{k+1} and not containing u_k. By adding the edges (y, u_k) and (u_k, u_{k+1}) to that path, we have a cycle. If the cycle does not contain the vertex u_{k-1}, we can replace the edge (y, u_k) with (u_k, u_{k-1}) and (u_{k-1}, y) to find a cycle through u_k and y that does not contain (u_k, y). Let z be the third vertex adjacent to u_{k-1}, besides u_k and y. Since the edge (u_{k-1}, y) is on two triangles, z must also be a neighbour of y. If the cycle passes through the vertex u_{k-1}, then because u_{k-1} has degree three, the cycle must include the edges (y, u_{k-1}) and (u_{k-1}, z). We can replace the sequence of consecutive vertices in the cycle $\langle u_k, y, u_{k-1}, z \rangle$ with $\langle u_k, u_{k-1}, y, z \rangle$ to find a cycle through u_k and y in P. Therefore, we can always find a cycle through u_k and y and not passing through (u_k, y), and that cycle provides two internally vertex disjoint paths between these vertices, so removal of (u_k, y) leaves a biconnected graph P. Since P fulfills the other conditions, it is a target graph.

If the degree of u_k is three, then the edge (u_k, y) is on only one triangle. Let

(u_k, z) be the edge incident to u_k that is not in that triangle. The edge (u_k, z) cannot be in any triangle because if it were in a triangle, that triangle would have to also contain either y or u_{k-1}. If the triangle contains y then the third vertex is a neighbour of y, (u_k, y) is in two triangles, and we would not have stopped at u_k. If $\{u_k, u_{k-1}, z\}$ is a triangle, then z must be a neighbour of u_{k-1} other than y, and so z must be a neighbour of y because (u_{k-1}, y) is in two triangles. Then $\{u_k, y, z\}$ must be a triangle, contradicting the claim that (u_k, y) was in only the triangle $\{u_k, u_{k-1}, y\}$.

Then (u_k, z) is an edge incident to the degree three vertex u_k and not on any triangle, so we can contract it with a reverse $\mathcal{T}_{1,1}$ move to leave a biconnected graph P with no multiple edges or vertices of degree less than three. Because the edge (u_k, y) is not red, by Lemma 3.1.3, P is nonplanar. Therefore P is a target graph. □

Theorem 3.1.8 shows the existence of a reverse move from any target graph other than K_5 or $K_{3,3}$ to a target graph with fewer edges. As noted at the start of this section, the existence of a target graph parent for any target graph child implies the existence of a target embedding parent for any target embedding child. So the sufficiency of the move set $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ to generate all target embeddings, with the torus embeddings of K_5 and $K_{3,3}$ as seeds, follows immediately.

3.2 The three-move set is minimal

Although we have intuitive justification for each of the three move types $\mathcal{C}_{0,1}$, $\mathcal{D}_{2,4}$, and $\mathcal{T}_{1,1}$, it may seem possible that some subset of these moves could still generate all target graphs. Our work began with two move types ($\mathcal{S}_{1,1}$ and $\mathcal{C}_{0,1}$) which were expanded to six move types ($\mathcal{C}_{0,1}$, $\mathcal{C}_{1,2}$, $\mathcal{C}_{2,3}$, $\mathcal{D}_{2,4}$, $\mathcal{D}_{3,5}$, and $\mathcal{D}_{4,6}$) to eliminate inefficiencies associated with degree two vertices. The six-move set was then cut to four moves by the adoption of the $\mathcal{T}_{1,1}$ move, which eliminated the need for $\mathcal{C}_{2,3}$,

$\mathcal{D}_{3,5}$, and $\mathcal{D}_{4,6}$.

But the $\mathcal{C}_{1,2}$ move was only eliminated much later, because at the time the four-move set was chosen we were attempting to work with embeddings on arbitrary surfaces, and there is an infinite set of target-like graphs on the plane, namely the wheels, which cannot be generated without $\mathcal{C}_{1,2}$. Actually, there are no target embeddings (under the current definition which requires nonplanarity) that cannot be generated by the three-move set above; but we only proved that late in the research, spurred by the experimental observation that removing $\mathcal{C}_{1,2}$ from our software did not reduce the list of graphs generated. Three-move sufficiency, proved in Theorem 3.1.8, is far from obvious. So there is some precedent for the idea that move sets may be reduced counter-intuitively.

In this section, not only do we prove that the three-move set is minimal in the strict sense that any proper subset requires an infinite set of seeds to generate all target graphs, but we also argue that the infinite set of seeds required by any further reduction of the three-move set would have to be inconveniently complicated. Thus, further reduction of the move set is not useful, even if we are willing to make a concession like generating most embeddings with our standard algorithm and using a different algorithm to generate the few not covered.

Lemma 3.2.1 *The set of moves $\{\mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ is not sufficient to generate all target embeddings with any finite set of seeds.*

Proof. These moves may add at most two edges for every vertex they add, so with a finite set of seeds the embeddings we can generate with n vertices have a maximum of $2n + k$ edges for some constant k. Triangulations on the torus with n vertices may have up to $3n$ edges, and for sufficiently large n, this will always exceed the number of edges in any n-vertex embedding we can generate. Therefore, there exist embeddings we cannot generate with this set of moves and any finite set of seeds. □

Attempting to compensate for the removal of $\mathcal{C}_{0,1}$ by expanding the set of seeds would require us to add as seeds all the torus triangulations. Generating triangulations for surfaces is an interesting problem which has been studied, for instance, by Barnette [6, 7], but a solution to that problem requires work comparable to our work here; and it is not obvious that the triangulations are the only things we would have to add. So removing $\mathcal{C}_{0,1}$ from the set of moves would almost certainly create more work than it saves.

Lemma 3.2.2 *The set of moves $\{\mathcal{C}_{0,1}, \mathcal{T}_{1,1}\}$ is not sufficient to generate all target embeddings with any finite set of seeds.*

Proof. By starting with a seed embedding and repeatedly replacing edges with the structure shown in Figure 3.1, which we call a super-diamond, we can construct a target embedding which contains an arbitrarily large number of copies of this structure. Indeed, we can choose a target embedding which contains the super-diamond more times than there are edges in any seed embedding. In such a target embedding there must be a super-diamond where none of the seven edges existed in the seed and so all of them were created by moves in our set. We consider what the last move used in the creation of that copy could have been.

Suppose the last move made was a $\mathcal{C}_{0,1}$ move. In that case, the last edge added could not have been one of the five edges incident to at least one of the two degree three vertices. If it was one of the two remaining edges, then the embedding prior to that move must have contained a diamond, and moreover a diamond that did not exist in the seed because all the seed edges are in use elsewhere. We do not have a move in our set that can create a diamond, so this is impossible and the last move cannot have been a $\mathcal{C}_{0,1}$ move.

A $\mathcal{T}_{1,1}$ move must always create a degree three vertex. But the two degree three vertices in the super-diamond are each contained in two distinct triangles. Reversing the $\mathcal{T}_{1,1}$ move from either of these vertices to any of its neighbours would

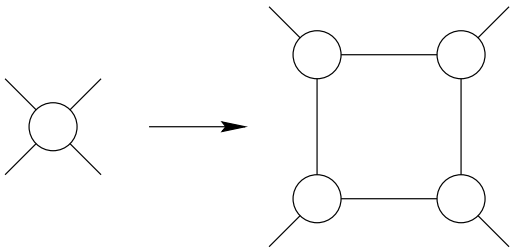


Figure 3.9: Exploding a vertex to create degree three vertices each adjacent to two others.

give us a previous state containing a multiple edge. So the last move cannot have been a $\mathcal{T}_{1,1}$ move.

Therefore there is no way to create more copies of the structure shown in Figure 3.1 than there are edges in any seed, and so for this set of moves and any finite set of seeds, we can always find a target embedding we cannot create. Thus, the set of moves $\{\mathcal{C}_{0,1}, \mathcal{T}_{1,1}\}$ is not sufficient. □

Eliminating $\mathcal{D}_{2,4}$would require us to add as seeds an infinite number of graphs differing from graphs we can generate without $\mathcal{D}_{2,4}$, only by $\mathcal{D}_{2,4}$ moves. This seems at least as complicated as including $\mathcal{D}_{2,4}$ in the set of allowable moves. The proof of Lemma 3.2.2 may seem unnecessarily complicated. A similar proof could be written to use ordinary diamonds instead of super-diamonds. The proof was written as above, using super-diamonds, because the diamond-based proof would involve graphs containing a large number of diamonds. As described in Chapter 4, we wish to avoid diamonds, and to make restrictions on the number of diamonds that may exist in our embeddings. Lemma 3.2.2 as proved here produces counterexamples to sufficiency that contain no diamonds but still require the $\mathcal{D}_{2,4}$ move.

Lemma 3.2.3 *The set of moves $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}\}$ is not sufficient to generate all target embeddings with any finite set of seeds.*

Proof. Neither of these moves can create a degree three vertex adjacent to two other degree three vertices, because $\mathcal{C}_{0,1}$ always increases the degree of two vertices

beyond three without introducing any new ones, and $\mathcal{D}_{2,4}$ introduces two new degree three vertices but makes them each adjacent to two vertices of degree greater than three. As shown in Figure 3.9, we can perform a move on any vertex of a target embedding to create a new target embedding containing degree three vertices each adjacent to two other degree three vertices. We can do this on an arbitrarily large target embedding, so we can produce an infinite number of counterexamples to the sufficiency of $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}\}$. □

Not only does Lemma 3.2.3 provide an infinite number of embeddings we cannot generate with $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}\}$, but the embeddings are of a form that we cannot generate conveniently enough to throw them in as seeds. For instance, the duals of triangulations on our surface are usually if not always three-regular target embeddings. All these would have to be included as seeds. Also, we can generate from any target embedding an exponential number of children, all target embeddings not generated by $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}\}$, by exploding a subset of the vertices in the manner of Figure 3.9. There would be some duplication among those children, but it does not look like an easy way to simplify our experiments. Even if we were to replace $\mathcal{T}_{1,1}$ with some new move for creating degree three vertices with two degree three neighbours, such a move would almost certainly not be easier to implement than $\mathcal{T}_{1,1}$.

We can now prove the main result of this section, the minimality of the three-move set.

Theorem 3.2.4 *The set of move types $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ is sufficient and minimal to generate all target embeddings, with the set of seeds equal to all embeddings on the torus of K_5 and $K_{3,3}$.*

Proof. We have sufficiency from Theorem 3.1.8. By Lemmata 3.2.1, 3.2.2, and 3.2.3, if we remove any one move type the remaining set is not sufficient. Therefore no proper subset of $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$ is sufficient, so this set is minimal. □

As well as being minimal in the technical sense of Section 2.4, this set of three move types appears to be especially convenient when we work with the stated definition of target graphs as having no degree two vertices. In the next section we discuss what might be accomplished by relaxing that requirement.

3.3 A two-move minimal sufficient set

The software written for this research began as a program to generate randomly selected embeddings of toroidal graphs, by starting with a seed K_5 or $K_{3,3}$ and making $\mathcal{S}_{1,1}$ and $\mathcal{C}_{0,1}$ moves. When we later began to consider the question of exhaustive generation without duplicates, we started with that set of moves and a more relaxed definition for target embeddings that permitted them to include vertices of degree two.

Theorem 3.3.1 *The set of moves $\{\mathcal{S}_{1,1}, \mathcal{C}_{0,1}\}$ is sufficient and minimal to generate all combinatorial embeddings of graphs homeomorphic to target graphs, with the set of seeds equal to all embeddings on the torus of K_5 and $K_{3,3}$.*

Proof. If an embedding E is like a target embedding except that it contains one or more vertices of degree two, then we can use a reverse $\mathcal{S}_{1,1}$ move to remove one of the degree two vertices and obtain an embedding E' with fewer edges which is similarly a target embedding except for possibly containing vertices of degree two. If E contains no vertices of degree two, then it is a target embedding, and if E is not a seed we can apply Lemma 3.1.7 to find an edge that we can remove with a reverse $\mathcal{C}_{0,1}$ move to obtain an embedding E', which is a target embedding except for possibly containing some vertices of degree two. Therefore, $\{\mathcal{S}_{1,1}, \mathcal{C}_{0,1}\}$ is sufficient.

The set of moves consisting of only $\mathcal{S}_{1,1}$ is not sufficient because it cannot be used to create embeddings with more vertices of degree greater than two than exist in any seed, and a target-like embedding could contain an arbitrary number of vertices of degree greater than two. Similarly, the set of moves consisting of only

$\mathcal{C}_{0,1}$ cannot create vertices at all, and so cannot create target-like embeddings with more vertices than any seed. Therefore, $\{\mathcal{S}_{1,1}, \mathcal{C}_{0,1}\}$ is minimal. $\square$

If we allow vertices of degree two, then the number of embeddings and therefore graphs we must consider increases without significant improvements in the uses we can make of the results. Any time we would want to embed a graph with degree two vertices on the torus, we could instead eliminate them with reverse $\mathcal{S}_{1,1}$ moves, embed the resulting graph, and then add the vertices of degree two back in afterwards. Degree two vertices have no effect on embeddability.

So the question arose of how many degree two vertices we had to permit in order to be able to generate all target embeddings, and the answer seemed to be three, because we needed to be able to draw a chord across a face, possibly creating one or two new degree three vertices at the ends of the chord, and we needed to be able to create diamonds, either replacing or inserted in the middle of existing edges. Creating diamonds with $\mathcal{S}_{1,1}$ and $\mathcal{C}_{0,1}$, as described below, could require the use of up to three degree two vertices at one time.

If we would be creating degree two vertices only under limited circumstances and only to immediately increase their degree with new edges, then we might as well create and destroy the degree two vertices in one step. That lead naturally to a set of six moves: $\{\mathcal{C}_{0,1}, \mathcal{C}_{1,2}, \mathcal{C}_{2,3}, \mathcal{D}_{2,4}, \mathcal{D}_{3,5}, \mathcal{D}_{4,6}\}$. Each of these moves corresponds to a sequence of $\mathcal{S}_{1,1}$ and $\mathcal{C}_{0,1}$ moves, as shown in Figures 3.10 and 3.11.

The six-move set appeared to be sufficient to generate all target embeddings, but was difficult to implement in practice. The $\mathcal{D}_{3,5}$ and $\mathcal{D}_{4,6}$ moves, in particular, presented difficulties because of the sizes of the subembeddings that had to be constructed and inserted. Our data structures involved two records containing three pointers each for every edge, requiring at least 42 pointers to be updated one by one in order to remove one old edge and add six new ones in a $\mathcal{D}_{4,6}$ move. Although there is nothing in principle difficult about updating a data structure this way, in practice such moves proved cumbersome to implement and debug.

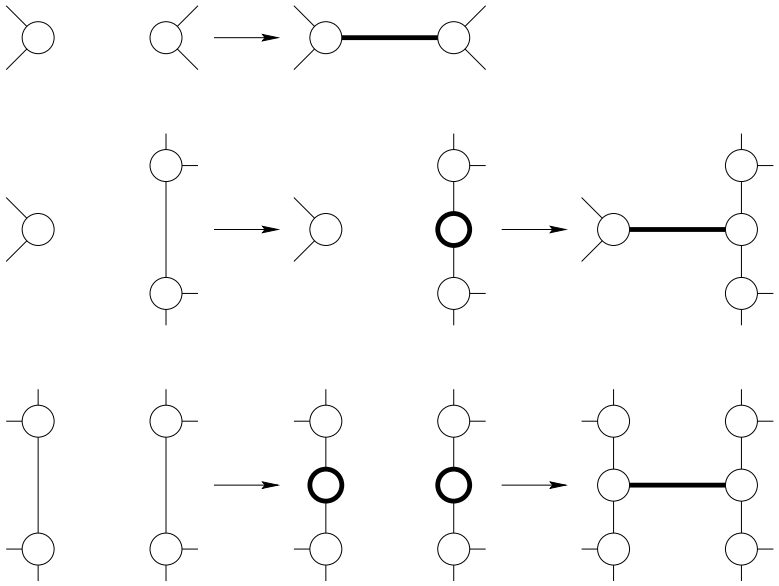


Figure 3.10: Chord moves as sequences of $\mathcal{C}_{0,1}$ and $\mathcal{S}_{1,1}$ moves.

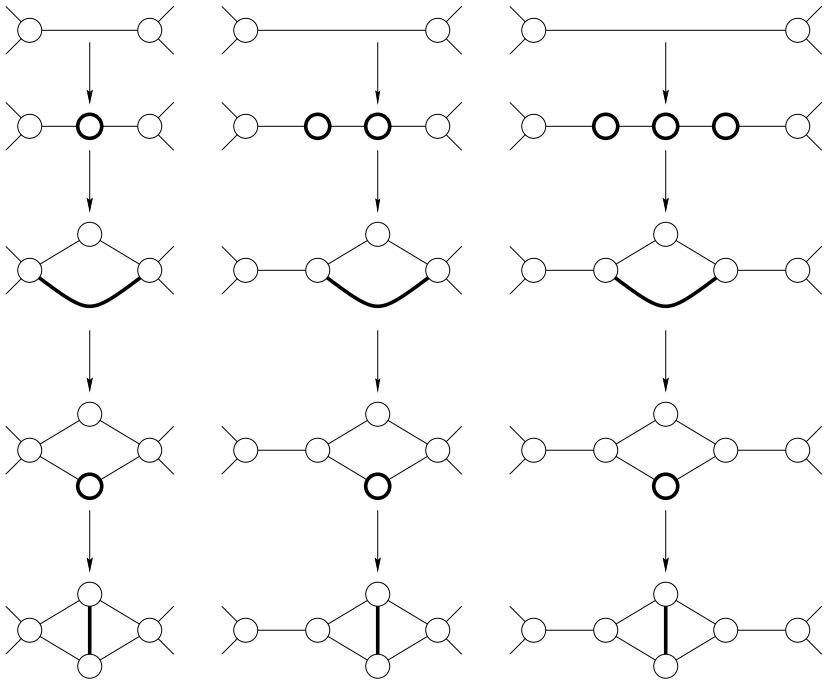


Figure 3.11: Diamond moves as sequences of $\mathcal{C}_{0,1}$ and $\mathcal{S}_{1,1}$ moves.

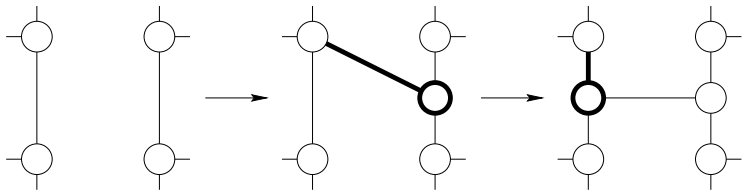


Figure 3.12: Simulating $\mathcal{C}_{2,3}$ with $\mathcal{C}_{1,2}$ and $\mathcal{T}_{1,1}$.

The six-move set, because of its complexity, was also cumbersome to deal with on the theoretical level. It appeared that the sufficiency of the six-move set would be easy enough to prove. We do not prove it here because with the introduction of the three-move set and proof of the two-move set's sufficiency independent of the six-move set, a complicated proof for the six-move set no longer seems useful. It was not clear whether the six-move set was minimal, nor how to prove that.

The $\mathcal{T}_{1,1}$ move was introduced to simplify the set of moves. As shown in Figures 3.12 and 3.13, the use of the $\mathcal{T}_{1,1}$ move along with $\mathcal{C}_{1,2}$ and $\mathcal{D}_{2,4}$ allows us to achieve the effect of the more complicated $\mathcal{C}_{2,3}$, $\mathcal{D}_{3,5}$, and $\mathcal{D}_{4,6}$ moves. That leads naturally to a set of four moves, $\{\mathcal{C}_{0,1}, \mathcal{C}_{1,2}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$. It is not generally possible to simulate a $\mathcal{C}_{1,2}$ move with a $\mathcal{C}_{0,1}$ move followed by a $\mathcal{T}_{1,1}$ move, because if we wished to perform the $\mathcal{C}_{1,2}$ move inside a triangular face, the initial $\mathcal{C}_{0,1}$ move would have to create a multiple edge.

The $\mathcal{C}_{1,2}$ move appears necessary because, if we imagine ourselves generating planar embeddings in the same way we generate torus embeddings, a wheel with many vertices clearly cannot be generated by any of the other moves in the four-move set. A wheel other than K_4 contains no diamonds, so $\mathcal{D}_{2,4}$ is unusable; it contains only one vertex of degree greater than three, so $\mathcal{C}_{0,1}$ is unusable; and a $\mathcal{T}_{1,1}$ move would require the parent to contain multiple edges. It seems reasonable, then, that there should be nonplanar graphs which also require $\mathcal{C}_{1,2}$. The necessity of $\mathcal{C}_{1,2}$ appeared so obvious that its proof could safely be left almost to the end of the project. So our intent during most of the project was to prove sufficiency of the six-move set, then sufficiency of the four-move set by the equivalences above.

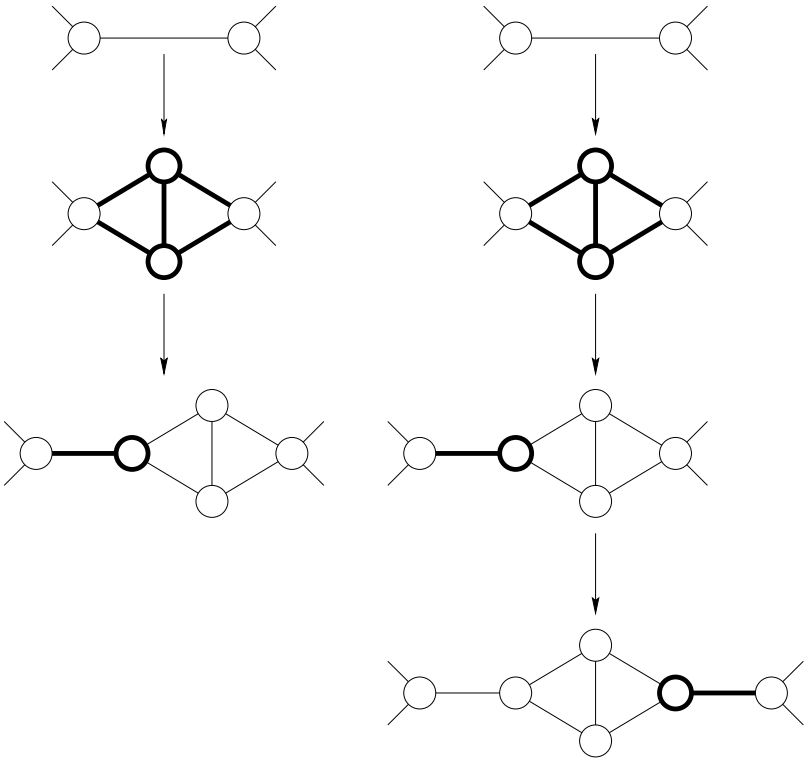


Figure 3.13: Simulating $\mathcal{D}_{3,5}$ and $\mathcal{D}_{4,6}$ with $\mathcal{D}_{2,4}$ and $\mathcal{T}_{1,1}$.

Then we would prove minimality of the four-move set, possibly with a note on the possibilities of using the original two moves, for a more wasteful but much simpler approach.

The discovery that $\mathcal{C}_{1,2}$ was not necessary, or at least not necessary when dealing with embeddings on the torus, was triggered by the difficulty of proving minimality of the four-move set. Despite the note above that $\mathcal{C}_{1,2}$ cannot be directly simulated by $\mathcal{C}_{0,1}$ and $\mathcal{T}_{1,1}$, we could not actually find any target embeddings for which it was necessary; not even in our computer experiments with hundreds of millions of embeddings. Obtaining a sufficiency proof, showing that in fact the four-move set is not minimal, was difficult but eventually possible.

Unfortunately, we have no simple explanation for why $\mathcal{C}_{1,2}$ can be eliminated; unlike $\mathcal{C}_{2,3}$, there is no easy sequence of other moves that can replace $\mathcal{C}_{1,2}$ in all cases. At best we can point to Theorem 3.1.8, which shows (after a complicated argument with several cases) that any target embedding which might appear to require $\mathcal{C}_{1,2}$, can be generated in some other way with the other three moves. Although the three-move set requires an elaborate proof and appears to be in some sense just barely sufficient, it is sufficient, and having only three moves simplifies the software a great deal.

Chapter 4

Diamonds

In this chapter we discuss a further refinement of the algorithm, intended to make the output more useful by eliminating embeddings of less interesting graphs. Graphs containing diamonds present fewer challenges in embedding because we can simply eliminate the diamonds, embed the remaining graphs, and then reinsert the diamonds. We begin with a description in Section 4.1 of diamonds and their consequences. Then in Section 4.2 we prove that the algorithm can be limited to examining target graphs with at most one diamond, and still generate all the diamond-free target graphs. Finally, in Section 4.3 we discuss how diamonds can be embedded in several ways on the torus, and the consequences of that fact for our work.

4.1 Some notes on diamonds

We have already mentioned that the presence of diamonds in a graph does not change its genus, since we can always perform a reverse $\mathcal{D}_{2,4}$ move to find a smaller graph embeddable on exactly the same surfaces. Just like degree two vertices and multiple edges, diamonds can be viewed as uninteresting embellishments to existing edges. For any graph G that we consider really interesting, there will be a large number of less interesting graphs consisting of G with one or more diamonds sub-

stituted into its edges. It would be preferable to eliminate them from consideration.

But as described in Chapter 3, forbidding diamonds entirely, by eliminating the $\mathcal{D}_{2,4}$ move, would require us to have some other way to create the more complicated structures which currently require diamonds. Perhaps we could make diamonds unnecessary by requiring target graphs to be 3-connected, but then the proofs that we can maintain that constraint, already difficult for biconnectedness, could become even more difficult. Also, some intended applications of the output, for instance to the search for torus obstructions (see Section 7.2) would suffer if the output were limited to 3-connected embeddings. It seems useful to permit at least a few diamonds.

We could make attempt to place a similar limit on how many degree two vertices are necessary at any one time to generate all target embeddings with moves in $\{\mathcal{S}_{1,1}, \mathcal{C}_{0,1}\}$, as discussed in Section 3.3, but it appears that we would still need at least three degree two vertices to be able to generate diamonds with those moves. Similarly, if we permitted multiple edges we might also need to permit enough of those to be inconvenient.

Permitting diamonds presents less of a problem than permitting multiple edges or degree two vertices, because every diamond includes two vertices which are not available to be included in any other diamond. When we generate graphs up to a fixed number of vertices n, the n-vertex graphs containing diamonds must correspond to graphs with $n-2$ or fewer vertices, and the fast growth in the number of embeddings with increasing n guarantees that there will be far fewer target embeddings with $n-2$ vertices than with n vertices. So the actual number of diamond-containing embeddings should not be overwhelming.

Nonetheless, we choose to avoid diamond-containing embeddings as far as possible. Lemma 3.2.2 shows that some diamonds are necessary with our move set, so we cannot simply require all parents and children to be diamond-free and expect to generate all other target embeddings that way. In the next section we show that we need tolerate only one diamond in an embedding at a time; the set of target

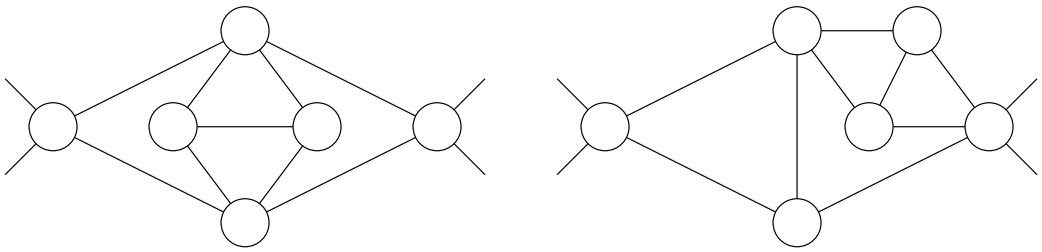


Figure 4.1: The two situations where a reverse $\mathcal{D}_{2,4}$ move would create a diamond

embeddings with at most one diamond can be generated without requiring the use of embeddings with more than one diamond.

4.2 Only one diamond is necessary

In order to create graphs that do not contain diamonds but do contain things like super-diamonds, we need to make use of parents containing diamonds. How many diamonds must we tolerate in parents in order to be able to generate all diamond-free children? The following theorem shows that the answer is just one.

Theorem 4.2.1 *If C is a target embedding containing at most one diamond, then either C is a seed or C can be obtained from a target embedding P with fewer edges and containing at most one diamond, by a move in $\{\mathcal{C}_{0,1}, \mathcal{D}_{2,4}, \mathcal{T}_{1,1}\}$.*

Proof. By Theorem 3.1.8, we can find a reverse move from any non-seed C to some target embedding. We consider the ways that such a reverse move could create a diamond, and show that we can always find a reverse move that will not increase the number of diamonds past one.

A reverse $\mathcal{D}_{2,4}$ move simultaneously reduces the degree of two vertices and makes them adjacent to each other. However, it also destroys a diamond. There are two cases: the new edge could form the crossbar of the diamond (becoming the diamond edge as such) or it could go into the side of the diamond. These two cases are shown in Figure 4.1. In each case, it is clear from the figure that the effect of the reverse

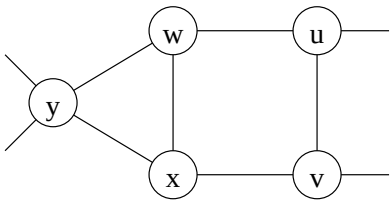


Figure 4.2: How a reverse $\mathcal{T}_{1,1}$ move can create a diamond.

move is limited to the vertices shown. With the new edge forming the crossbar of the diamond, both its endpoints can only be part of that one diamond, so only one diamond is created. With the new edge forming a side of the diamond, only one vertex has its degree reduced to three, and so, again, only one diamond is created. Since one diamond is always destroyed by the reverse $\mathcal{D}_{2,4}$ move, this reverse move can never increase the total number of diamonds in the embedding. Theorem 3.1.8 has already established that P is a target embedding when it is obtained by a reverse $\mathcal{D}_{2,4}$ move. Note that in the exceptional case of making a $\mathcal{D}_{2,4}$ move on K_4, the resulting graph C has two diamond edges in it and a reverse $\mathcal{D}_{2,4}$ move gives us the original K_4, with six diamond edges. We consider only target graphs with at most one diamond, so that situation is excluded.

A reverse $\mathcal{T}_{1,1}$ move always increases the degree of a vertex past three, so it can never create a diamond by creating a degree three vertex. Any diamond created by a reverse $\mathcal{T}_{1,1}$ move must result from the contraction of a 4-cycle in the target embedding, as shown in Figure 4.2. The edge (u, v) is the one being contracted. Such a contraction could in fact create two diamonds at once, if we imagine the structure repeated again on the other side of the edge (u, v), as shown in Figure 4.3. We can instead contract the edge (u, w) in C to obtain P unless u and y are adjacent, or the edge (v, x) unless v and y are adjacent. The vertices u and v cannot both be adjacent to y, because then the original reverse $\mathcal{T}_{1,1}$ move would have been forbidden for creating a multiple edge.

We can assume without loss of generality that u is not adjacent to y, and so we can contract (u, w). The reverse $\mathcal{T}_{1,1}$ move always preserves connectivity and the

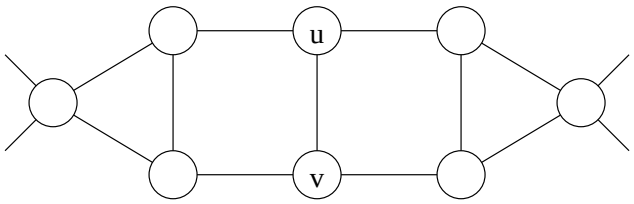


Figure 4.3: How a reverse $\mathcal{T}_{1,1}$ move can create two diamonds.

minimum degree of vertices; no multiple edges are created because we are using it to contract an edge that is not on any triangle. It remains only to show that we can preserve nonplanarity.

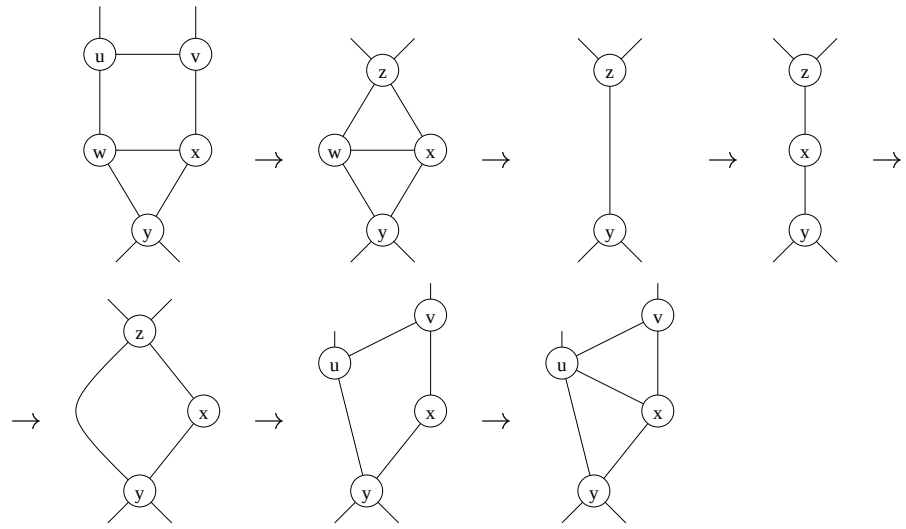


Figure 4.4: Why contracting (u, w) does not change the genus.

Suppose we contract (u, v), notwithstanding that it would create a diamond, and label the resulting vertex z. Since that was the reverse move chosen by Theorem 3.1.8, the resulting graph is still nonplanar, although it might contain too many diamonds. Then we eliminate vertices w and x with a reverse $\mathcal{D}_{2,4}$ move, which by Theorem 3.1.1 cannot make the graph planar. We subdivide the edge (y, z), labelling the new vertex x; that also cannot make the graph planar. Then we add a new edge from y to z; adding an edge cannot make the graph planar. We split z back into u and v, as they were before; splitting a vertex cannot make the

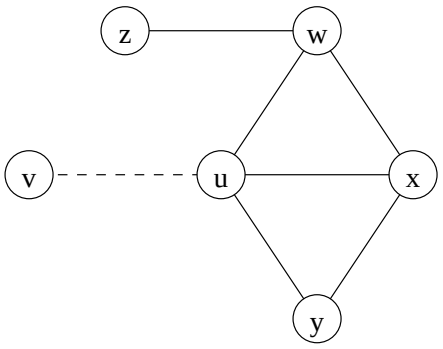


Figure 4.5: How a reverse $\mathcal{C}_{0,1}$ move can create a diamond.

graph planar. Finally we add an edge from x to u. This process and its result are shown in Figure 4.4. We have obtained, by a series of operations that maintain the nonplanarity of the graph, exactly the same embedding we would have obtained by contracting (u, w) in the original embedding C. Therefore this embedding, which we call P, is of a nonplanar graph and so P is a target embedding containing no more diamonds than C contained.

A reverse $\mathcal{C}_{0,1}$ move can create a diamond only by reducing the degree of a degree four vertex which (except for the edge being removed) forms a diamond with a degree three vertex, as shown in Figure 4.5. The edge (u, v) is the one being removed. We know that (u, v) could not have been red in the colouring used in Theorem 3.1.8, because then the edge removal would not have been permitted. Note that v could also be degree four and create a diamond, so that this reverse move could create two diamonds at once.

If one of w and y, say without loss of generality w, has degree greater than three, then we will remove the edge (u, w) instead of (u, v). Suppose we did remove the edge (u, v), even though it would make the graph contain too many diamonds; since that is the reverse move found by Theorem 3.1.8, the resulting graph must be nonplanar. We then apply a reverse $\mathcal{D}_{2,4}$ move to replace the diamond by an edge (w, y); this must leave a nonplanar graph G by Theorem 3.1.1. If we colour the resulting graph with a red subgraph homeomorphic to K_5 or $K_{3,3}$, we can then

replace the diamond and then (u, v); in so doing we colour the replaced edges (w, x) and (x, y) red if the edge (w, y) was red, and colour no other new edges. The result is a red-coloured subgraph homeomorphic to K_5 or $K_{3,3}$ in the original C, with the edge (u, w) not red. Therefore we can remove the edge (u, w) to obtain a nonplanar graph P.

Since $C - (u, v)$ is known to be biconnected and w has degree greater than three, we can choose a neighbour z of w, where z is not u, x, or y, and find a path in $C - (u, v)$ from z to y without passing through w. Then that path, plus the edges (w, z) and (y, u) provide one path from w to u in P; and the path $\langle w, x, u\rangle$ is a second path from w to u in P, internally vertex disjoint from the first. Therefore removal of (u, w) leaves P biconnected. Since both u and w have degree greater than three, every vertex of P has degree at least three. All the conditions are now satisfied and P is an embedding of a target graph.

If w and y both have degree three in C, then we cannot remove the edge (u, w) with a reverse $\mathcal{C}_{0,1}$ move to obtain a target embedding. But then we can contract the edge between w and its neighbour z with a reverse $\mathcal{T}_{1,1}$ move to obtain P. Such a reverse move necessarily preserves biconnectedness and minimum degree. It cannot create multiple edges unless z and v are the same vertex, a case which we handle later. If w is degree three and we are contracting (w, z), then it only remains to show that this leaves the graph nonplanar. The vertices w and y cannot both be degree three and adjacent to v, or v would be degree two or a cut vertex.

It only remains to show that the graph in P is nonplanar. As above, we imagine removing (u, v) from C even though it would create a diamond, replacing the diamond with an edge (w, y), and colouring the result with a red subgraph homeomorphic to K_5 or $K_{3,3}$. Reversing these steps and maintaining the colouring, we obtain a colouring for C where the edges (w, u) and (w, x) are not both red. Then by Lemma 3.1.3, we can contract the edge (w, z) and leave the graph nonplanar. Therefore P is a target embedding.

One possibility remains with the reverse $\mathcal{C}_{0,1}$ move: that w could be degree three and v and z could be the same vertex, so that there is a triangle with vertices $\{u, v, w\}$ and we cannot contract the edge (w, v) without creating a multiple edge. In that case, the graph C must contain a super-diamond, as shown in Figure 3.1. Obviously, an embedding (either the target embedding C or some ancestor necessary to create it) could contain an arbitrarily large number of super-diamonds. However, the super-diamond does not contain a diamond itself, and it can be created from an edge by making a $\mathcal{D}_{2,4}$ move followed by a $\mathcal{T}_{1,1}$ move and a $\mathcal{C}_{0,1}$ move, creating and destroying one diamond along the way. So if we need to build an embedding containing one or more of these and possibly a diamond as well, we can first build the corresponding embedding with the diamond and super-diamonds replaced by edges and any degree two vertices and multiple edges eliminated. Then we can insert the diamond and super-diamonds, one at a time, never having more than one diamond in the embedding at one time, and then we can do any final splitting and adding of edges as in Theorem 3.1.4 to create the desired embedding. □

If we are primarily interested in embeddings of diamond-free graphs, then Theorem 4.2.1 allows us to prune our computation tree considerably. Noting that our seed embeddings, of K_5 and $K_{3,3}$, are all diamond-free, we can implement the pruning in the generator software simply by discarding any parents or children that have more than one diamond. We also restrict actual output to embeddings of diamond-free graphs, although it is necessary to examine embeddings with one diamond in order to generate all diamond-free embeddings.

4.3 Twisted diamonds

We normally imagine diamonds as being embedded nicely on the plane, as in the drawing in Figure 2.1. But on the torus, there are more possibilities. Two other ways to embed a diamond on the torus are shown in Figure 4.6. We call any

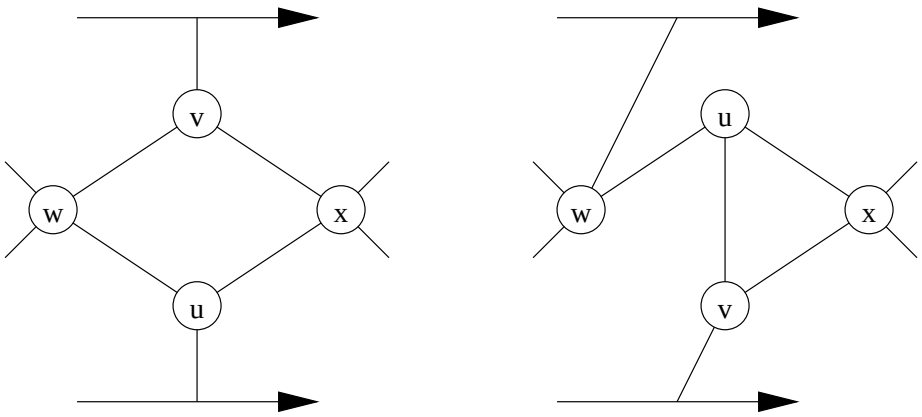


Figure 4.6: Some twisted diamonds.

diamond that is not embedded in the obvious planar way depicted in Figure 2.1, a *twisted diamond*.

Twisted diamonds are necessary because some target embeddings, even some without diamonds like the one shown in Figure 4.7, can only be generated from ancestors that contain twisted diamonds. If we imagine making reverse moves on Figure 4.7, the only target embeddings we can find as possible parents, are embeddings that contain twisted diamonds. Theorem 4.2.1 shows that we need tolerate only one diamond, but it might happen that that diamond must be twisted.

So we cannot simply forbid twisted diamonds; our $\mathcal{D}_{2,4}$ move must be able to create them. The definition of the $\mathcal{D}_{2,4}$ move given in Section 2.4 is designed to be able to create any possible twisted diamond, and the move label for it described in Section 5.2 is designed to describe any possible $\mathcal{D}_{2,4}$ move. Here, we describe the $\mathcal{D}_{2,4}$ move in detail, to clarify its operation.

Let (u, v) be a diamond edge in a target embedding C, and let w be the next vertex after v in the clockwise adjacency list of u and x be the remaining neighbour of u. This is the same naming of diamond vertices shown in Figure 3.3 earlier, but here we emphasize that the diamond could be embedded in any of several ways; some other embeddings of the same subgraph are shown in Figure 4.6. In all those diagrams, the clockwise adjacency list of u is $\langle v, w, x \rangle$.

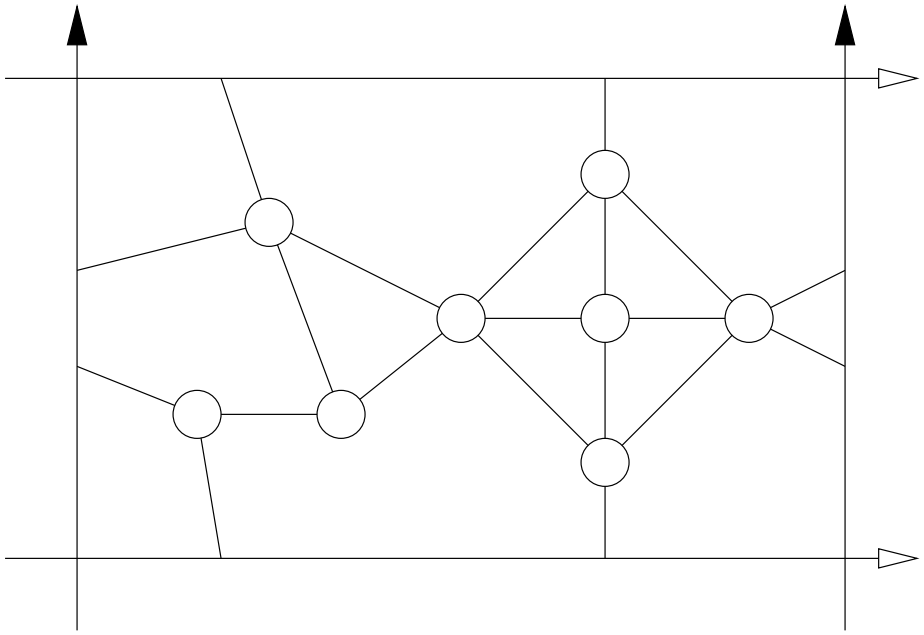


Figure 4.7: A diamond-free target embedding that cannot be generated without a twisted diamond.

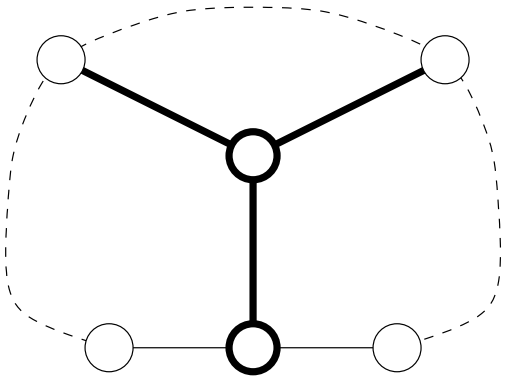


Figure 4.8: Illustration of the general $\mathcal{D}_{2,4}$ move.

If we imagine removing the vertex v and all its incident edges, then we would have an embedding P', which is the same as the embedding P obtained by replacing the diamond with a path $\langle w, u, x \rangle$. To obtain C from P', we must insert v into a face containing the vertex u; v goes inside that face and its incident edges connect it to some appearance of each of u, w, and x around the face. Because v must come after x and before w in the clockwise adjacency list of u, there is only one way to add the edge (u, v) even if u appears twice on the face. The other two edges incident to v, however, may attach to any appearances of w and x on the face. We can break down the addition of v and its edges further into the steps of adding an edge between the two appearances of w and x that will be adjacent to v, then subdividing that edge to create v, and finally adding the edge (u, v).

To remove a diamond, we can label it as above, remove v and its edges, then remove y and replace the edges (w, u) and (u, x) with an edge (w, x). We can remove any diamond this way, twisted or not. By reversing the steps and making sure that we can choose any appearances of w and x on the face, we obtain the definition of the $\mathcal{D}_{2,4}$ move, which can create any diamond, twisted or not. In detail, the steps are as follows:

1. Choose a face F of an embedding P.
2. Choose an edge e in F.
3. Let w and x be the endpoints of e so that w comes immediately before x in a clockwise traversal of F. There may be two ways to do that if e appears twice on the same face.
4. Choose one appearance of each of w and x on F (each may appear more than once).
5. Subdivide (w, x), creating u.

6. Add an edge through F between the chosen appearances of w and x. This divides F into two distinct faces.

7. Subdivide the new edge, creating v. Because the edge appears on two distinct faces, v can only appear once on any given face.

8. Add an edge from v to u, so that v appears before w and after x in the clockwise adjacency list of u. There is only one way to do this because v appears only once on any given face.

These steps can insert any embedding of a diamond, no matter how many times a vertex may appear on a face in P. When looking for all possible moves to apply to P in the algorithm GENERATE (see Section 2.5), we loop through all possible choices of F, u, v, and the additional appearances of u and v. Then we are sure of examining every possible twisted diamond and thus generating all diamond-free target embeddings. In the next chapter we discuss how to recognize and prevent duplication, so that even if we can describe the same diamond move in more than one way (for instance, by making a different choice about which vertex is u), we will still only generate each child once.

Chapter 5

Other aspects of the algorithm

There still remain some details which have not been described but which are necessary for the implementation of the algorithm. Those details are discussed in this chapter. First, in Section 5.1, we describe a canonical form for embeddings, which leads to an isomorphism test. In Section 5.2 we discuss move labels, used to ensure that each equivalent move is made exactly once from each parent. Our algorithm requires the use of a planarity testing algorithm, and in Section 5.3 we describe an enhancement used to reduce the number of planarity tests we must perform.

5.1 A canonical form for embeddings

The generation algorithm as described requires us, each time we derive a possible child embedding C from a possible parent P, to find the actual parent for C. Then we recurse to C and its descendants if and only if P is the parent of C. This limitation prevents us from processing C at more than one place in the computation tree. Saying that P must be the parent of C begs the question of how to compare P with PARENT(C). Checking all possible labellings would consume $O(n!)$ time for an n-vertex embedding. Since embeddings include graphs, and graph isomorphism is a difficult problem, not known to be solvable in polynomial time, the need for an

embedding isomorphism test may appear to be a significant obstacle.

Fortunately, embeddings are much easier to compare than graphs. Given a labelled combinatorial embedding of a connected graph, we can generate in $O(n^2)$ time, using the algorithm below, a sequence of symbols representing the isomorphism class of the embedding. That sequence of symbols is called the canonical form; two embeddings have the same canonical form if and only if they are isomorphic.

Suppose we have a combinatorial embedding, and we have already chosen a direction (clockwise or counterclockwise) and assigned the labels zero and one to two adjacent vertices. We perform a breadth-first search, starting with the vertex labelled zero as the root and using the vertex labelled one as its first child. At each vertex, we visit the neighbours in the chosen direction, clockwise or counterclockwise, starting from the parent. The order of edges visited by this search is then fully determined. When the vertices of the embedding are labelled with nonnegative integers, we can record the breadth-first search with a sequence of integers. It remains only to choose which traversal direction and pair of adjacent vertices to use. One obvious way to make that choice would be to try all possible starting points and use the lexically least representation of the embedding.

However, doing four breadth-first searches for every edge in the embedding seems inefficient. It would be preferable to reduce the number of possible starting points as far as possible. As described below, we begin our canonical form with the integers $\langle n, m, f \rangle$, the counts of vertices, edges, and faces respectively. Those are the same for all representations of the embedding. We then insert two more integers before recording the sequence generated by the breadth-first search, namely $\langle n - \mathit{degree}(u), n - \mathit{degree}(v) \rangle$ where u is the starting vertex (labelled zero) and v is its first neighbour (labelled one). Since the three terms $\langle n, m, f \rangle$ are the same in all sequences representing the embedding, then the lexically least sequence must necessarily have minimum possible $n - \mathit{degree}(u)$, and minimum $n - \mathit{degree}(v)$ subject to the previous condition. In other words, u must have maximum degree among

the vertices in E, and v must have maximum degree subject to that.

These conditions limit the number of possible starting points for the search; if there is a degree five vertex in E, for instance, then we need not run the search for any u with degree three or four; we know that the result could not be lexically least. We chose to maximize the degrees of u and v instead of minimizing them, because we expect our embeddings to usually have relatively many vertices of small degree and relatively few of large degree. Choosing maximum-degree vertices for the starting point should tend to give a smaller number of starting points to examine. Of course there are embeddings where many or all vertices have the maximum degree, and then this condition gives little or no speed benefit; but it is cheap to implement, and in practice it saves time often enough to provide a significant speed improvement overall.

We use the integer -1 to represent the end of an adjacency list. The sequence recording the breadth-first search then consists of three integers denoting the number of vertices, edges, and faces in the embedding, two more integers to force a desirable ordering as described above, and then the adjacency lists of vertex zero, vertex one, and so on, up to the last vertex in the embedding. Each adjacency list is in the order determined by the breadth-first search, and terminated by -1; the breadth-first search also assigns the vertex labels except for zero and one. The pseudocode below describes the breadth-first search to label an embedding E, starting from adjacent vertices u and v and traversing in direction d, which is clockwise or counterclockwise.

BFS(E, d, u, v) :
initialize *seq* with $\langle n, m, f \rangle$, the numbers of vertices, edges, and faces in E
append $\langle n - \mathit{degree}(u), n - \mathit{degree}(v) \rangle$ to *seq* (explained above)
all vertices begin unlabelled
label u with 0
label v with 1

nextlabel $\leftarrow 2$
for $i \leftarrow 0$ to $n-1$
    $w \leftarrow$ vertex labelled i
    for each neighbour x of w, starting with the one with minimum label
        among those that have a label, and proceeding in the direction d
        if x has no label yet
            label x with *nextlabel*
            nextlabel $\leftarrow$ *nextlabel* $+ 1$
        end if
        append label of x to *seq*
    end for
    append -1 to *seq*
end for
return *seq*

If we run this traversal on E with all possible values of d, u, and v (note that when (w, x) is an edge in E, we must try both $u = w, v = x$ and $u = x, v = w$), then the lexically least result is the canonical form for the embedding. Since we know that u must have maximum degree and v must have maximum degree subject to that, we need only run the traversal for values of u and v satisfying those conditions. Since this sequence includes an ordered adjacency list for every vertex, it is easy to construct a combinatorial embedding isomorphic to E from the canonical form. Thus, two embeddings with the same canonical form must be isomorphic. Conversely, two isomorphic embeddings must have the same canonical form. Changing the labelling of vertices or the starting point of a list has no effect on the canonical form because the breadth-first search determines its own labelling and starting points. Reversing all adjacency lists (mirror-reversing the embedding) has no effect on the canonical form because we make the search both clockwise and counterclockwise.

As a side effect of the canonical form calculation, we obtain the automorphism group of the embedding in the form of a list of all permutations from the original labelling to a labelling that yields the canonical representation. If the embedding has some symmetry, then there will be more than one starting point that produces the canonical representation, so there will be more than one such permutation. Since each permutation is generated by one of our breadth-first searches, and there can be at most $4m$ of those in a graph with m edges (two starting points for each edge multiplied by two for clockwise or counterclockwise), that provides an upper limit on the number of permutations. After applying to the embedding the canonical labelling, which is a permutation π_0 from the list of permutations we generated, we also replace each entry π_k with $\pi_k \circ \pi_0^{-1}$, so that we have the automorphism group as the set of permutations of the canonical vertex labels that leave the breadth-first search representation unchanged. The automorphism group is used with move labels in eliminating duplicate moves.

5.2 Move labels

As we observed when attempting to implement an embedding generator, we can eliminate almost all duplication of isomorphic embeddings in the output by finding a parent for each possible child, and keeping the child if and only if it was generated from the parent according to the canonical form above. This restriction prunes the computation tree a great deal. However, it is still possible that a child could be kept more than once, resulting in duplicate embeddings in the output. That could happen if there is more than one way to make a move on the parent to produce the same child.

If the parent is highly symmetric, there could be many duplicate moves. For example, in Figure 5.1, any of four edges can be replaced by a diamond to give the pictured child. The software must have a way to recognize that these four edges are equivalent, and only apply the move to one of them.

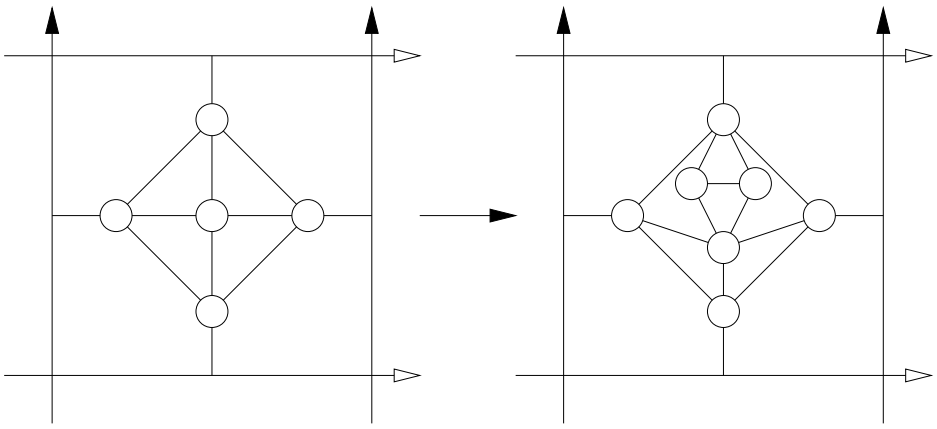


Figure 5.1: A $\mathcal{D}_{2,4}$ move can be applied to any of four edges in this parent to give the same child.

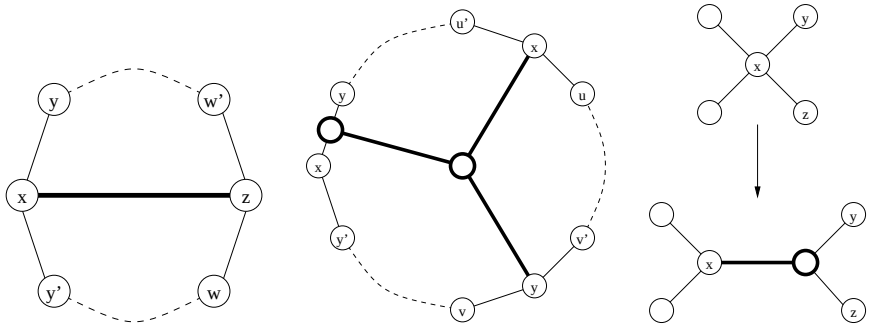
We address this need by assigning a name called a move label to each way we can make a move on the parent. The move label is a sequence of vertex labels. We already know the automorphism group of the parent because we computed that as a side effect of computing the canonical form. So by applying each element of the automorphism group to the move label of the move under consideration, and taking the lexically least result, we obtain a canonical form for the move label. Then we actually make the move if and only if its move label matches the canonical form.

Lemma 5.2.1 *If the vertex sequence $\langle u, v, w \rangle$ occurs consecutively clockwise or counterclockwise around some face of an embedding of a graph G with no vertices of degree less than three, then it does so only once in the entire embedding.*

Proof. The occurrence of this sequence once means that u and w are both neighbours of v, and moreover that they appear consecutively in the cyclic adjacency list of v. Since we do not allow multiple edges, u and w can each only appear once in the cyclic adjacency list of v. If the sequence of labels $\langle u, v, w \rangle$ on a face occurs more once in the embedding, then u and w must be consecutive on both sides: the next neighbour of v after u must be w in both the clockwise and counterclockwise directions. Therefore u and w must be the only neighbours of v, which contradicts

the definition of G as having no vertices of degree less than three. □

Lemma 5.2.1 means that by giving the labels of three consecutive vertices $\langle u, v, w\rangle$ on a face, we can uniquely identify the face, the particular appearances of those three vertices on the face, and a direction for traversing the face. Conceptually, we are identifying an appearance of v; to disambiguate the many places it may appear in the embedding, we give its successor around the face, w. To specify which face we mean, if there are two containing that edge, we also give the immediate predecessor of v, which is u; that also identifies the direction of traversal.



(a) $\mathcal{C}_{0,1}$ is labelled by $\langle y', x, y, w', z, w\rangle$.

(b) $\mathcal{D}_{2,4}$ is labelled by $\langle y', x, y, u', u, v', v\rangle$.

(c) $\mathcal{T}_{1,1}$ is labelled by $\langle x, y, z\rangle$.

Figure 5.2: How to label moves.

As a result, we can label a $\mathcal{C}_{0,1}$ move with a sequence of six vertex labels: three to identify one endpoint of the new edge, and three to identify the other endpoint. A $\mathcal{T}_{1,1}$ move is labelled with the vertex to split, and the two neighbours that will become neighbours of the degree three vertex created by the move. Labelling a $\mathcal{D}_{2,4}$ move is more complicated. Conceptually, this move consists of adding a new edge between two vertices that are already adjacent, then subdividing the old and new edges and adding another edge between their midpoints. We use three vertex labels $\langle y', x, y\rangle$ to identify the old edge and the face in which we operate. We must

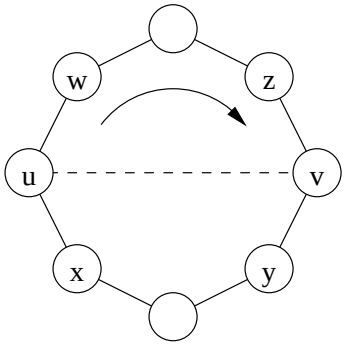


Figure 5.3: A potential $\mathcal{C}_{0,1}$ move, which could be labelled in four different ways.

then specify where to draw the new edge, by naming its two endpoints. The label $\langle u', x, u\rangle$ names one endpoint and $\langle v', y, v\rangle$ names the other, as for the $\mathcal{C}_{0,1}$ move, but since x and y were already specified in naming the old edge, we need only seven numbers to name the entire move: $\langle y', x, y, u', u, v', v\rangle$.

One problem with this approach is that there may be several inequivalent ways to label the same move. For instance, with the $\mathcal{C}_{0,1}$ move, we could label it clockwise or counterclockwise starting from either endpoint of the edge being created. If the automorphism group of the parent is trivial, all four of these would result in different canonical labellings. When assigning the canonical move label to a move, then, we must find the lexical minimum of all images of all possible labels for the move. When we evaluate possible moves in a software loop, we carefully decide which labellings will be associated with which iterations of the loop, and take the least of the labellings assigned to the current iteration as the thing to compare with the canonical image.

For instance, our move-selection process for the $\mathcal{C}_{0,1}$ move goes around each face in only one direction, clockwise or counterclockwise, but attempts to draw a chord across the face from every vertex to every other vertex on the face. In Figure 5.3, we traverse a face clockwise and attempt to draw the chord between vertices u and v twice: once conceptually from u to v and once from v to u. When drawing the chord from u to v we label it with the least of $\langle x, u, w, z, v, y\rangle$ and $\langle w, u, x, y, v, z\rangle$

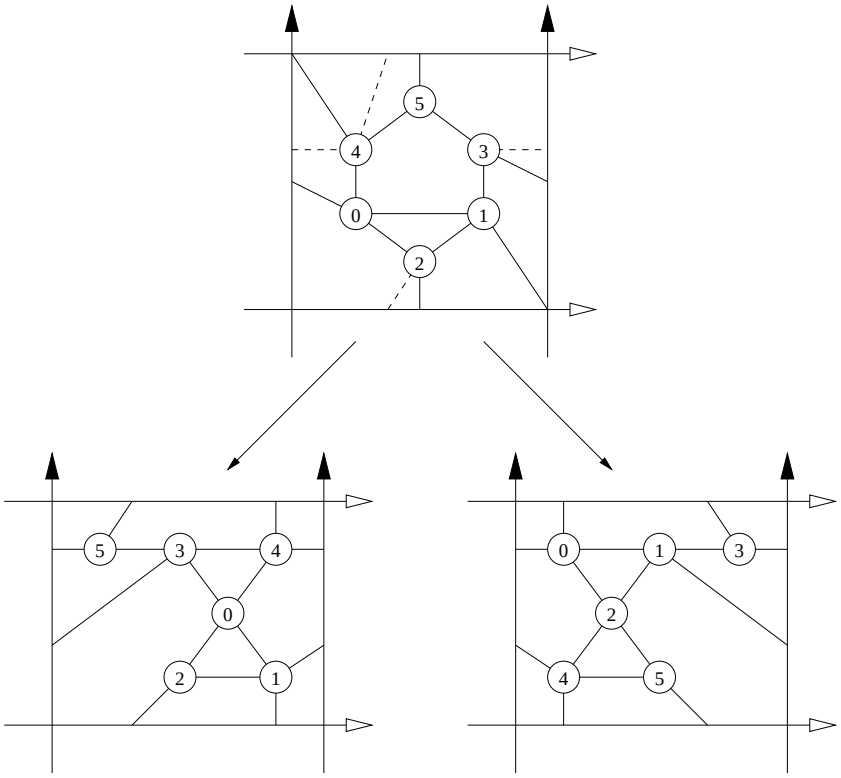


Figure 5.4: Two different moves may create the same child from the same parent.

and when drawing the chord from v to u we label it with the least of $\langle z, v, y, x, u, w\rangle$ and $\langle y, v, z, w, u, x\rangle$. The canonical label for the move is the least image of any of these. Assuming that the automorphism group is trivial, we will decide to make the move exactly once on this face. If the automorphism group were not trivial we might make the move on some other face instead; but in any case, we would make it exactly once in the embedding.

A more serious problem occurs when two inequivalent moves lead from the same parent to the same child. Figure 5.4 shows an example of such a situation. First, note that the parent's automorphism group consists of the identity and a permutation that swaps vertex zero with vertex one and vertex three with vertex four. Two different $\mathcal{C}_{0,1}$ moves are shown by dashed lines. The canonical move label for one is $\langle 0, 2, 5, 5, 4, 1\rangle$ and for the other $\langle 0, 3, 5, 1, 4, 0\rangle$. These moves are not equivalent; for instance, one endpoint of the first move is on a triangular face and

that is not the case for the second move.

The lower half of the figure shows the result of each move, using the same vertex labelling as in the parent to make clear what happens to the vertices. Although the drawings of the children have been adjusted to show their relationship to each other rather than to the parent, it is clear by careful examination of each vertex that these are the children produced by the two moves. As is also clear from the diagram, these two children are the same up to a mirror reversal and relabelling of the vertices. They have the same canonical form.

Fortunately, move labels provide an easy solution to this kind of problem as well. When we examine a move we could make on an embedding P, we first check that the move's label matches its own least image under the automorphism group of P. If it does, we construct the resulting embedding C, construct the parent of C, and check that the parent is isomorphic to P. So far we have done nothing to prevent the situation of Figure 5.4. But when we call PARENT(C), it also returns a move label describing a way to get from the parent to C. We then check not only that P matches the parent of C, but also that the move label we used to find C matches the move label returned by PARENT(C).

Just as with selection of parents, it does not matter at all how we choose the move label to return from PARENT, provided we choose some move label that actually will be visited and does lead from the parent to the child. However, PARENT(C) must always choose the same move label for all isomorphic values of C, and the easiest way to be sure of that is to force PARENT to examine only the canonical form of C. Any additional information available to the software, as for instance the edge marks described in the next section, is carefully excluded from influencing the selection of the parent and move label.

5.3 Edge marking

When we select a parent for a given C, it is important that the parent be an embedding whose children we actually will examine. Otherwise, the child will never be processed. Since we examine only target embeddings, the parent must be a target embedding; therefore, it must be biconnected, have no vertices of degree less than three, and have orientable genus one. If we are limiting the number of diamonds to at most one, then we look for a parent containing at most one diamond. To generate embeddings efficiently, we must be able to quickly test, or avoid testing, each of these conditions.

Most of the target embedding conditions are easy to deal with. Genus no greater than one, for instance, is guaranteed because the child has genus one and none of our reverse moves can increase the genus. Minimum degree of vertices is easy to assure. We simply forbid making any reverse move that would reduce the degree of a vertex to less than three. Checking for biconnectedness requires a simple, linear-time traversal of the graph. But it may be much more expensive to check that a possible parent is not planar.

Some planarity testing algorithms are simple to implement but do not achieve linear time complexity, like the one known as Demoucron's Algorithm [15]; others are linear-time but require complicated structures like PQ-trees [10, 14]. The linear-time planarity algorithm of Boyer and Myrvold [11] is designed for easy implementation, but is still complicated enough to present some problems. So if we make an embeddability test, or possibly several of them, for every potential child we visit, then we could spend most of our programming labour or computation time doing that alone.

First of all, we can arrange the tests we apply to potential parents in order of increasing cost, so that if we can reject a potential parent for a reason we can determine cheaply, we will do so and avoid doing the more expensive tests. But it still seems undesirable to do planarity testing if we can possibly avoid it, especially

as the graphs become larger. On a very large toroidal graph, for instance, it seems unlikely that removing any one edge would ever render it planar.

We reduce the number of planarity tests by using *edge marks*. If an edge e in an embedding E is contained in every subembedding of E homeomorphic to K_5 or $K_{3,3}$, then e is marked. The converse is usually, but not always, true: edges that are not contained in every Kuratowski subgraph may or may not be marked. Note that this concept differs from the red colouring used in proving the existence of parents, because there we chose a specific subgraph homeomorphic to K_5 or $K_{3,3}$ and coloured all of it. Edge marks, however, only need to be applied to edges that are in all subgraphs homeomorphic to K_5 or $K_{3,3}$. All target embeddings contain edges that would be coloured red, but some target embeddings have no edges that need to be marked.

Removing an edge with a reverse $\mathcal{C}_{0,1}$ move can only make the graph planar if in so doing we destroy every nonplanar subgraph in the embedding. So any time we would remove an edge to obtain the parent and that edge is not marked, we get the planarity test result ("not planar") for free. In that case the edge can remain unmarked.

If we attempt to remove a marked edge, we must still do the planarity test. But if we do the test and it returns "not planar", then we know that the edge did not really need to be marked; obviously there exists some nonplanar subgraph not destroyed by the removal of the edge. So in that case, we can unmark the edge. Since we are examining all possible children for each embedding, marked edges tend to be tested, and unmarked if appropriate, sooner rather than later. So in practice, the set of marked edges is usually close to minimal. The following theorem shows that we can easily maintain a set of marked edges.

Theorem 5.3.1 *If the edges of a parent P are marked such that an edge is marked if it is in every subgraph of P homeomorphic to K_5 or $K_{3,3}$, then we can obtain a marking for the child C also satisfying that condition by following these rules:*

- *Edges marked in P are marked in C.*
- *New edges introduced by $\mathcal{C}_{0,1}$ moves are not marked.*
- *New edges introduced by $\mathcal{D}_{2,4}$ moves are not marked.*
- *New edges introduced by $\mathcal{T}_{1,1}$ moves are marked.*

Furthermore, if we ever do a planarity test on a graph $G - e$ consisting of a graph G minus an edge e, and $G - e$ is found to be nonplanar, then we can unmark e in G.

Proof. When we obtain C from P by a $\mathcal{C}_{0,1}$ move, obviously every subgraph of P is a subgraph of C also. So any edge that is in every subgraph of C homeomorphic to K_5 or $K_{3,3}$ must also be in every subgraph of P homeomorphic to K_5 or $K_{3,3}$. The set of edges that must be marked in C is a subset of the set of edges that must be marked in P, so if we make the marked edges of C equal to the marked edge of P, we obtain a legal marking.

When we obtain C from P by a $\mathcal{D}_{2,4}$ move, we do not mark any of the new edges. Call the endpoints of the edge being removed u and v. If the edge (u, v) was unmarked in P then obviously there is some subgraph homeomorphic to K_5 or $K_{3,3}$ in P that did not include that edge, and that subgraph is retained in C. Even if the edge being removed was marked, removal of any one of the new edges maintains a path between u and v, and so does not make C planar by destroying all subgraphs homeomorphic to K_5 or $K_{3,3}$.

We mark all edges added by $\mathcal{T}_{1,1}$ moves in order to err on the side of caution, because it is possible that an edge added by such a move could be in every subgraph homeomorphic to K_5 or $K_{3,3}$, even when no other edges need to be marked. For instance, supposed we take an embedding of $K_{3,3}$, subdivide all nine edges as with $\mathcal{S}_{1,1}$ moves, and then perform a $\mathcal{D}_{2,4}$ move on every edge of the result; so we have an embedding of $K_{3,3}$ with every edge replaced by two diamonds in a row.

Any one edge from the resulting embedding could be removed without rendering the graph planar. Suppose we split the degree four vertex joining two diamonds in this construction. We could either preserve both diamonds or destroy them both, depending on how we make the split. If we preserve the diamonds, the new edge is on every subgraph homeomorphic to K_5 or $K_{3,3}$ and so needs to be marked. Rather than attempting to make some elaborate test for which new edges from $\mathcal{T}_{1,1}$ moves need to be marked, we simply mark them all. Any edges marked unnecessarily by that rule will soon be unmarked as a result of a planarity test anyway.

Finally, we can remove the mark from an edge e if removal of e leaves the graph nonplanar, because that is the definition of marking. Edges must be marked if their removal makes the graph planar, and may or may not be marked if their removal does not make the graph planar. $\square$

Chapter 6

Experimental results

Our algorithm is designed for practical implementation. This chapter begins with Section 6.1, which describes our implementation of an embedding generator based on this work. In Section 6.2 we describe some results obtained by running our generator, and give tables of the embeddings and graphs found. We also comment briefly on the number of embeddings per graph.

6.1 Implementation of the algorithm

We implemented several versions of an embedding generator during the project, as the theoretical work developed. The final version, used to calculate the results given here, contains approximately 5,100 lines of C language source code, plus some additional utilities written in C and Perl, the GNU getopt library function [21], and a makefile to manage the compilation process. This version is based on Theorem 4.2.1 and the algorithm of Section 2.5, to generate lists of diamond-free target embeddings by examining all target embeddings containing at most one diamond.

Development was conducted on the author's dual 433MHz Intel Celeron-based personal computer, under the GNU/Linux operating system. The computational

experiments were conducted there and on various computers running Solaris at the University of Victoria and Rochester Institute of Technology. All the CPU times listed here are for the Celeron unless otherwise specified, and are measured in user-space CPU time to reduce the effect of other processes running on the same computers.

The embedding generator includes some additional features, like the edge-marking technique of Section 5.3 to reduce the number of planarity tests performed, and the technique described by McKay [28, Section 8] for splitting the computation into parallel slices. We include an implementation of the planarity algorithm of Demoucron, Malgrange, and Pertuiset [15]. Although this algorithm does not offer the linear asymptotic time complexity of some other planarity algorithms, it performs well with the relatively small graphs our code processes.

The data structure we use for embeddings is a simplified version of that described by Boyer and Myrvold for their planarity algorithm [11, Section 4]. Each vertex has a circular doubly-linked list of records representing the neighbours around that vertex in clockwise order; the two records representing the endpoints of each edge are joined by pointers called *twin links*. We do not use the special feature of treating the two linked-list pointers equivalently, because we do not need to be able to reverse the order of a list in constant time.

As well as the linked-list representation of the embedding, we also maintain an adjacency matrix as a packed bit array. Adjacency matrices inherently require $O(n)$ or $O(n^2)$ time for some operations that could be done faster on other structures. But our implementation, although it can handle almost any number of vertices in theory with the appropriate compiled-in options, is limited to embeddings of up to about eleven or twelve vertices in practice simply by output size and computation time. Packed bit arrays of this size can be implemented so efficiently on current bit-parallel computers that the adjacency matrices are extremely fast and convenient, despite their asymptotic disadvantages.

One special feature of the present software allows running the generation pro-

cess in reverse: with an embedding provided as input, it prints out the parent of the input embedding, the parent of that embedding, and so on, until a seed embedding is reached or a consistency check fails. That procedure proved invaluable when debugging the parent-selection code, since a common failure mode was for the software to select as a parent an embedding (or an inconsistent data structure vaguely resembling an embedding) that would not ever be generated by the generation algorithm.

6.2 Diamond-free targets up to $n = 10$

We ran the embedding generator to make a list of all diamond-free target embeddings on up to nine vertices, and stored the results as compressed text files with one line, containing the canonical form, for each embedding. The resulting files store approximately 17 million embeddings of 75 thousand graphs in approximately 120 megabytes of disk space. This run was split into three equal slices and consumed approximately 9.4 hours of CPU time.

Since a similar set of files for the ten-vertex case would be too large to store conveniently, we split the computation into 100 slices and had the program output only the graph for each diamond-free target embedding. We then counted the number of embeddings for each isomorphism class of graphs. Generating these embeddings required approximately 13 days of CPU time. The resulting files, containing graphs in `nauty` canonical form [27] and a count of embeddings for each graph, consume approximately 19 megabytes compressed. There were approximately 3.9 million graphs with 462 million embeddings in this run. Storing the embeddings would be prohibitive: extrapolating the space consumption of the nine-vertex graphs gives an estimate of at least 3,200 megabytes to store all the ten-vertex embeddings of diamond-free target graphs, even in compressed form.

To provide a reference for debugging purposes, we also obtained an independent list of diamond-free target graphs with up to nine vertices, by using the `geng`

software by McKay [28] to generate all graphs on up to nine vertices with all vertices having minimum degree three and few enough edges to be toroidal. We passed those graphs through a simple filter to remove the ones containing diamonds, and used the torus embedding software of Neufeld and Myrvold [34] to find the graphs with genus one.

Generating the independent list of diamond-free target graphs required approximately four months of CPU time (compare to 9.4 hours to generate the same list with our generator program), but the result proved to be invaluable for debugging our generator. Most programming mistakes in our own software manifested either as duplicate embeddings in the output, or as target graphs that failed to appear in the output, so we tested our package by checking for duplicates, then using `nauty` [27] to find a list of graphs in our output up to isomorphism, and checking that list against the reference list.

The list from our current generator agrees with the reference list, and that adds to our confidence not only that our software is correct but that the packages used to make the reference list are also correct. Since our generator uses a completely different algorithm from the algorithms used by `geng` and the torus embedder, it seems highly unlikely that both lists would accidentally omit exactly the same graphs.

Our counts of diamond-free target embeddings and graphs with up to ten vertices are shown in Tables 6.1 and 6.2 respectively. We also found the maximum number of torus embeddings for any one diamond-free target graph with a fixed number of vertices n and edges m; these numbers are shown in Table 6.3.

In Table 6.4 we show the mean count of torus embeddings per graph for each value of n, obtained by dividing the number of embeddings by the number of graphs. Note that the number of embeddings per graph increases with more vertices up to eight, but then decreases a little for nine-vertex graphs and decreases considerably more for ten-vertex graphs. However, the maximum number of embeddings for a single target graph at each value of n, shown in the bottom line of Table 6.3,

	$n=5$	6	7	8	9	10
$m=9$		2				
10	6	7				
11		45	14			
12		110	218	9		
13		113	1,287	364		
14		24	3,702	4,822	241	
15		4	4,990	28,851	8,106	74
16			3,184	88,564	96,129	7,012
17			911	150,724	556,190	170,443
18			168	144,888	1,814,463	1,771,171
19			21	79,845	3,574,097	9,951,881
20			2	26,194	4,408,741	33,834,400
21			1	5,613	3,475,526	74,055,939
22				790	1,784,251	108,160,019
23				69	611,193	107,805,321
24				7	142,144	74,469,982
25					21,760	36,069,910
26					2,000	12,326,126
27					112	2,938,905
28						467,434
29						44,739
30						2,109
total	6	305	14,498	530,740	16,494,953	462,075,465

Table 6.1: Counts of diamond-free target embeddings on the torus.

	$n=5$	6	7	8	9	10
$m=9$		1				
10	1	1				
11		2	1			
12		2	6	2		
13		2	14	13		
14		1	23	59	11	
15		1	23	180	132	9
16			17	339	784	171
17			9	441	2,757	2,003
18			5	415	6,473	12,726
19			2	307	10,757	51,060
20			1	187	13,548	142,358
21			1	103	13,565	295,863
22				51	11,271	482,277
23				22	7,920	640,518
24				7	4,639	706,707
25					2,113	643,664
26					628	466,213
27					101	250,369
28						90,649
29						19,190
30						1,866
total	1	10	102	2,126	74,699	3,805,643

Table 6.2: Counts of diamond-free target graphs on the torus.

	$n=5$	6	7	8	9	10
$m=9$		2				
10	6	7				
11		25	14			
12		88	68	5		
13		66	239	45		
14		24	372	199	44	
15		4	866	570	158	20
16			491	1,084	498	100
17			281	2,232	1,056	308
18			64	1,818	2,216	1,010
19			11	2,112	3,384	2,110
20			2	828	4,196	3,860
21			1	375	5,164	5,808
22				66	2,918	9,144
23				8	1,380	9,748
24				1	499	7,476
25					162	3,828
26					24	1,521
27					2	718
28						166
29						48
30						6
all m	6	88	866	2,232	5,164	9,748

Table 6.3: Maximum numbers of torus embeddings for diamond-free target graphs.

$n = 5$	6	7	8	9	10
6.00	30.50	142.14	249.64	220.82	121.42

Table 6.4: Mean number of torus embeddings per diamond-free target graph.

continues to increase with additional vertices, at least for the numbers of vertices we examined. An embedding of the single diamond-free target graph we examined with most torus embeddings, 9,748 of them, is shown in Figure 6.1.

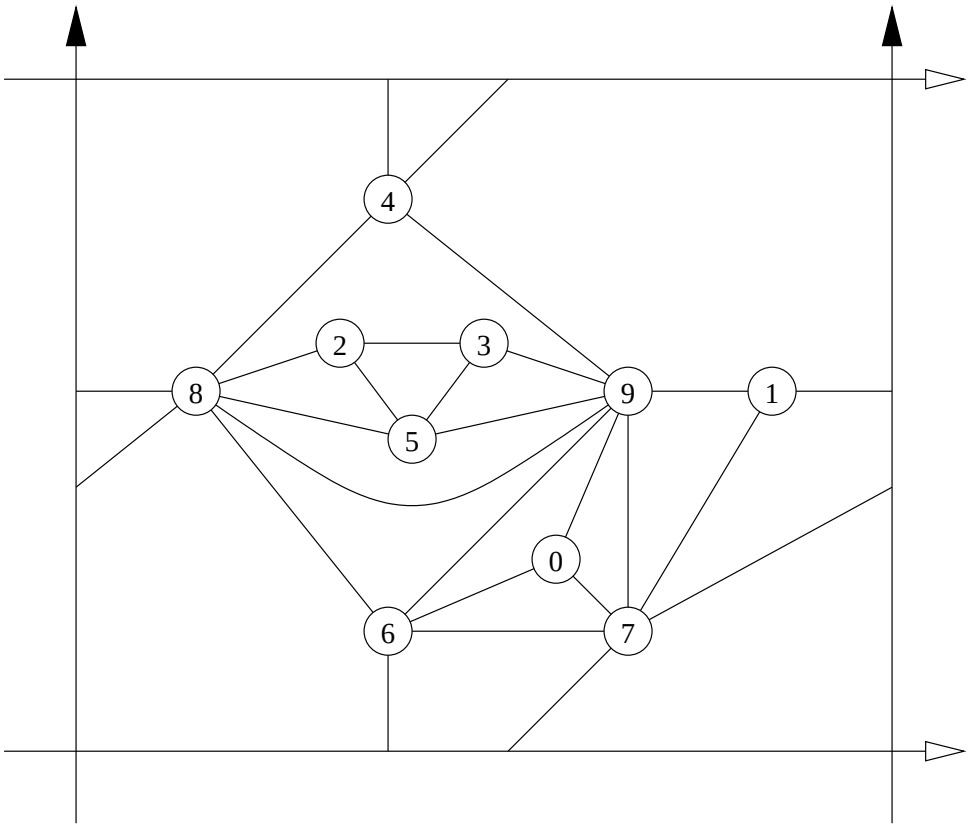


Figure 6.1: One of the 9,748 torus embeddings of the unique ten-vertex diamond-free target graph with maximum number of torus embeddings.

Chapter 7

Applications and future work

Although the lists of diamond-free target embeddings generated by our software and described in the previous chapter may have some interest in themselves, the algorithm is intended to be useful in some specific applications. This chapter describes some of those applications. We begin by describing a fast lookup-based toroidality test in Section 7.1. In Section 7.2 we apply that test to the search for topological obstructions to embeddability on the torus. Future work with this algorithm could focus on searches for additional obstructions. Another possible direction for future work would be the application of these techniques to other surfaces, described in Section 7.3. We end the chapter with a summary of our conclusions, in Section 7.4.

7.1 A lookup-based toroidality tester

We can use the output of the embedding generator to build a database of diamond-free target graphs, and then use that database as the basis for a fast toroidality test. Given a graph G with n vertices and the database of diamond-free target graphs with up to n vertices, we can eliminate any vertices of degree less than three from G, find its biconnected components, and check them for planarity. If more than one biconnected component is nonplanar then G must have genus greater than

one; if all are planar then G is planar; otherwise, we look up the one nonplanar biconnected component in the database. The following pseudocode describes the algorithm:

```
FastGenus(G) :
genus ← 0
while G is not empty
    H ← some biconnected component of G
    if H is nonplanar
        replace diamonds with edges by reverse D_{2,4} moves, and eliminate
            vertices of degree less than three with reverse S_{1,1} moves
        if H is in the list of diamond-free target graphs
            genus ← genus + 1
        else
            return "greater than one"
        end if
        if genus > 1
            return "greater than one"
        end if
    end if
    G ← G − H
end while
return genus
```

Our implementation of FastGenus consists of a filter that writes out, for each input graph G, either K_4 if G is planar; a constant genus two graph if G has genus at least two; or a graph isomorphic to the nonplanar biconnected component of G except for diamonds and vertices of degree less than three, if G has exactly one nonplanar biconnected component. So the output of the filter is a graph that falls into the same category (genus zero, genus one, or genus at least two) as the input;

if the genus is zero then the output is K_4, and if the genus is one then the output is some labelling, possibly not canonical, of a diamond-free target graph.

We pass the output of our filter through the `nauty` [27] canonical labelling utility; then we look up the result in a table to find the category for the graph. To handle input graphs up to ten vertices, the table has about 3.9 million entries and consumes about 39 megabytes of disk space (ten bytes per entry). The table contains K_4, and the diamond-free target graphs from the generator program; if the output of the filter is not in the table, then we know the input graph G must have had genus at least two. Running the test on large batches of graphs, using our filter, `nauty`, and the sorting and lookup utilities provided by the operating system, we can categorize about ten thousand graphs per CPU second.

7.2 Searching for torus obstructions

The generalized Kuratowski theorem states that for any surface S, there is a finite list of graphs called *topological obstructions* such that a graph G is embeddable on the surface if and only if it does not contain a subgraph homeomorphic to a graph on the list. The theorem can also be stated in terms of minors: G is embeddable if and only if it does not contain as a minor one of the graphs on a finite list of *minor-order obstructions*. The result was proved for non-orientable surfaces by Archdeacon and Huneke [4] and for orientable surfaces by Bodendiek and Wagner [9]. Robertson and Seymour proved a stronger conjecture that includes the general Kuratowski theorem [37]. These citations are from a survey by Archdeacon [3].

For the plane, the sets of topological and minor order obstructions are both equal to $\{K_5, K_{3,3}\}$. For the projective plane, there are 103 topological obstructions corresponding to 35 forbidden minors [1, 2, 20]. The torus embedding code of Neufeld and Myrvold [34, 33] led to a complete list of torus obstructions with up to ten vertices, and a partial list of larger obstructions.

A few obstructions for various surfaces can also be generated by simple rules;

for instance, suppose we take $k+1$ copies of K_5, choose one vertex from each, and identify all the chosen vertices. The result, containing $4k+5$ vertices, must be a topological and minor-order obstruction for the k-handled torus because it contains $k+1$ biconnected components each with genus one, and removing or contracting any edge reduces the genus of that 3-connected component to zero, reducing the genus of the entire graph by a theorem of Battle, Harary, Kodama, and Youngs [8]. We know of no known embeddability obstructions apart from these kinds of constructions and the results mentioned above for the plane, projective plane, and torus.

A topological obstruction G for the torus has the property that G is not toroidal but removing any one edge from G gives a toroidal graph. This property leads naturally to a technique for finding obstructions: if we take a list of all toroidal graphs, add one edge to each of them in all possible ways, and remove any toroidal graphs from the resulting list, all topological obstructions for the torus will be included in the resulting list. All topological obstructions are diamond-free as a consequence of Theorem 3.1.1, and if we confine our attention to biconnected obstructions, we can find them all on the list derived from our software's list of diamond-free target graphs.

We took the list of about 3.9 million diamond-free target graphs generated by our software for up to ten vertices, subdivided between zero and two edges in all possible ways to get graphs with no more than ten vertices, and eliminated duplicate graphs. Then we added an edge to each graph in all possible ways, eliminated duplicates, and removed all graphs that were on the list of diamond-free target graphs. That produced a list of 1,028,118 graphs, including all topological obstructions for the torus with up to ten vertices. The step of subdividing edges was necessary in order to be able to create obstructions where every edge is incident to a degree three vertex; such a graph clearly cannot be created by adding just an edge to a graph with no vertices of degree less than three. The only biconnected obstruction we found with every edge incident to a degree three vertex is the graph $K_{7,3}$, shown in

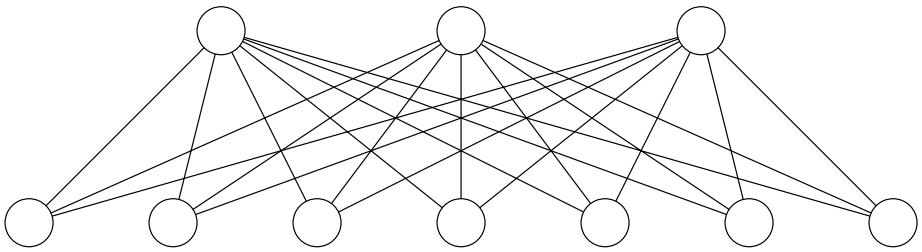


Figure 7.1: The graph $K_{7,3}$, a topological obstruction to torus embeddability.

Figure 7.1.

For every graph G on this list of candidate obstructions, we removed one edge in all possible ways and made another list of all those graphs $G - e$. We applied the fast lookup-based torus test described in the previous section to the list of $G - e$ graphs, and generated a list of all graphs G that were not toroidal but where removing an edge e would always make $G - e$ toroidal; in other words, a list of topological obstructions. A summary of that list is shown in Table 7.1. The CPU time consumption for this obstruction search was difficult to measure because the processing was divided between several different pieces of sorting, merging, and lookup software. We estimate the consumption at six hours, about a third spent running the system sort utility, and excluding the 13 days required to compile the database as discussed in Section 6.2.

We checked the 707 obstructions on our list with `nauty` to make sure they were distinct, and with the simplification utility from our lookup-based torus test to make sure they were biconnected. We also verified that each of our believed topological obstructions really was a topological obstruction, using the torus tester of Neufeld and Myrvold [34, 33]. Our counts of obstructions agree with theirs except in the case of ten vertices and 26 edges, where we count one more obstruction. After obtaining their list of 656 ten-vertex topological obstructions [31], we found that the one missing obstruction was the one shown in Figure 7.2. We were unable to determine why their search missed this obstruction.

	$n = 8$	9	10
$m = 19$	0	2	14
20	0	4	8
21	0	2	34
22	1	9	40
23	0	17	190
24	1	6	170
25	1	2	102
26	0	5	76
27	0	0	21
28	0	0	1
29	0	0	0
30	0	0	1
total	3	47	657

Table 7.1: Biconnected topological obstructions for the torus with up to ten vertices.

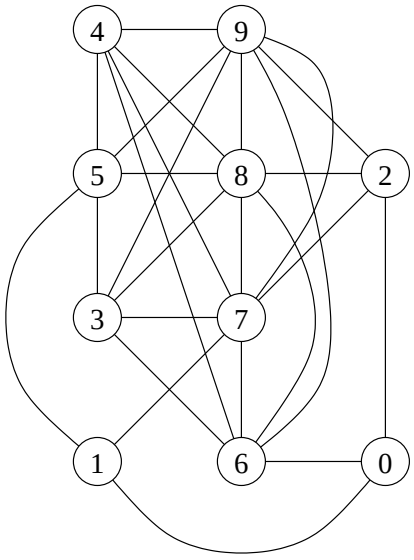


Figure 7.2: The obstruction not found by Neufeld and Myrvold [34, 33].

7.3 Other surfaces

Much of this work could also be applied to other surfaces besides the torus. Indeed, much of the programming in our project was originally done with an extension to arbitrary surfaces in mind. Generating embeddings of projective planar graphs embedded on the projective plane would require us to extend the concept of a combinatorial embedding to express embeddings on nonorientable surfaces, but that is not difficult. A technique involving positive and negative signs placed on the edges of the embedding is used in the projective planarity algorithms of Mohar [29] and Myrvold and Roth [32]. While developing the toroidal graph generator, we implemented but did not test or use some subroutines to handle embeddings extended this way.

For the projective plane, the edge signs are the only enhancement obviously necessary. The proofs of Chapters 3 and 4 should be easy to extend to the projective plane. For surfaces of higher genus, both orientable and non-orientable, the problem may be somewhat more complicated.

First, there is the question of what embeddings to use as seeds. The same set of eight seeds we used for the torus would seem to be a good choice for the projective plane also, but it is not clear what seeds to use, for example, for the two-handled torus. Should we use the set of topological obstructions for the torus? If we did, it would raise a problem, because some torus obstructions are not connected, let alone biconnected; we would have to either revise our definition of target graphs to allow the use of those as seeds, or somehow prove them unnecessary.

The second problem involves verifying the genus of target graphs. Our algorithm for the torus requires that we test, when examining possible parents, that each graph really is genus one; the construction provides that the genus is no more than one, and we use a planarity test to verify that the genus is no less than one. We could use the planarity test when generating graphs on the projective plane also. But when generating graphs on a surface of higher genus we would need a higher-genus

testing algorithm; for instance, a torus tester to generate graphs on the two-handled torus.

The torus tester of Neufeld and Myrvold [34] seems too slow to be useful in this context, where millions of graphs must be processed; but perhaps it could be sped up by some kind of memoization technique, because the same graphs will be tested many times. The lookup-based toroidality tester we constructed in Section 7.1 could also be useful. Note that the implemented projective planarity tester of Myrvold and Roth [32], and the known list of obstructions for the projective plane [1, 2, 20], suggest a relatively easy application to the Klein bottle.

We could consider generating embeddings on the plane. Indeed, some early versions of our software (before we completed the theoretical work) were designed to also generate planar embeddings. On the plane, of course, there is no concern about testing genus; we can start with planar embeddings as seeds and then the construction moves can keep them planar. However, something would have to be done about the wheels, which are an infinite set of biconnected planar graphs with no degree three vertices, each of which is not amenable to any reverse $\mathcal{C}_{0,1}$, $\mathcal{D}_{2,4}$, or $\mathcal{T}_{1,1}$ move to leave a biconnected planar graph with no degree three vertices.

Also, some special accommodation might possibly have to be made for K_4, which is a graph with the unique feature that every one of its edges is a diamond edge. The proof of Theorem 3.1.8, for instance, depends on the fact that a target graph cannot be a wheel. The only move that can be made on an embedding of K_4 is a $\mathcal{D}_{2,4}$ move that destroys five diamonds and creates one, which could cause problems for the line of reasoning developed in Section 4.2.

7.4 Conclusions

We have described an algorithm to generate one representative from every isomorphism class of diamond-free target embeddings up to a chosen number of vertices or edges, and proved that algorithm correct. We have described some additional

issues relating to implementation, and our own C language implementation of the algorithm. Some experimental results from our implementation have been presented, including the determination of all biconnected topological obstructions to torus embeddability containing ten or fewer vertices. Finally, we have proposed some additional applications for the algorithm.

Bibliography

[1] D. Archdeacon. *A Kuratowski theorem for the projective plane.* Thesis, Ohio State University, 1980.

[2] D. Archdeacon. A Kuratowski theorem for the projective plane. *Journal of Graph Theory*, 5:243–246, 1981.

[3] D. Archdeacon. Topological graph theory: A survey. *Congressus Numeratum*, 115:5–54, 1996.

[4] D. Archdeacon and J. P. Huneke. A Kuratowski theorem for nonorientable surfaces. *Journal of Combinatorial Theory, Series B*, 46:173–231, 1989.

[5] A. Argyle. Toroidal embeddings of $K_{3,3}$ and K_5. CSC 499 Technical Project, University of Victoria, 1999.

[6] D. Barnette. Generating the triangulations of the projective plane. *Journal of Combinatorial Theory, Series B*, 33:222–230, 1982.

[7] D. Barnette. Generating the 4-connected and strongly connected triangulations on the torus and projective plane. *Discrete Mathematics*, 85:1–16, 1990.

[8] J. Battle, F. Harary, Y. Kodama, and J. Youngs. Additivity of the genus of a graph. *Bulletins of the American Mathematical Society*, 68:565–568, 1962.

[9] R. Bodendiek and K. Wagner. Solution to König's graph embedding problem. *Math. Nachr.*, 140:251–272, 1989.

[10] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[11] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, Maryland, January 17–19, 1999)*, pages 140–146, 1999.

[12] G. Brinkmann and B. McKay. Fast generation of some classes of planar graphs. preprint.

[13] J. Cai. Counting embeddings of planar graphs using DFS trees. *SIAM Journal on Discrete Mathematics*, 6(3):335–352, 1993.

[14] N. Chiba, T. Nishizeki, A. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30:54–76, 1985.

[15] G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires. *Revue Française Recherche Opérationnelle*, 8:33–47, 1964.

[16] J. R. Fiedler, J. P. Huneke, R. B. Richter, and N. Robertson. Computing the orientable genus of projective graphs. *Journal of Graph Theory*, 20(3):297–308, 1995.

[17] I. S. Filotti. An algorithm for embedding cubic graphs in the torus. *Journal of Computer and System Sciences*, 2:255–276, 1980.

[18] M. Fontet. A linear algorithm for testing isomorphism of planar graphs. In S. Michaelson and R. Milner, editors, *Third International Colloquium on Automata, Languages and Programming*, pages 411–424, University of Edinburgh, July 20–23 1976. Edinburgh University Press.

[19] O. Frink and P. Smith. Abstract 179. *Bulletins of the American Mathematical Society*, 36:214, 1930.

[20] H. Glover, J. Huneke, and C. Wang. 103 graphs that are irreducible for the projective plane. *Journal of Combinatorial Theory, Series B*, 27:332–370, 1979.

[21] GNU Project. *GNU `getopt`*. Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA, 1995. Computer software, from the Fetchmail 4.1.1 distribution by Eric S. Raymond.

[22] M. Henle. *A combinatorial introduction to topology*. Dover Publications, Inc., New York, 1994.

[23] J. Hopcraft and J. Wong. Linear time algorithm for isomorphism of planar graphs. In *6th ACM SIGACT*. Association for Computing Machinery, New York, 1974.

[24] M. Juvan, J. Marinček, and B. Mohar. Embedding graphs in the torus in linear time. In *Integer Programming and Combinatorial Optimization*, volume 920 of *Lecture Notes in Computer Science*, pages 360–363. Springer, Berlin, 1995.

[25] L. C. Kinsey. *Topology of Surfaces*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1993.

[26] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[27] B. D. McKay. `nauty` user's guide (version 1.5). Technical Report TR-CS-90-02, Department of Computer Science, Australian National University, 1990.

[28] B. D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, Feb. 1998.

[29] B. Mohar. Projective planarity in linear time. *Journal of Algorithms*, 15:482–502, 1993.

[30] B. Mohar. A linear time algorithm for embedding graphs in an arbitrary surface. *SIAM Journal of Discrete Mathematics*, 12(1):6–26, 1999.

[31] W. Myrvold. Personal communication.

[32] W. Myrvold and J. Roth. Simpler projective plane embedding. Submitted to *Discrete Mathematics*, June 2000.

[33] E. Neufeld. Practical toroidality testing. Master's thesis, Department of Computer Science, University of Victoria, 1993.

[34] E. Neufeld and W. Myrvold. Practical toroidality testing. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, Louisiana, January 5–7, 1997)*, pages 574–580, 1997.

[35] B. Peruničić and Z. Durić. An efficient algorithm for embedding graphs in the projective plane. In *Proceedings of the Fifth Quadrennial International Conference on the Theory and Applications of Graphs with special emphasis on Algorithms and Computer Science Applications (Kalamazoo, Michigan, June 4–8, 1984)*, pages 637–650, 1985.

[36] R. Read and D. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[37] N. Robertson and P. Seymour. Graph minors VIII: A Kuratowski theorem for general surfaces. *Journal of Combinatorial Theory, Series B*, 48:255–288, 1990.

VITA

Surname: Skala Given Names: Matthew Adam

Place of Birth: Victoria, British Columbia, Canada

Educational Institutions Attended:

University of Victoria	1995 to 2001
Camosun College	1994 to 1995

Degrees Awarded:

B.Sc.	University of Victoria	1999

Honours and Awards:

NSERC Postgraduate Scholarship (PGS A)	2000 to 2001
University of Victoria Fellowship	1999 to 2000
President's Research Scholarship	2000
BC ASI Graduate Scholarship	1999

Publications and Presentations:

Skala, M., and Myrvold, W. (2001) Fast Generation of Graphs Embedded on the Torus. Presented at 32nd Southeastern International Conference on Combinatorics, Graph Theory, and Computing, Baton Rouge, Louisiana, February 26–March 2, 2001.

Goodenough, D.G., Charlebois, D., Bhogal, A.S., Dyk, A., and Skala, M. (1999) SEIDAM: A Flexible and Interoperable Metadata-Driven System for Intelligent Forest Monitoring. Proceedings of the International Geoscience and Remote Sensing Symposium 1999 (IGARSS'99), Hamburg, Germany, pp. 1338–1341.

Skala, M. (1998) A Limited-Diffusion Algorithm for Blind Substring Search. Proceedings of the 10th Annual Canadian Information Technology Security Symposium, 1–8 June 1998, Ottawa, Ontario, pp. 397–410.

Title of Thesis:

Generation of Graphs Embedded on the Torus

Author _____

Matthew Adam Skala

August 27, 2001