A Practical Torus Embedding Algorithm and Its Implementation

by

Jiahua Yu
B.Eng., University of Science and Technology of China, 2011

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

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Jiahua Yu 2014
SIMON FRASER UNIVERSITY
Summer 2014

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Jiahua Yu

Degree: Master of Science

Title of Thesis: A Practical Torus Embedding Algorithm and Its Implementation

Examinining Committee:

Dr. Ke Wang, Professor
Chair

Dr. Qianping Gu,
Professor, Computing Science,
Simon Fraser University
Senior Supervisor

Dr. Andrei Bulatov,
Professor, Computing Science,
Simon Fraser University
Supervisor

Dr. Jiangchuan Liu,
Associate Professor, Computing Science,
Simon Fraser University
Internal Examiner

Date Approved: June 13, 2014
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the non-exclusive, royalty-free right to include a digital copy of this thesis, project or extended essay[s] and associated supplemental files (“Work”) (title[s] below) in Summit, the Institutional Research Repository at SFU. SFU may also make copies of the Work for purposes of a scholarly or research nature; for users of the SFU Library; or in response to a request from another library, or educational institution, on SFU’s own behalf or for one of its users. Distribution may be in any form.

The author has further agreed that SFU may keep more than one copy of the Work for purposes of back-up and security; and that SFU may, without changing the content, translate, if technically possible, the Work to any medium or format for the purpose of preserving the Work and facilitating the exercise of SFU's rights under this licence.

It is understood that copying, publication, or public performance of the Work for commercial purposes shall not be allowed without the author’s written permission.

While granting the above uses to SFU, the author retains copyright ownership and moral rights in the Work, and may deal with the copyright in the Work in any way consistent with the terms of this licence, including the right to change the Work for subsequent purposes, including editing and publishing the Work in whole or in part, and licensing the content to other parties as the author may desire.

The author represents and warrants that he/she has the right to grant the rights contained in this licence and that the Work does not, to the best of the author’s knowledge, infringe upon anyone's copyright. The author has obtained written copyright permission, where required, for the use of any third-party copyrighted material contained in the Work. The author represents and warrants that the Work is his/her own original work and that he/she has not previously assigned or relinquished the rights conferred in this licence.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2013
Abstract

Embedding graphs on the torus is a problem with both theoretical and practical significance. It is required to embed a graph on the torus for solving many application problems in graphs. Such problems appear in disciplines including VLSI design and graph drawing. Although polynomial time algorithms for embedding graphs on the torus exist, they are complex and no working implementation exists. To develop a practical tool for embedding graphs on the torus, we propose a new algorithm with exponential running time. Compared with a previous well known exponential time algorithm, our algorithm has better practical performance. Furthermore, we show that our implementation covers most modules of a polynomial time algorithm and can serve as a good foundation for its implementation.

Keywords: Graph theory; Algorithm; Embedding; Torus; Exponential time
Acknowledgments

I would like to express my deepest gratitude to my senior supervisor Dr. Qian-ping Gu, without whom the completion of this thesis would not have been possible. Dr. Gu introduced me into the field of algorithmic graph theory and the world of research. His encouraging guidance, thoughtful ideas, meticulous inspections and easy-going character enabled me to explore the unknown and discover the possibilities. I would like to thank Dr. Andrei Bulatov for being my supervisor during my graduate study. I would also like to thank Dr. Jiangchuan Liu for being my examiner and Dr. Ke Wang for taking the time to chair my thesis defense.

Thanks to Dr. Bojan Mohar and Dr. Petr Skoda from Department of Mathematics, Simon Fraser University, for their meaningful discussions and sample of implementation, which brings numerous values to this work. Thanks to Dr. Guochuan Zhang from Zhejiang University, for his inputs into this research.

I am grateful to Mingzhe Zhu for the discussions on both my research and courses and for providing me with well-formatted datasets. Thanks to all my lab mates for their help in my graduate study and the helpful discussions in seminars. Thanks are also due to my friends, who accompanied me, supported me and laughed with me through the three years.

Last but not least, I would like to give the most special thanks to my family: my parents and my elder sister. They gave me unconditional support and continuous love throughout my life. Without them, neither my graduate study nor this thesis could be possible.
# Contents

Approval ii  
Partial Copyright License iii  
Abstract iv  
Acknowledgments v  
Contents vi  
List of Tables viii  
List of Figures ix  
List of Algorithms x  

## 1 Introduction 1  

## 2 Preliminaries 4  

- 2.1 Graph theory basics ........................................ 4  
- 2.2 Surfaces ...................................................... 8  
- 2.3 Graphs on surfaces .......................................... 11  
- 2.4 Combinatorial representation ............................... 13  
- 2.5 Projective plane embedding .................................. 15  

## 3 Review on Torus Embedding Algorithms 19  

- 3.1 Conversion to planar embedding problem .................. 19  
  - 3.1.1 Finding a proper cycle ................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Generating torus embedding</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Bridge based embedding algorithm</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Enumerative bridge embedding</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Recursive bridge embedding</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Recursive bridge assignment</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Recursive bridge attachment</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Path based embedding algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Polynomial time embedding algorithm</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>A New Torus Embedding Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Analysis of previous algorithms</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Scheme for the new algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Projective plane embedding tools</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Finding all embeddings of a graph</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Improvement on non-embeddable graphs</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>Component management</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Towards polynomial time algorithm implementation</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>Computational Results</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Implementation details</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiments environment setup</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Category 1 instances</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Random instances</td>
<td>56</td>
</tr>
<tr>
<td>5.5</td>
<td>Analysis</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Appendix A   Enumeration of Embeddings</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Appendix B   User Manual</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>B.1 Data format</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>B.2 The EMB library</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>B.3 The program</td>
<td>72</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Comparison among torus embedding algorithms . . . . . . . . . . . . 38
5.1 Results with C1 graphs as input . . . . . . . . . . . . . . . . . . . 54
5.2 Results with C2 graphs as input . . . . . . . . . . . . . . . . . . . 56
5.3 Results with C3 graphs as input . . . . . . . . . . . . . . . . . . . 58
A.1 Number of embeddings on the torus for some projective plane ob-
   structions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
List of Figures

2.1 $K_5$ and $K_{3,3}$ ......................................................... 5
2.2 An example of bridges ................................................. 6
2.3 Construction of surfaces .............................................. 9
2.4 Plane model and space model for some surfaces ................. 10
2.5 Equivalent rotation systems for one planar embedding ........ 13

3.1 Cycles around handle in torus embeddings ..................... 20

4.1 Efficiency-implementation trade off for torus embedding algorithms . 37
4.2 An example of framing graph selection ............................ 43
4.3 Calling graph for the new Algorithm ............................... 48
4.4 Calling graph for JM Algorithm .................................... 49

5.1 Running time graph on C1 inputs ................................. 55
5.2 Running time graph on C2 inputs .................................. 57
5.3 Running time graph on C3 inputs .................................. 59

B.1 Labelled $K_5$ and $K_{3,3}$ ............................................. 69
B.2 Embeddings of $K_5$ and $K_{3,3}$ on the torus ................... 69
B.3 Embeddings of $K_5$ and $K_{3,3}$ on the projective plane ........ 70
List of Algorithms

1. ComputeBridges(graph \( G \), subgraph \( K \)) ........................................ 7
2. DfsEdges(graph \( G \), subgraph \( K \), edge \( e \), bridge \( B \)) ............... 7
3. UnfoldFace(face \( F \)) ................................................................. 12
4. ComputeFaces(graph \( G \), rotation system \( \Pi(G) \), edge signatures \( \text{Sign} \)) 14
5. ProjectivePlaneEmbed(graph \( G \)) ................................................. 17
6. EdgeElimination(graph \( G \), condition \( \text{Condition}(G) \)) ............... 18
7. TorusEmbedNM(graph \( G \)) .......................................................... 21
8. TorusEmbedBridgeEnumerative(graph \( G \)) ........................................ 25
9. TorusEmbedBridgeRecursive(graph \( G \)) .......................................... 28
10. ExtendEmbedding(embedding \( \Pi(K) \), \( K \)-bridge set \( B \)) .............. 28
11. EmbedBridgeInFace(bridge \( B \), face \( F \)) ......................................... 29
12. ExtendEmbeddingRecursive(embedding \( \Pi \), \( K \)-bridge set \( B \)) ........... 31
13. AttachBridgeRecursive(bridge \( B \), unfolded Face \( F' \), embedding \( \Pi \), \( K \)-bridge set \( B \)) .................................................. 31
14. StartTorusEmbedWoodcock(graph \( G \)) ........................................... 32
15. TorusEmbedWoodcock(graph \( G \), subgraph \( H \), embedding \( \Pi(H) \)) ...... 33
16. TorusEmbedNew(graph \( G \)) .......................................................... 39
17. TorusEmbedAll(graph \( G \)) ............................................................. 41
18. TorusPreprocessNM(graph \( G \)) ..................................................... 44
19. TorusEmbedCoreGraph(graph \( G \)) ................................................ 45
20. StartTorusEmbedNew(graph \( G \)) .................................................... 46
21. GenerateRandomToroidalGraph(order \( n \)) .................................... 53
Chapter 1

Introduction

Graph theory has been widely used in various areas to model the relationship between elements. Classical applications include modeling roads on a map, modeling the relationship among users in a social network, modeling the conductive tracks among electronic components in a very-large-scale integration (VLSI) design, etc. Many practical problems can be modeled as optimization problems in graphs. For example, given a list of cities and their distances between each other, a truck delivering parcels tries to find the shortest possible route that visit each city exactly once and return to its origin city. Among the induced problems many are \textit{NP-hard} such as travelling salesman, vertex cover and hamiltonian path. Based on the assumption \( P \neq NP \), these problems do not have polynomial time algorithms in general. However, some efficient algorithms may exist when extra structural information are contained in the inputs. One important property of a graph \( G \) is whether \( G \) is embeddable on a surface.

A graph \( G \) consists of a set of vertices \( V \) and a set of edges \( E \). A drawing \( U(G) \) of a graph \( G \) on a surface is to draw each vertex of \( G \) as a point on the surface and each edge between two vertices as a segment connecting the two corresponding points of vertices on the surface. A graph is embeddable on a surface if the graph has a drawing where no two edges cross except at their ends on that surface. Many graphs in reality problems are embeddable on some surface. For example, roads on a map is a graph embedded on the plane, with the intersections as vertices. A graph modeling provinces/states and their adjacency is also embeddable on the plane. A graph induced by the circuits in VLSI can be embedded on some simple surface
CHAPTER 1. INTRODUCTION

given that minimum number of crosses are desired. Research on developing efficient algorithms has been conducted for graphs embedded on surfaces [13, 36]. The embedding of a graph on surfaces is also important in the field of graph drawing. Modeling and drawing graphs has been widely applied to visualize the relationships and structural properties among data in many disciplines. Edge crossings are confusing in such circumstances and it is thus desirable to draw graphs with none or few of them. Graph drawing tools [10, 14] create layouts based on graph embeddings.

Embedding graphs on the plane has been well studied and linear time algorithms are implemented [12, 19, 8] and widely applied. A linear time algorithm for embedding graphs on the projective plane (See Section 2.2 for definition) has also been proposed [26] and a simplified version with $O(n^3)$ running time has been implemented [31]. For embedding graphs on a surface with bounded genus (See Section 2.2 for definition), a linear time algorithm has been proposed [27, 28] and later simplified [23]. However, neither of the algorithms for bounded-genus surfaces is simple enough to be implemented and is mostly of theoretical interest. The problem of whether a graph is embeddable on a surface of genus $k$ or not is NP-complete when $k$ is part of the input [37, 38].

Embedding graphs on the torus (See Section 2.2 for definition) is a more prevailing problem where many researches are in progress and an efficient implementation is promising. A linear time algorithm was proposed in 1994 [21] and a simplified version with $O(n^3)$ running time was introduced in 1998 [22]. Although an implementation of the $O(n^3)$ algorithm was announced [1], it does not seem working [30, 33]. To the author’s best knowledge, a working implementation of polynomial time algorithm for embedding graphs on the torus is not known (See Section 3.4 for details). Progress has also been made in exponential time algorithms with practical running time and two implementations exist [32, 40].

In this thesis, we propose a faster exponential time algorithm for embedding graphs on the torus. We implemented this new algorithm and tested it against a previous well-known algorithm. Computational results show that the new algorithm has better practical performance both in average and worst case running time. The implementation is also a part of a larger project where we attempt to develop a
practical embedding tool based on the $O(n^3)$ algorithm. We show that this implementation covers many significant modules required in the $O(n^3)$ algorithm.

The rest of this thesis is organized as follows. In Chapter 2, we give the preliminary. A comprehensive review of existing algorithms, including their key implementation details, is given in Chapter 3. In Chapter 4, we conduct an analysis of existing algorithms and propose our new algorithm, including details in implementation and data preparation. We also include a couple of independent pre-processing steps. An implementation scheme for the $O(n^3)$ algorithm is then provided and we show the remaining challenges. Computational results are shown in Chapter 5 by comparing performance between the new algorithm and a previous well known algorithm. The final chapter concludes this thesis and presents some works that could be done in future.
Chapter 2

Preliminaries

2.1 Graph theory basics

Readers can refer to an introductory text book (e.g., *Introduction to Graph Theory* by West [39] or *Applied and Algorithmic Graph Theory* by Chartrand and Oellermann [9]) for basic concepts and definitions in graph theory. Unless otherwise specified, all graphs discussed in this thesis are undirected.

A graph \( G = (V, E) \) consists of two sets \( V \) and \( E \). \( V \) is a finite non-empty set of *vertices* and is the vertex set for \( G \). \( E \) is a finite set of *edges* and is the edge set for \( G \). Each edge \( e \) in \( E \) is a subset of \( V(G) \) with at most two elements. Each element is called an *end* of \( e \). Given an edge set \( X \subseteq E(G) \), \( V(X) \) represents the union of all ends of edges in \( X \), namely \( V(X) = \bigcup_{e \in X} e \). The *order* of a graph \( G \) is \(|V(G)|\), usually denoted by \( n \). The number of edges \(|E(G)|\) is usually denoted by \( m \).

An edge with only one end is called a *self-loop*. If two or more edges contain the same two vertices, they are called *multiple edges*. A graph is *simple* if it contains neither self-loop nor multiple edges. Unless otherwise specified, all graphs discussed in this thesis are simple. In this case, an edge \( e \) connecting \( u \) and \( v \) can also be denoted by \( \{u, v\} \) or \( uv \).

In a graph \( G \), the two ends of an edge \( e \) are *incident* to \( e \). Two vertices incident to a same edge are *adjacent* to each other and they are *neighbors*. The *degree* of a vertex \( v \) is the number of edges incident to \( v \) and denoted by \( \deg(v) \). A vertex with degree 0 is an *isolated vertex*.

A graph is *complete* if all vertices are pairwise adjacent. A complete graph with
n vertices is denoted by $K_n$. A graph $G$ is bipartite if there is a non-empty set $S \subset V(G)$ such that (s.t.) each edge is incident to a vertex in $S$ and a vertex in $V(G) - S$. A complete bipartite graph is a bipartite graph where each vertex in $S$ is adjacent to all vertices in $V(G) - S$. It is denoted by $K_{r,s}$ where $s = |S|$ and $r = |V(G) - S|$. For example, $K_5$ and $K_{3,3}$ are shown in Figure 2.1.

![Figure 2.1: $K_5$ and $K_{3,3}$](image)

A walk in a graph is a sequence of vertices and edges $v_0, e_1, v_1, \cdots, e_k, v_k$, where each edge $e_i = \{v_{i-1}, v_i\}$. This walk connects $v_0$ and $v_k$. The sequence can be simplified as $v_0, v_1, \cdots, v_k$ since each edge is uniquely defined by its two ends. The length of the walk is the number of edges in this walk. A walk is closed if $v_0 = v_k$. A path is a walk with no repeated vertex. A cycle is a closed walk with no repeated vertex except for the first and the last vertices.

Given two graphs $G$ and $H$, $H$ is a subgraph of $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Deleting an edge $e$ from $G$ results in a subgraph $G - e$ with vertex set $V(G)$ and edge set $E(G) - \{e\}$. Deleting a vertex $v$ from $G$ results in a subgraph $G - v$ with vertex set $V(G) - \{v\}$ and edge set $E(G) - \text{inc}(v)$ where $\text{inc}(v)$ is the set of edges incident to $v$. More generally, given an edge set $X$, $G - X$ is obtained by deleting all edges in $X$ from $G$. Similarly, given a vertex set $S$, $G - S$ is obtained by deleting all vertices in $S$ from $G$. Deleting a subgraph $H$ from $G$ is equivalent to deleting all vertices of $H$ ($G - V(H)$), and denoted by $G - H$.

Given an edge $e = \{u, v\}$ in $G$, subdividing $e$ means replacing $e$ with a new vertex $w$ and two edges $\{u, w\}$ and $\{w, v\}$. A graph $H$ is a subdivision of $G$ if it can be obtained from $G$ with a series of edge subdivisions. Two graphs $G$ and $G'$ are homeomorphic if they have a common subdivision $H$. 
A graph $G$ is connected if there is a path connecting any pair of vertices, otherwise it is disconnected. A maximal connected subgraph of $G$ is called a connected component of $G$. A vertex cut of $G$ is a set $S \subset V(G)$ s.t. $G - S$ becomes disconnected. A vertex cut with $k$ vertices is also called a $k$-vertex cut. $G$ is $k$-connected if it has no $(k - 1)$-vertex cut. We also call 2-connectivity biconnectivity and 3-connectivity triconnectivity. A biconnected component of $G$ is a maximal biconnected subgraph of $G$ and sometimes referred to as a block.

![Diagram of a graph and its subgraphs](image)

(a) A graph $G$ and a highlighted subgraph $K$

(b) $K$-bridges in $G$

Figure 2.2: An example of bridges

Let $K$ be a subgraph of a connected graph $G$. A $K$-bridge [29] in $G$ is a subgraph $B$ of $G$ that falls into one of the following two types:

1. An edge $e = \{u, v\} \in E(G)$ s.t. $u, v \in V(K)$ but $e \notin E(K)$

2. A connected component $C$ of $G - K$, along with edge set $X = \{\{u, v\}|u \in C, v \in K\}$
$V(C), v \in V(K)$} and vertex set $S = V(X)$

An example for bridges can be seen in Figure 2.2. In the figure, $B_2$, $B_3$, and $B_4$ are category 1 bridges. The remaining bridges are in category 2 and each of their connected component $C$ is highlighted. Given $K$, the set of $K$-bridges $B$ can be calculated with a modified version of DFS on the edges of $G$, as shown in Algorithm 1 and Algorithm 2.

**Algorithm 1** ComputeBridges(graph $G$, subgraph $K$)

1: Let $B$ be the set of all bridges
2: Mark all edges of $G$ as unvisited
3: Mark all edges of $K$ as visited
4: for each edge $e \in E(G)$ do
5:     Let $B$ be an empty bridge
6:     DfsEdges($G, K, e, B$)(Algorithm 2)
7: if $B$ is not empty then
8:     Add $B$ to $B$
9: end if
10: end for
11: return $B$

**Algorithm 2** DfsEdges(graph $G, subgraph K$, edge $e$, bridge $B$)

1: if $e$ is visited then
2:     return
3: end if
4: Mark $e$ as visited
5: Add $e$ to $B$
6: Let $u$ and $v$ be the ends of $e$
7: if $u \notin V(K)$ then
8:     for each edge $f = \{u, w\}, w \neq v$ do
9:         DfsEdges($G, K, f, B$)
10:     end for
11: end if
12: if $v \notin V(K)$ then
13:     for each edge $f = \{w, v\}, w \neq u$ do
14:         DfsEdges($G, K, f, B$)
15:     end for
16: end if

The vertex set $V(B) \cap V(K)$ is $B$’s attachment vertex set, denoted by $Att(B)$. 
Each vertex in this set is called an \textit{attachment vertex} of $B$. The \textit{attachment edge set} of $B$ is the set of all edges with at least one end in $\text{Att}(B)$, denoted by $\text{AttEdge}(B)$. An \textit{attachment entry} of $B$ is a pair $(e,v)$ where $e \in \text{AttEdge}(B), v \in \text{Att}(B), v \in e$. The set of all attachment entries of $B$ is denoted with $\text{AttEntry}(B)$. Let $B_0$ be a bridge of category 1 with $\text{Att}(B_0) = \{u,v\}$, then $\text{AttEdge}(B_0) = \{\{u,v\}\}$ and $\text{AttEntry}(B_0) = \{\{\{u,v\},u\},\{\{u,v\},v\}\}$. A path $p$ in $B$ is \textit{bisecting} if its first and last vertices are attachment vertices of $B$ and the others are not.

A vertex of $K$ is a \textit{branch vertex} if it has degree different from 2. A \textit{branch} in $K$ is a path connecting two branch vertices without other branch vertices inside. A $K$-bridge $B$ is \textit{local} if $\text{Att}(B) \subseteq V(P)$ where $P$ is some branch of $K$. In Figure 2.2, $B_7$ is local and others are not.

\section*{2.2 Surfaces}

Concepts of surfaces can be found in Henle’s book \textit{A Combinatorial Introduction to Topology} [18] and Mohar and Thomassen’s book \textit{Graphs on Surfaces} [29]. In topology, a \textit{surface} is a topological space where each point has a neighborhood topologically equivalent to an open disk. The sphere ($\mathbb{R}^3$) is a commonly seen surface.

A surface is \textit{orientable} if a consistent positive sense of rotation (e.g. clockwise) can be made around all points, otherwise it is non-orientable. The sphere is an orientable surface and other orientable surfaces can be obtained by adding \textit{handles} to the sphere. To add a handle to a surface, we remove two disjoint open disks from the surface, resulting in two boundaries $C_1$ and $C_2$ on the surface. Then we identify them with the two ends of a cylinder (Figure 2.3a and Figure 2.3b). The cylinder is thus called a handle. An orientable surface obtained by adding $k$ handles to the sphere is called a \textit{surface with genus $k$}, denoted by $S_k$. $S_0$ is the sphere and $S_1$ is the torus.

Non-orientable surfaces can be obtained by adding \textit{crosscaps} to the sphere. To add a crosscap to a surface, we remove one open disk from the surface and identify its boundary with the boundary of a Mobius strip (Figure 2.3c and Figure 2.3d). A non-orientable surface obtained by adding $k$ crosscaps to the sphere is called a \textit{surface with crosscap number $k$}, denoted by $N_k$. $N_1$ is the projective plane and $N_2$...
CHAPTER 2. PRELIMINARIES

(a) A cylinder and its construction  
(b) Adding a handle to the sphere

(c) A mobius strip and its construction  
(d) Adding a crosscap to the sphere

Figure 2.3: Construction of surfaces

is the Klein bottle.

One way to represent the surfaces on the plane is by using a plane model. The plane model of a sphere is a plane itself. Plane models for other mentioned surfaces are shown in Figure 2.4. In the plane models, two edges with same identifier are identical and their directions are indicated by arrows. From the plane model of a surface, its space model can be obtained by identifying the corresponding boundary edges. Edges are identified by identifying corresponding pairs of points following the directions of the edges. For example, in the plane model of the sphere, two edges with same directions are identified to form its space model. The corresponding pairs of points 1, 2, and 3 are identified in the order indicated by the arrows. On the contrary, the projective plane is obtained by identifying two edges with opposite direction. The torus or the Klein bottle can be obtained from a cylinder depending on the directions of two ending cycles.
CHAPTER 2. PRELIMINARIES

(a) The sphere

(b) The projective plane

(c) The torus

(d) The Klein bottle

Figure 2.4: Plane model and space model for some surfaces [18]
2.3 Graphs on surfaces

Mohar and Thomassen’s book *Graphs on Surfaces* [29] and Beineke and Wilson’s book *Topics in Topological Graph Theory* [6] provide comprehensive summarizations of issues with respect to (w.r.t) graph embedding in general surfaces. We introduce some of them in this and the following section.

A graph $G$ is *embedded* on a surface $S$ if it is drawn on $S$ so that the edges are pairwise disjoint except at their common ends. The drawing of $G$ is an *embedding* of $G$ on $S$. $G$ is *embeddable* on $S$ if such an embedding exists. Embedding on the sphere is often called embedding on the plane for convenience. The sphere and the plane are used interchangeably w.r.t graph embedding. So a graph embeddable on the sphere is called *planar*. A graph embeddable on the projective plane or on the torus is *projective planar* or *toroidal*, respectively.

An embedding $\Pi(G)$ has *genus* (plural form: *genera*) $k$ if it is embedded on $S_k$. The *genus of a graph* $G$ is the minimum genus of an orientable surface on which $G$ can be embedded and it is denoted by $\gamma(G)$. An embedding $\Pi(G)$ has *crosscap number* $k$ if it is embedded on $N_k$. The *crosscap number of a graph* $G$ is the minimum crosscap number of a non-orientable surface in which $G$ can be embedded and it is denoted by $\tilde{\gamma}(G)$.

A *face* of an embedding $\Pi(G)$ is a maximal connected set of points in the relative complement of $G$ on the surface. Intuitively, it is a maximal connected area after removing the embedded graph from the surface. The *boundary* of a face $F$ is a minimal closed walk of $G$ that bounds $F$. The number of edges on its boundary is called the *size* of the face.

The boundary of a face may contain repeated vertices and edges. The repeated vertices and edges form up maximal repeated paths, each of which is called a *singularity* in the face. A face with $k \geq 1$ repeated paths is a *$k$-singularity face*, or a *singular* face. Given a singular face $F$, an *unfolded face* $F'$ is a copy of $F$ where all repeated vertices and edges are substituted with multiple copies. This can be obtained with Algorithm 3. In the unfolded face $F'$, all copies of a vertex $v \in V(F)$ forms a copy set denoted as $C(v)$.

In an embedding $\Pi(G)$, a cycle is *contractible* if it bounds an area that is topologically equivalent to an open disk. Otherwise it is *non-contractible*. The *edge-width*
Algorithm 3 UnfoldFace(face $F$)

1: Let $W$ be the boundary of $F$
2: Create a copy $v_0^0$ for the first vertex $v_0$ in $W$
3: Let $U$ be a walk only including $v_0^0$
4: Set $v_{\text{prev}} = v_0^0$
5: for all edge $e = \{v_{i-1}, v_i\}$ along the walk do
6:   if $e$ is the last edge in the walk then
7:      Add an edge $\{v_{\text{prev}}, v_0^0\}$ to $U$
8:   else
9:      Create a copy of $v_i$ as $v'$
10:     Add an edge $\{v_{\text{prev}}, v'\}$ and the vertex $v'$ to $U$
11:    Set $v_{\text{prev}} = v'$
12: end if
13: end for
14: Embed $U$ on the plane
15: return The face $F'$ bounded by $U$

of $\Pi(G)$ is the length of a shortest non-contractible cycle in $\Pi(G)$. The face-width of $\Pi(G)$ is the minimum number of faces in $\Pi(G)$ that the union of edges on their boundaries contains a non-contractible cycle.

A graph $H$ is forbidden for surface $S$ if

- $H$ is non-embeddable on $S$
- For any edge $e \in E(H)$, $H - e$ is embeddable on $S$

A graph $G$ is a minimal forbidden subgraph for $S$ if it is forbidden for $S$ and $\deg(v) > 2$ for each vertex $v \in V(G)$.

A minimal forbidden subgraph is sometimes referred to as an obstruction. The set of obstructions for a surface $S$ is denoted as $\text{Obst}(S)$. A graph cannot be embedded on a surface if and only if it has no subgraph homeomorphic to an obstruction for that surface. This has been formalized as Theorem 1 and Theorem 2.

Theorem 1. [24] Kuratowski’s Theorem A graph is planar if and only if it does not contain a subgraph homeomorphic to $K_5$ or $K_{3,3}$.

Theorem 2. [35, 7, 3] Generalization of Kuratowski’s Theorem A graph is embeddable on a surface $S$ if and only if it does not contain a subgraph homeomorphic to a graph in $\text{Obst}(S)$. $|\text{Obst}(S)|$ is finite for any $S$. 
The only surfaces whose complete obstruction set are known are the plane and the projective plane. $K_5$ and $K_{3,3}$ are also referred to as the Kuratowski obstructions. A graph homeomorphic to a Kuratowski obstruction is called a Kuratowski graph or a Kuratowski subgraph. The projective plane has 103 obstructions in total and all of them are known [2] and have been listed by Mohar and Thomassen [29].

2.4 Combinatorial representation

For an embedding of a graph $G$ on an orientable surface $S$, the rotation of a vertex $v \in V(G)$ is a cyclic ordering of its neighbors. Without loss of generality (w.l.o.g), we define the ordering to be in clockwise direction. The rotation system of $G$ is then the combination of the rotations of all vertices. A rotation system can be used to represent an embedding as a group of lists.

One embedding on an orientable surface can have multiple equivalent rotation systems depending on the choice of first neighbor in a vertex’s rotation and the direction of ordering (clockwise vs counter-clockwise). Therefore, two rotation systems are equivalent if one can be obtained from another through a series of operations (1) and/or (2) below:

1. Cyclic shifting of a vertex’s rotation.

2. Flipping of the whole graph by reversing rotation of all vertices.

Figure 2.5 shows an example for two equivalent rotations systems for one planar embedding.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$x, v, w$</td>
</tr>
<tr>
<td>$v$</td>
<td>$u, x, w$</td>
</tr>
<tr>
<td>$w$</td>
<td>$u, v, x$</td>
</tr>
<tr>
<td>$x$</td>
<td>$u, w, v$</td>
</tr>
</tbody>
</table>

Figure 2.5: Equivalent rotation systems for one planar embedding

An embedding $\Pi(G)$ on a non-orientable surface $N$ can not be represented with rotation system only. In addition, it should also include edge signatures where each
edge is signed as +1 or -1. Since a global direction of ordering cannot be defined, the direction of two adjacent vertices’ rotations may either agree or disagree. An edge is signed +1 if the direction of its two ends agree and -1 otherwise.

Algorithm 4 by Woodcock [40] and Myrvold and Roth [31] constructs faces of an embedding from its rotation system. For embeddings on orientable surfaces, all edges are signed +1.

Algorithm 4 ComputeFaces(graph $G$, rotation system $\Pi(G)$, edge signatures $Sign$)

1: Let $\mathcal{F}$ be the result set of faces
2: for every edge $\{u, v\} \in E(G)$ do
3: Create two records $[u, v]$ and $[v, u]$
4: end for
5: for all Records $[a, b]$ do
6: if $[a, b]$ is not visited then
7: Create a walk $W$
8: Set $direction = +1$
9: while $[a, b]$ is not visited do
10: Mark $[a, b]$ as visited
11: Add $[a, b]$ to $W$
12: Set $direction = direction * Sign((a, b))$
13: if $direction = +1$ then
14: In the rotation of $b$, let $c$ be the cyclic successor of $a$
15: else
16: In the rotation of $b$, let $c$ be the cyclic predecessor of $a$
17: end if
18: Set $a = b$ and $b = c$
19: end while
20: Add the face bounded by $W$ to $\mathcal{F}$
21: end if
22: end for
23: return $\mathcal{F}$

Equivalence of non-orientable embeddings can thus be checked with Theorem 3.

**Theorem 3.** [29] Two embeddings of $G$ are equivalent if and only if they have the same set of faces.

Theorem 4 presents a generalized version for Euler’s polyhedron formula. Given a rotation system (and edge signatures), the genus (and crosscap number) of its
represented embedding can be calculated by first constructing the list of faces with Algorithm 4.

**Theorem 4.** If a graph $G$ is embedded on $S$ and the embedding has $m$ edge, $n$ vertices, $f$ faces, then

$$n - m + r = \begin{cases} 2 - 2k, & \text{if } S = S_k. \\ 2 - k, & \text{if } S = N_k. \end{cases}$$

### 2.5 Projective plane embedding

Embedding graphs on the projective plane is used as a subroutine for some torus embedding algorithms introduced in this thesis. For projective plane embedding, a few algorithms have been proposed but no implementation is publicly accessible. Therefore, we have developed our own implementation of this subroutine. For this problem, a linear time algorithm was created by Mohar [26] in 1993. A simplified algorithm was later introduced by Myrvold and Roth [31] in 2000, which has a compromised running time of $O(n^2)$. The later algorithm is easier to understand and implement and its compromise in efficiency does not affect the order of running time for algorithms in following chapters. Therefore, we apply the $O(n^2)$ algorithm in our implementations. We give a brief review of the algorithm in this section.

This algorithm is based on the embeddings of Kuratowski subgraphs on the projective plane. Assuming that input graph $G$ is non-planar, a Kuratowski subgraph $K$ can be found. Once an embedding $\Pi(K)$ of $K$ on the projective plane is given, the rest of the graph can be filled into the faces of $\Pi(K)$. In order to search all possibilities, all embeddings of $K$ need to be checked. $K_5$ has 27 non-equivalent embeddings on the projective plane and $K_{3,3}$ has 6.

The rest of the graph consists of a set of bridges. In any embedding of $G$, each bridge $B$ needs to be assigned to some face $F$ of $\Pi(K)$. **Attaching** a bridge $B$ on an unfolded face $F'$ is to, for each attachment entry $\text{Ent} = (\{u,v\}, v) \in \text{AttEntry}(B)$, attach $v$ to one copy of $v$ on $F'$. An attachment is **legal** if a planar embedding of $F' + B$ exists s.t. $B$ is embedded in the region bounded by $F'$. A bridge $B$ may have zero to multiple attachments on $F'$ depending on $|\text{AttEntry}(B)|$ and
$|C(v)|, v \in Att(B)$, among which zero to multiple may be legal depending on the structures. $B$ and $F$ are compatible if $B$ has at least one legal attachment on the unfolded face $F'$ of $F$ and incompatible otherwise. $F$ is an admissible face for $B$ if they are compatible. The set of admissible faces for a bridge $B$ is denoted by $Adm(B)$.

It may happen that a face $F$ is admissible for two bridges $B_1$ and $B_2$, but attaching both bridges to the region of $F'$ causes it to be non-planar with any attachment. In this case, $B_1, B_2$ and $F$ are incompatible. They are compatible if a planar embedding for $B_1$ and $B_2$ in the face bounded by $F'$ exists.

The algorithm tries to solve the bridge assigning problem as a 2-SAT problem. Before transforming, it needs to guarantee that each bridge has two candidate faces at most. This is achieved by arranging one or two candidate faces for each bridge with more than two admissible faces and then enumerating a constant number of arrangements. For each embedding of $K_5$ and $K_{3,3}$ on the projective plane, only a few bridges can have three admissible faces. Such a bridge is called a 3-face bridge. All other bridges have at most two admissible faces. The number of arrangements is thus bounded by 20 for $K_5$ and 4 for $K_{3,3}$.

A 2-SAT problem has the form of a 2-SAT formula. A boolean variable $x$ can be assign value either true or false. A literal is a variable $x$ or its complement $\bar{x}$. A 2-clause is a disjunction between two literals. A 2-SAT formula consists of a set of 2-clauses joined by conjunctions. The 2-SAT problem is to determine weather there is an assignment of true/false values to the variables s.t. the value of the formula is true. This problem has been well studied and can be solved in running time linear to the number of variables and clauses [4].

Given a graph $G$, a subgraph $K$, and its embedding $\Pi(K)$, the $K$-bridge set is denoted with $B$ and the face set is denoted with $F$. To transform the bridge assignment problem to the 2-SAT problem, a variable $(B,F)$ is created for each pair $\{(B,F)|B \in B \land F \in F\}$. A true value indicates $B$ is assigned to $F$ and each $B$ has only one $F$ assigned in each 2-SAT solution. For a bridge $B$ that has only one candidate face $F$, the clause $\{(B,F),(B,F)\}$ is added. For a bridge with two candidate faces $F_1$ and $F_2$, two clauses are added as $\{(B,F_1),(B,F_2)\}$ and $\{(B,F_1),(B,F_2)\}$ to ensure it has exactly one face assigned. For any incompatible 3-tuple $(F,B_1,B_2)$, a clause $\{(B_1,F),(B_2,F)\}$ is added so that they would not be
assigned together.

Given that there are $O(n)$ bridges, $O(1)$ faces and $O(n^2)$ incompatible 3-tuples, the 2-SAT formula has $O(n)$ variables and $O(n^2)$ clauses and can be solved efficiently.

Algorithm 5 shows the scheme for this whole algorithm.

**Algorithm 5** ProjectivePlaneEmbed(graph $G$)

1: if $m > 3n - 3$ then
2:     return false
3: end if
4: if PlanarEmbed($G$) then
5:     return the planar embedding
6: end if
7: Find a Kuratowski subgraph $K$
8: for each embedding $\Pi(K)$ of $K$ on the projective plane do
9:     Set $\mathcal{F} = \text{ComputeFaces}(\Pi(K))$ (Algorithm 4)
10:    Find all $K$-bridges and determine their admissible faces
11:    if a bridge $B$ has no admissible face then
12:        return false
13:    end if
14:    Compute compatibility between each pair of bridges w.r.t. each face $F \in \mathcal{F}$
15:    for each arrangement of 3-face bridges do
16:        Construct a 2-SAT problem and solve it
17:        if the 2-SAT problem is solved then
18:            Embed each bridge into its assigned face
19:            return the formed projective plane embedding
20:        end if
21:    end for
22: end for
23: return false

Algorithm 6 describes an *edge elimination* process that is used to, given a graph $G$, find a minimal subgraph $H$ fulfilling the properties indicated by $\text{Condition}(G)$. Given a graph $G$ which is not projective planar, this process can be used to find a forbidden subgraph that is homeomorphic to one of the obstructions of the projective plane. The $\text{Condition}(G)$ in this case is substituted by $\text{NotProjectivePlanar}(G)$, deploying Algorithm 5 above. This process can also be used to find a forbidden subgraph for a non-toroidal graph on the torus.
Algorithm 6 EdgeElimination(graph $G$, condition $Condition(G)$)

1: Set $K = G$
2: for each edge $e \in E(K)$ do
3:     if $Condition(K - e)$ then
4:         Set $K = K - e$
5:     end if
6: end for
7: return $K$
Chapter 3

Review on Torus Embedding Algorithms

In this chapter, we provide a general review of the existing torus embedding algorithms. First, an algorithm that converts the torus embedding problem into a planar embedding problem is presented. Then we introduce an approach based on bridges, including both the enumerative version and improvements with recursion. Following that is a finer-grained algorithm based on paths in bridges. All these algorithms have exponential running time. Finally, we briefly review existing algorithms with polynomial running time.

3.1 Conversion to planar embedding problem

Since planar embedding algorithms have been well established and implemented, one approach to tackle the torus embedding problem is to embed some subgraphs of $G$ on the plane to get an embedding of $G$ on the torus. Neufeld and Myrvold [32] introduced such an algorithm (NM Algorithm) in Algorithm 7 with exponential running time.

NM Algorithm exploits the property that the torus is topologically equivalent to the sphere with a handle. Given a graph $G$ that is non-planar but embeddable on the torus, it must have at least one cycle embedded around the handle and the rest of $G$ embedded on a plane in an embedding of $G$ on the torus. An embedding of $K_5$
is shown in Figure 3.1a as an example and Figure 3.1b represents the general case.

More formally, in any embedding $\Pi(G)$ of $G$ on the torus, there must be a cycle of $G$ whose embedding in $\Pi(G)$ forms a non-contractible cycle $C$. NM Algorithm cuts both the graph and the torus along $C$, duplicates $C$ (Figure 3.1c), and transforms the problem of embedding $G$ on the torus into embedding a graph on the cylinder, which is essentially a planar embedding problem.

Figure 3.1: Cycles around handle in torus embeddings
Algorithm 7 TorusEmbedNM(graph $G$)

1: if $m > 3n$ then
2:  return $G$ cannot be embedded on the torus
3: end if
4: Find a set $C$ of candidate cycles to be embedded around the handle
5: for each cycle $C$ in $C$ do
6:  Embed $C$ around the handle
7:  Cut the torus along $C$, create two copies $C_1$ and $C_2$ for $C$
8:  For each vertex $v \in V(C)$, the copy of $v$ in $C_1$ is denoted by $v'$ and that in $C_2$ by $v''$.
9:  Let the two faces bounded by $C_1$ and $C_2$ be $F_1$ and $F_2$
10: Add a center vertex $v_{c1}$ and $v_{c2}$ into $F_1$ and $F_2$
11: Add edges between $v_{c1}$ and every vertex on $C_1$
12: Add edges between $v_{c2}$ and every vertex on $C_2$
13: Let $A$ be the set of edges in $E(G) \setminus E(C)$ incident to a vertex of $V(C)$
14: An attachment of $A$ to $F_1$ and $F_2$ is that for each edge $e \in A$ and an end $v$
15:  for each attachment of $A$ to $F_1$ and $F_2$ do
16:  Let $G'$ be the graph after attachment
17:  if PlanarEmbed($G'$) then
18:    if $F_1$ and $F_2$ have same orientations then
19:      Find minimum vertex cut $Cut(F_1, F_2)$ between both faces in $G'$
20:      if $|Cut(F_1, F_2)| \leq 2$ then
21:        Flip the graph on the cut vertices
22:      else
23:        return false
24:    end if
25:  end if
26:  return combined torus embedding $\Pi(G)$ by identifying vertices
27: end if
28: end for
29: end for
30: return $G$ cannot be embedded on the torus
3.1.1 Finding a proper cycle

While NM Algorithm is based on a cycle embedded around the torus, the cycle varies among all possible embeddings. It cannot be uniquely identified until the embedding is given.

To fix the problem, the algorithm provides a set of candidate cycles for $G$, which contains, in any embedding $\Pi(G)$, at least one non-contractible cycle. By checking all these cycles, it is guaranteed that the algorithm either finds an embedding of $G$ or announces $G$ unembeddable on the torus correctly. The set of cycles is computed based on Theorem 5 and Theorem 6 below:

**Theorem 5.** [32] If $G$ has genus one, at least two cycles of a cycle basis are not contractible in any particular toroidal embedding (which two can depend on the torus embedding considered).

**Theorem 6.** [32] If $G$ is genus one, then it contains a Kuratowski obstruction ($K_5$ or $K_{3,3}$), and at least two cycles in the cycle basis for the obstruction are noncontractible in any particular torus embedding.

According to a former implementation of Skoda and Mohar [33], a better conclusion can be drawn as Theorem 7. The result in this theorem can be obtained with a closer observation and analysis of all embeddings of $K_5$ and $K_{3,3}$ on the torus.

**Theorem 7.** [33] If $G$ is genus one, then it contains a Kuratowski obstruction ($K_5$ or $K_{3,3}$). In the cycle basis of any $K_4(K_{3,2})$ subgraph of the obstructions, at least one cycle is non-contractible in any particular torus embedding.

With Theorem 7, it is only necessary to check one $K_4(K_{3,2})$ subgraph of the Kuratowski obstruction. Since it is free to choose any of the obstructions and their subgraphs, a heuristic can be applied to select one that requires smallest exponential term in later stages. This can be helpful in avoiding long branches of a Kuratowski subgraph.

3.1.2 Generating torus embedding

As indicated in Algorithm 7, $A$ is the set of edges in $E(G) \setminus E(C)$ incident to a vertex of $V(C)$. For each edge $e \in A$ and an end $v$ of $e$ in $V(A) \cap V(C)$, $e$ is incident
to either $F_1$ or $F_2$ after $A$ is attached. The number of different attachments of $A$ to $F_1$ and $F_2$ is bounded by $2^{2^{|A|}}$ since each edge has at most two ends to be attached. In order to find the toroidality of original graph $G$, all these attachments have to be checked and that leads to an exponential running time $2^{O(n)}$, since $|A|$ is bounded by $m = O(n)$.

Let $G'$ be the graph after cutting $G$ along $C$, duplicating $C$, and attaching $A$. Given an embedding of $G'$ on the plane, we identify $F_1$ and $F_2$. More specifically, for each vertex $v \in C$, we identify its two copies $v'$ and $v''$. For each edge $e \in E(C)$, we identify its two copies $e'$ and $e''$. For a vertex $v \in V(C)$, let the two edges on $C$ and incident to $v$ be $e_1$ and $e_2$. The rotation of each $v \in V(C)$ is constructed by combining the rotation of $v'$ and $v''$, in accordance to the order of copies of $e_1$ and $e_2$. In this way, we obtain a rotation system $\Pi(G)$ of $G$.

However, a planar embedding for $G'$ does not necessarily guarantee that $\Pi(G)$ is an embedding in the torus. To observe this we need to first check the torus and the Klein bottle. Both surfaces can be transformed into a plane by first cutting into a cylinder. Likewise, both of them can be transformed from a plane by first constructing a cylinder and then identifying the two ending cycles. However, the directions of identifying the cycles differ from each other, resulting in different surfaces, as shown in Figure 2.4c and Figure 2.4d.

For a planar embedded graph, it has a global unified orientation that applies to all its faces. The orientation defines a direction for the bounding cycle of one face, which is an order of the vertices of the cycle. For $F_1$ and $F_2$ in a planar embedding, the vertices on boundary are correspondent but the orientations could be either same or on contrary.

In the case that $F_1$ and $F_2$ have the same orientations, the surface would follow Figure 2.4d and transform into a Klein bottle when we identify the cycles. Consequently, we need to change the orientation of one face before identification $F_1$ and $F_2$. This can be done by flipping part of the graph that contains only one face in $F_1$ and $F_2$, while ensuring the flip does not bring in edge cross that breaks the embedding.

To change the orientation of a face, we first partition the graph at its minimum vertex cut between $F_1$ and $F_2$. Denote the vertex cut set as $C(F_1, F_2)$, then $G' - C(F_1, F_2)$ consists of two connected components $G_1$ and $G_2$, containing $F_1$ and $F_2$, respectively.
correspondingly. Without loss of generality, we can flip $F_2$ by reversing rotations of all vertices in $V(G_2)$. For any vertex $v \in C(F_1, F_2)$, let $\text{Connection}(v, G_2)$ be $\{\{v, u\}|u \in V(G_2)\}$, then the order of $\text{Connection}(v, G_2)$ should also be reversed in $v$’s rotation.

The flip requires $|C(F_1, F_2)| \leq 2$ so that no edge cross would be introduced. Otherwise, the relation of orientations between $F_1$ and $F_2$ is forced by the planar embedding and we can safely draw the conclusion that $\Pi(G')$ cannot be converted into a torus embedding of $G$.

### 3.2 Bridge based embedding algorithm

According to Kuratowski’s Theorem (Theorem 1), a non-planar graph $G$ must have a subgraph $K$ homeomorphic to $K_5$ or $K_{3,3}$. It is then possible to embed $K$ on the torus first and then embed the rest of the graph into the faces of $\Pi(K)$. The procedure is called the embedding extension, where $\Pi(K)$ is extended to an embedding $\Pi(G)$.

In this section, we introduce an approach based on embedding extension from embeddings of $K_5$ and $K_{3,3}$. To ensure that all possibilities are checked, we need to enumerate all the non-equivalent embeddings of $K$ on the torus, which amounts to 231 for $K_5$ and 20 for $K_{3,3}$. The rest of the graph consists of a list of $K$-bridges and we need to embed each of them into some face of $\Pi(K)$.

#### 3.2.1 Enumerative bridge embedding

Given a non-planar graph $G$ and its Kuratowski subgraph $K$, we can compute the $K$-bridge list $\mathcal{B}$. According to each embedding $\Pi(K)$ of $K$ on the torus, a face set $\mathcal{F}$ can be calculated. Each bridge $B \in \mathcal{B}$ needs to be tested against every face $F \in \mathcal{F}$. In a given pair $(B, F)$, each vertex $v \in \text{Att}(B)$ may have more than one copy in the unfolded face $F'$. Therefore, there are multiple ways to attach $B$ onto $F$ (See Section 2.5). Combining these two aspects, we have a high-dimensional searching space. With the enumerative approach (Algorithm 8), we traverse this space and enumerate all possibilities.

Time complexity of the algorithm depends on the size of the search space. For the innermost part of the nested loop, every bridge is attached to its assigned face
Algorithm 8 TorusEmbedBridgeEnumerative(graph $G$)

1: if $m > 3n$ then
2: \hspace{1em} return false
3: end if
4: if PlanarEmbed($G$) then
5: \hspace{1em} return the planar embedding
6: else
7: \hspace{1em} Find a Kuratowski subgraph $K$ of $G$
8: \hspace{1em} Set $\mathcal{B} = \text{ComputeBridges}(G, K)$ (Algorithm 1)
9: \hspace{1em} for each embedding $\Pi(K)$ of $K$ on the torus do
10: \hspace{2em} Set $\mathcal{F} = \text{ComputeFaces}(\Pi(K))$ (Algorithm 4)
11: \hspace{2em} for each assignment of $B \in \mathcal{B}$ into $F(B) \in \mathcal{F}$ do
12: \hspace{3em} for each attachment of $B$ in $F(B)$ do
13: \hspace{4em} Attach every bridge into its assigned face
14: \hspace{4em} if all faces are planar then
15: \hspace{5em} return an embedding on the torus
16: \hspace{4em} end if
17: \hspace{3em} end for
18: \hspace{2em} end for
19: \hspace{1em} end for
20: \hspace{1em} return $G$ is not embeddable on the torus
21: end if
and planarity is tested for all faces. In this part, a call to the linear time function \textit{PlanarEmbed()} costs $O(n_i)$ for each face, where $n_i$ is degree of graph consisting of the face and assigned bridges, and sums up to $O(n)$ altogether.

As mentioned in Section 2.5, a bridge can have multiple attachments into a face. For the embeddings of $K_5$ and $K_{3,3}$ on the torus, each vertex has at most two copies in an unfolded face. Therefore, each attachment entry has at most two ways to be attached and the number of different attachments we need to consider for each pair $(B, F)$ is at most $2^{|\text{AttEntry}(B)|}$.

Embeddings of $K_5$ and $K_{3,3}$ on the torus have constant numbers of faces, which set rough upper bounds for the number of faces each bridge needs to check against. A tighter upper bound can be found by only considering a subset of all faces. To embed a bridge $B$ into a face $F$, all attachment vertices of $B$ have to lie on the boundary of $F$. A face $F$ fulfilling this requirement is called an attachable face for $B$. The attachable face set of $B$ is denoted by $\mathcal{AF}(B)$. Any admissible face of a bridge $B$ must be attachable for $B$, so we have $\text{Adm}(B) \subseteq \mathcal{AF}(B)$. The implementation can thus be improved by only considering $\mathcal{AF}(B)$ for each bridge $B$. For all embeddings of $K_5$ and $K_{3,3}$ on the torus, $\mathcal{AF}(B)$ is bounded by 4 for any bridge $B$.

Combining above results, running time of the algorithm is shown as Equation 3.1 below. Since $|\mathcal{B}| < m$ and $\sum_{B_i \in \mathcal{B}} |\text{AttEntry}(B_i)| < 2m$, the total running time is exponential in the order $n$ of input graph $G$ as shown below.

$$\prod_{B_i \in \mathcal{B}} (|\mathcal{AF}(B_i)| \cdot 2^{|\text{AttEntry}(B_i)|})$$

$$= O(C_1^m) \cdot 2^{\sum_{B_i \in \mathcal{B}} |\text{AttEntry}(B_i)|}$$

$$= O(C_1^m \cdot C_2^m)$$

$$= 2^{O(n)}$$

where $C_1, C_2 \leq 4$ (3.1)

### 3.2.2 Recursive bridge embedding

While the enumerative approach guarantees correctness by nature, we observe that it brings a large overhead by checking all possible combinations. Among the combinations, many of them share common structures which indicate that the embeddings are not viable. However, the algorithm does not use this information and
has to wait until all bridges are fully assigned and attached, and the embeddabil-
ity for each face is checked. This leads to a huge amount of repetitions that could
otherwise be avoided.

We take a recursive approach instead to improve the average efficiency. Bridges
are assigned and attached one by one and a next bridge is processed only if the
current bridge does not lead to any violation to the embeddability. In this way,
the algorithm would traceback or stop at points where current structure is already
not embeddable and thus save the work on enumerating combinations of remaining
structures.

In the enumerative algorithm, reason causing exponential running time comes
two-fold:

1. Each bridge $B$ may have multiple attachable faces

2. Each bridge $B$ may have multiple ways to attach to an attachable face $F$.

They correspond to two phases for embedding extension: bridge assignment and
bridge attachment. With the first property, a bridge should be treated as a basic unit
in bridge assignment. In the second property, in contrast, a bridge should be further
divided into attachment entries as basic unit for bridge attachment. Improvement
over these two phases are introduced in the following two sections.

3.2.3 Recursive bridge assignment

Based on the enumerative approach, a recursive algorithm follows naturally by
embedding all bridges one by one. Each remaining bridge is embedded against origi-
nal faces plus their assigned bridges. Embedding one bridge could block all remaining
bridges but all remaining bridges still have to be tried to draw the conclusion.

An alternative approach is to update the face set after each bridge attaching, as
shown in Algorithm 9, Algorithm 10 and Algorithm 11. Remaining bridges are then
considered against new face set. Embedding a bridge $B$ into a face $F$ breaks $F$ into
multiple faces. For a remaining bridge $B$, it could possibly benefit from the face set
update in two aspects:

1. Reduce the number of duplicated vertices in each attachable face
Algorithm 9 TorusEmbedBridgeRecursive(graph $G$)

1: if $m \neq 3n$ then
2:     return false
3: end if
4: if PlanarEmbed($G$) then
5:     return the planar embedding
6: else
7:     Find a Kuratowski subgraph $K$ of $G$
8:     Set $B = \text{ComputeBridges}(G, K)$ (Algorithm 1)
9:     for each non-equivalent embedding $\Pi(K)$ of $K$ on the torus do
10:        $\Pi(G) = \text{ExtendEmbedding}(\Pi(K), B)$ (Algorithm 10)
11:        if Extension succeeds then
12:            return Embedding found: $\Pi(G)$
13:        end if
14:     end for
15:     return $G$ is not embeddable on the torus
16: end if

Algorithm 10 ExtendEmbedding(embedding $\Pi(K)$, $K$-bridge set $B$)

1: if $B$ is empty then
2:     return true
3: end if
4: $F = \text{ComputeFaces}(\Pi(K))$ (Algorithm 4)
5: Select a bridge $B$ from $B$
6: for each face $F$ in $F$ do
7:     Find all embeddings of $B$ in $F$ by calling $\text{EmbedBridgeInFace}(B, F)$ (Algorithm 11)
8:     for each embedding $\Pi(K + B)$ do
9:         if $\text{ExtendEmbedding}(\Pi(K + B), B - B)$ then
10:             return true
11:         end if
12:     end for
13:     Remove $B$ from $F$
14: end for
15: return false
Algorithm 11 EmbedBridgeInFace(bridge $B$, face $F$)

1: Set $F' = \text{UnfoldFace}(F)$ (Algorithm 3)
2: Create a vertex $v$
3: for each vertex $u$ along the boundary of $F'$ do
4:    Add an edge $\{u, v\}$
5: end for
6: Find the attachment entry set $\text{AttEntry}(B)$ of $B$
7: for each possible attachment of $\text{AttEntry}(B)$ to $F'$ do
8:    Attach $B$ to $F'$ and form a graph $H$
9:    if PlanarEmbed($H$) then
10:       Map $\Pi(H)$ back to embedding of $B$ in $F$
11:       Add the embedding to result set $\Pi$
12:    end if
13: end for
14: return $\Pi$

2. Reduce the number of attachable faces for $B$

   Since the embedding within a face is a planar embedding, no more duplicated vertex would be introduced after embedding a new bridge. Therefore, each new face is guaranteed to have no more duplicated vertex on the boundary than the original one. On average, it reduces the number of duplicated vertices within each face and thus decrease the exponential term from attaching phase.

   Also, since $B$ is embedded in $F$, $B$’s attachment vertices $\text{Att}(B)$ should all lie on the boundary of $F$, namely $\text{Att}(B) \subseteq V(F)$.

   Let $H$ be a graph constructed by attaching zero to multiple bridges to $K = K_5$ or $K_{3,3}$. For all embeddings $\Pi(H)$ of $H$ on the torus, each vertex $v \in V(H)$ has at most two copies in $V(F), F \in \mathcal{F}(\Pi(H))$. Therefore, embedding a bridge $B$ into a face $F$ of $H$ may result in three cases w.r.t. the number of attachable faces.

   - No new face is attachable for $B$. In this case, it saves all trials on embedding $B$ and other remaining bridges.
   - One new face is attachable for $B$. In this case, the number of attachable faces remains unchanged.
   - Two new faces are attachable for $B$. Although the number of attachable
faces has increased, each new face has less duplication for $\text{Att}(B)$. The lower overhead in attaching phase then offsets the cost of increased face number.

Although the modification incurs extra overhead for maintaining face set, this overhead can first be reduced by only updating the face into which the latest bridge has been assigned and attached. Furthermore, the overhead can generally be ignored compared with the efficiency gained from pruned branches of recursion tree. The average performance of recursive bridge assignment is therefore expected to exceed that of the enumerative version.

### 3.2.4 Recursive bridge attachment

A further optimization with recursion is to accelerate the bridge attachment process by reducing the number of undesirable trials. Similar to the bridge assignment phase, the attachment of a bridge can also be resolved to a series of decisions on smaller graph units. For this purpose, we allow part of attachment entries of a bridge $B$ to be attached and the others are not. Each unattached entry $\{\{u,v\}, u\}$ is attached to a new isolated copy of the vertex $u$. The situation where a bridge has 0 to many unattached entries is called a partial attachment. An attachment of $B$ is also a partial attachment.

Provided a bridge $B$ and its assigned face $F$, since a vertex might be duplicated on the unfolded face $F'$, we need to specify which copy of the vertices an edge attaches to. In the enumerative approach, each attachment entry $\text{Ent} \in \text{AttEntry}(B)$ is attached to a specific vertex copy first and a planarity test would either accept or decline the combination as a whole, revealing no further information to assist judging other combinations. A recursive approach, on the contrary, can attach the edges incrementally and exit early on declined combinations. A detailed description is provided in Algorithm 12 and Algorithm 13. By replacing Algorithm 10 with Algorithm 12, we obtain the full recursive variance (Bridge-base Algorithm) of embedding algorithm based on bridges.
Algorithm 12 ExtendEmbeddingRecursive(embedding Π, K-bridge set B)

1: if \( B = \emptyset \) then
2:     return true
3: end if
4: Set \( \mathcal{F} = \text{ComputeFaces}(\Pi) \) (Algorithm 4)
5: Select a bridge \( B \) from \( \mathcal{B} \)
6: Calculate the attachable face set \( \mathcal{AF}(B) \) of \( B \)
7: for each face \( F \) in \( \mathcal{AF}(B) \) do
8:     Set \( F' = \text{UnfoldFace}(F) \) (Algorithm 3)
9:     Create a vertex \( v \)
10:    for each vertex \( u \) along the boundary of \( F' \) do
11:       Add an edge \( \{u, v\} \)
12:    end for
13:    if AttachBridgeRecursive\((B, F', \Pi, \mathcal{B})\) (Algorithm 13) then
14:       return true
15:    end if
16: end for
17: return false

Algorithm 13 AttachBridgeRecursive(bridge \( B \), unfolded Face \( F' \), embedding \( \Pi \), K-bridge set \( \mathcal{B} \))

1: Denote the subgraph formed by \( F' \) and current partial attachment of \( B \) as \( H \)
2: if \( H \) is non-planar then
3:     return false
4: end if
5: Find the attachment entry set \( \text{AttEntry}(B) \) of \( B \)
6: if \( \text{Att}(B) = \emptyset \) then
7:     return ExtendEmbeddingRecursive(\( \Pi, \mathcal{B} - \mathcal{B} \)) (Algorithm 12)
8: end if
9: Select an entry \( \text{Ent} = (\{u, v\}, v) \in \text{AttEntry}(B) \)
10: for each copy \( v' \) of \( v \) on \( F' \) do
11:    Attach \( \text{Ent} \) to \( v' \)
12:    if AttachBridgeRecursive\((B, F, \Pi, \mathcal{B})\) then
13:       return true
14:    else
15:       Remove \( \text{Ent} \) from \( v' \)
16:    end if
17: end for
18: return false
3.3 Path based embedding algorithm

Another representation for incremental edge attaching has been introduced by Woodcock [40] as Algorithm 14 (Woodcock Algorithm), which attaches at most two attachment edges during each recursive call.

Algorithm 14 StartTorusEmbedWoodcock(graph $G$)

1: if PlanarEmbed($G$) then
2:    return the planar embedding of $G$
3: else
4:    Choose a Kuratowski subgraph $K$ of $G$
5:    for every non-equivalent embedding $\Pi(K)$ of $K$ do
6:      if TorusEmbedWoodcock($G,K,\Pi(K)$) (Algorithm 15) then
7:        return the embedding
8:      end if
9:    end for
10:   return false
11: end if

Instead of embedding each bridge as a whole contiguously, the algorithm embeds a bisecting path $P$ from the bridge $B$ and leave the remaining part $B - P$ for later calls. Given an attachable face $F$ for $P$, at most four attachments in the unfolded face $F'$ need to be considered since $P$ has only two attachment entries. Once the path has been embedded, smaller bridges can be composed from the remaining part $B - P$. The face $F$ will also be replaced by two new faces partially bounded by $P$. While embedding of $P$ and $B - P$ can be noncontiguous, the algorithm chooses a path as the basic embedding unit instead of a bridge.

To further improve efficiency, the algorithm also enforces a heuristic for selecting the bridge with least assignment and attachment possibilities. More formally, it selects a bridge with least penalty $P(B)$. The penalty function is defined as

$$P(B) = \sum_{F \in A_F(B)} \min_{u,v \in Att(B), v \neq u} x_F(v) \ast x_F(u)$$

where $x_F(v)$ is defined as the number of copies of $v$ on $F$ and $A_F(B)$ is the attachable faces of $B$ as defined earlier. A bridge with $P(B) = 0$ indicates that the bridge cannot be embedded into any face.
Algorithm 15 TorusEmbedWoodcock(graph $G$, subgraph $H$, embedding $\Pi(H)$)

1: Set $\mathcal{B} = \text{ComputeBridges}(G, H)$ (Algorithm 1)
2: Set $\mathcal{F} = \text{ComputeFaces}(\Pi)$ (Algorithm 4)
3: Calculate penalty $P(B)$ for each bridge $B \in \mathcal{B}$
4: if $\mathcal{B} = \emptyset$ then
5:   return $\Pi(H)$ is an embedding of $G$
6: else if $\exists B \in \mathcal{B}$ s.t. $P(B) = 0$ then
7:   $\Pi(H)$ cannot lead to an embedding of $G$
8:   return false
9: end if
10: Choose a bridge $B \in \mathcal{B}$ with minimum $P(B)$
11: for every attachable face $F$ of $B$ do
12:   Set $F' = \text{UnfoldFace}(F)$ (Algorithm 3)
13:   Choose a bisecting path $P$ from $B$
14:   Let $u$ and $v$ be the attachment vertices of $P$
15:   for every copy $\{u_{F'}, v_{F'}\}$ of $\{u, v\}$ in $F'$ do
16:     Embed $P$ in $F$ using endpoints $\{u_{F'}, v_{F'}\}$ and result in $\Pi(H + P)$
17:     if TorusEmbedWoodcock($G, H + P, \Pi(H + P)$) then
18:       return true
19:     else
20:       Remove $P$ from $\Pi(H)$
21:     end if
22:   end for
23: end for
24: return false
3.4 Polynomial time embedding algorithm

Polynomial time algorithms for torus embedding also exist. A linear time algorithm was introduced by Juvan, Marinc’ek and Mohar [21] in 1994. An effort to simplify the algorithm was made by Juvan and Mohar [22] (JM Algorithm) in 1998 with running time $O(n^3)$. Both of them follow the embedding extension scheme in Section 3.2 while applying different techniques for the extension stage.

Recall that the running time of Bridge-based Algorithm in Section 3.2 consists of two exponential terms as stated in Equation 3.1 and Section 3.2.2:

$$O(C_m^1) \times 2^{\sum_{B_i \in B} |\text{AttEntry}(B_i)|} = O(C_m^m \times C_m^2)$$

These polynomial time algorithms try to eliminate the second exponential term $C_m^2$ by avoiding singular faces so that each bridge have only one embedding in every face. They further transform the bridge assignment problem into a constant number of $2$-restricted embedding extension without singularity problems, each of which can be modelled as a $2$-SAT problem and be solved in polynomial time [4]. In this approach, they eliminate the first exponential term $O(C_m^m)$.

JM Algorithm uses projective plane obstructions as $K$ and extend from its embeddings. In any embedding $\Pi(K)$ of a graph $K \in \text{Obst}(\mathbb{N}_1)$ on the torus, each face has at most 1 singularity (i.e. has at most one repeated path). JM Algorithm eliminates this singularity by separating such faces with selected paths s.t. duplications are divided into different faces. A further step is taken to arrange at most two candidate faces for each bridge and a constant number of arrangements need to be considered. Within each arrangement, the bridges are assigned to faces by constructing $2$-SAT problems following the process in Section 2.5. We claim that the number of literals in constructed $2$-SAT problem is bounded:

**Claim 1.** In the process of $2$-SAT formula construction, the total number of literals created is linear to the number of bridges.

**Proof.** Given a graph $G$, its subgraph $K$ and an embedding $\Pi(K)$, we denote $K$’s branch vertices with $BR(K)$. Then each $K$-bridge $B$ falls into one of following three categories:

1. $\text{Att}(B) \notin BR(K)$. With an attachment vertex in the interior of a branch, $B$ can be embedded in at most the two faces incident to the branch.
2. $\text{Att}(B) \subseteq B(R(K))$ and $|\text{Att}(B)| \leq 2$. All such bridges can be grouped according to their attachment vertices, yielding a constant limit on the number of bridges in this category.

3. $\text{Att}(B) \subseteq B(R(K))$ and $|\text{Att}(B)| > 2$. Any two bridges with same group of over two attachment vertices cannot both be embedded into one face without duplicated vertex. For a fixed set of attachment vertices, the number of acceptable bridges is therefore bounded by the number of faces. By enumerating all attachment vertex sets, the total number of bridges in this category is also constant-bounded.

Bridges in category 1 generates two literals for each bridge. Category 2 and category 3 each generates constant number of literals. While the total number of bridges is bounded by the number of edges, the construction procedure generates $O(n)$ literals in total.

While these algorithms have polynomial running time, the constants behind the big Oh notation can get large, which reduces their practical efficiency. For example, in Category 2 and Category 3 of Claim 1, the constant bound of literals can reach $2^{|V(K)|}$ where $|V(K)|$ may sometimes exceed 13 in the algorithms.

Furthermore, the algorithms are complex by nature and their implementation become impractical. An implementation of JM Algorithm was announced [1]. However, there are bugs in this implementation [30]. The libraries used in the implementation are also outdated [33]. All these make the implementation not working. In an effort to implement this algorithm, we also observed that the algorithm consists of many subroutines and modules that each needs to be implemented from scratch and requires respectable amount of work. A more detailed description of the modules can be seen in Figure 4.4.

For these reasons, the polynomial running time algorithms have remained for theoretical interest so far.
Chapter 4

A New Torus Embedding Algorithm

In this chapter, we introduce a new algorithm for torus embedding. The algorithm has exponential running time in the worst case but polynomial time in many cases. We start by analyzing existing algorithms to figure out their essential differences and develop a guideline for new algorithms. Then we introduce a general scheme of a new algorithm followed by more detailed explanations on the key issues. Some pre-processing techniques are then discussed to apply to both the new and the previous algorithms. Finally, we discuss the challenges for the algorithm to become polynomial.

4.1 Analysis of previous algorithms

From the algorithms introduced in Chapter 3, a trade off between time complexity and implementation complexity can be observed. Existing torus embedding algorithms can be divided into two categories: exponential time algorithms and polynomial time algorithms. While the exponential algorithms have been implemented, they do not scale well to graphs with large orders. The polynomial time algorithms, on the contrary, are too complex to be implemented. This trade off is shown in Figure 4.1 as the relation between an algorithm’s time complexity and implementation complexity. Here the implementation complexity is roughly measured by the total
number of necessary statements in an implementation.

![Efficiency-implementation trade off for torus embedding algorithms](image)

Figure 4.1: Efficiency-implementation trade off for torus embedding algorithms

While the existing algorithms are at two extremes of the trade off, it is desirable to find a balance in the middle where efficiency is improved but the implementation does not become a burden. To accomplish this, we try to combine the clear outline of exponential algorithms and effective but less sophisticated techniques from polynomial ones.

Despite the variance in efficiency, all existing algorithms follow a common scheme as embedding extension: first embed part of the graph as the frame and then fill in the remaining parts gradually. The exponential time algorithms differ from each other in the selection of the *framing subgraph* and the basic unit for filling (the *filling units*) (Table 4.1).

A general trend can be observed from the table that an efficient algorithm under this scheme would

- Start from a larger framing subgraph, so that more information can be provided in the first place.
- Continue filling with smaller graph units, so to allow early branch pruning on


4.2 Scheme for the new algorithm

We propose a new algorithm (Algorithm 16) based on projective plane embedding. Given a graph $G$ as input, a projective plane embedding algorithm either yield an embedding that provides information for $G$’s torus embedding, or find a forbidden subgraph for the projective plane, with which $G$ can be framed and filled on the torus.

Given an embedding $\Pi(G)$ of $G$ on the projective plane, we can calculate its face-width $fw(\Pi(G))$. $G$ cannot be embedded on the torus if $\Pi(G)$ has face-width at least four [15]. Otherwise, a torus embedding can be constructed efficiently from the projective plane embedding.

For the case where $G$ is not embeddable on the projective plane, a forbidden subgraph $K$ can be found as a subdivision of an obstruction in $\text{Obst}(N_1)$. These obstructions have larger sizes and more complicated structures than those of $K_{3,3}$ and $K_5$, enabling us to set down more specific frames for the whole graph.

4.3 Projective plane embedding tools

One important advantage of this algorithm is the adoption of projective plane embedding tools. A few projective plane embedding algorithms have been introduced in Section 2.5. For consistency with our algorithm, we use the $O(n^2)$ algorithm for this task. An obstruction can be found using edge elimination in time $O(n^3)$.

---

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Framing subgraph</th>
<th>Filling unit</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NM</td>
<td>Non-contractible cycle</td>
<td>Remaining graph</td>
<td>Evade long branches</td>
</tr>
<tr>
<td>Bridge-based</td>
<td>$K_{3,3}$ and $K_5$</td>
<td>Bridge</td>
<td></td>
</tr>
<tr>
<td>Woodcock</td>
<td>$K_{3,3}$ and $K_5$</td>
<td>Path in a bridge</td>
<td></td>
</tr>
<tr>
<td>JM</td>
<td>Projective plane obstructions</td>
<td>Bridge</td>
<td>Avoid singular face</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison among torus embedding algorithms
Algorithm 16 TorusEmbedNew(graph $G$)

Require: Graph $G$ is a connected graph

1: if $m \not\equiv 3n$ then
2: \hspace{1em} return false
3: end if
4: if PlanarEmbed($G$) then
5: \hspace{1em} return the planar embedding
6: end if
7: if ProjectivePlaneEmbed($G$) (Algorithm 5) then
8: \hspace{1em} Let $\Pi'$ be the embedding on the projective plane
9: \hspace{1em} if Face-width($G, \Pi'$)$ \geq 4$ then
10: \hspace{2em} Find minimal subgraph $H$ of $G$ with face width $\geq 4$ through edge
11: \hspace{2em} elimination (Algorithm 6)
12: \hspace{2em} return $H$ as an obstruction for the torus
13: \hspace{1em} else
14: \hspace{2em} Convert $\Pi'$ into an embedding $\Pi$ of $G$ on the torus
15: \hspace{2em} return $\Pi$
16: end if
17: else
18: \hspace{1em} Set $B = \text{ComputeBridges}(G, K)$ (Algorithm 1)
19: \hspace{1em} Find the projective plane obstruction $K$ of $G$
20: \hspace{1em} for each embedding $\Pi(K)$ of $K$ on the torus do
21: \hspace{2em} if ExtendEmbeddingRecursive($\Pi(K), B$) (Algorithm 12) then
22: \hspace{3em} return $\Pi(G)$
23: end if
24: end for
25: \hspace{1em} Find a minimal forbidden subgraph $H$ of $G$ on the torus through edge
26: \hspace{1em} elimination (Algorithm 6)
27: return $H$ as an obstruction for the torus
28: end if
Once an embedding $\Pi(G)$ is found on the projective plane, we need to calculate face-width for the embedding. To do this, we first build a vertex-face graph $G^{vf}$ of $\Pi(G)$ by adding a face vertex $v_{F_i}$ for each face $F_i$ and connect $v_{F_i}$ with each original vertex bounding $F_i$. The new graph is naturally embedded on the projective plane as $\Pi(G^{vf})$. Then the edge-width $ew(\Pi(G^{vf}))$ can be calculated by finding a shortest non-contractible cycle. By construction, this would yield the face width of original graph as $fw(\Pi(G)) = \frac{ew(\Pi(G^{vf}))}{2}$.

In the case where a projective plane embedding is found with face-width less than 4, an approach to convert it into a torus embedding has been introduced by Fiedler, Huneke, Richter and Robertson [15]. According to this conversion, a projective planar graph has orientable genus $\left\lfloor \frac{fw(\Pi_p(G))}{2} \right\rfloor$. A projective plane embedding with face-width 1, 2, or 3 can be converted into a planar embedding or a torus embedding.

### 4.4 Finding all embeddings of a graph

In the embedding extension scheme, we need to enumerate all non-equivalent embeddings of the framing graph $K$. Woodcock [40] showed that the complete set for $K_{3,3}$ and $K_5$ can be found manually. $K_{3,3}$ has 20 non-equivalent embedding on the torus and $K_5$ has 231. However, this manual calculation requires heavy workload and careful analysis and may be error-prone when it comes to 103 obstructions for the projective plane.

To accomplish this task, designing an algorithm that finds all embeddings for a graph would be desirable. Algorithms introduced in previous chapters intend to find only one feasible embedding and do not apply in this case. Amendment from these algorithms also require careful equivalence examination. To ensure that the result set is complete and duplication-free, we propose an algorithm (Algorithm 17) that enumerates all possible rotation systems for a graph and calculate the genus for each of them. Given a rotation system, its genus can be calculated by computing its faces (Algorithm 4) and applying Theorem 4.

The algorithm is deliberately designed to avoid duplicated or equivalent embeddings. For each vertex $v$, it regards the rotation as a circular permutation of incident edges. By fixing a reference edge $e_1^i$ in the rotation of $u_i$, it guarantees that any different permutation of remaining edges yields a non-equivalent rotation for $u_i$. To
Algorithm 17 TorusEmbedAll(graph $G$)

1: Let $U(G) = \{v | v \in V(G) \land \text{deg}(v) > 2\}$ and $W(G) = V(G) - U(G)$
2: Fix an arbitrary rotation for each vertex in $W(G)$
3: if $U(G) = \emptyset$ then
4:    return current embedding is the only embedding for $G$
5: end if
6: for each vertex $u_i \in U(G)$ do
7:    Fix an edges $e_1^i$ as the first edge in its rotation
8: end for
9: Let $S$ be an empty set of embeddings.
10: Select one vertex $u_1$ from $U(G)$ as the reference vertex
11: Select two different edges $e_2^1$ and $e_3^1$ from $\text{inc}(u_1)$ s.t. $e_1^1 \notin \{e_2^1, e_3^1\}$
12: Set $d = \text{deg}(u_1)$
13: if $d$ is odd then
14:    Require that $e_2^1$ be in position range $[2, \frac{d+1}{2}]$ in the rotation of $u_1$
15:    for each ordering combination of remaining edges do
16:        Calculate genus $g$ of the embedding $\Pi$ defined by current orderings
17:        if $g \leq 1$ then
18:            $S = S \cup \{\Pi\}$
19:        end if
20:    end for
21: else
22:    Require that $e_2^1$ be in position range $[2, \frac{d}{2}]$ in the rotation of $u_1$
23:    for each ordering combination of remaining edges do
24:        Calculate genus $g$ of the embedding $\Pi$ defined by current orderings
25:        if $g \leq 1$ then
26:            $S = S \cup \{\Pi\}$
27:        end if
28:    end for
29:    Require that $e_2^1$ be at position $\frac{d}{2} + 1$ in the rotation of $u_1$
30:    Require that $e_3^1$ be in position range $[2, \frac{d}{2}]$ in the rotation of $u_1$
31:    for each ordering combination of remaining edges do
32:        Calculate genus $g$ of the embedding $\Pi$ defined by current orderings
33:        if $g \leq 1$ then
34:            $S = S \cup \{\Pi\}$
35:        end if
36:    end for
37: end if
38: return $S$
avoid the case where two graphs may have reversed rotations in all vertices, an orientation is enforced on a selected reference vertex $u_1$. For this vertex, we require that a second edge $e_1^2$ be in the first half of the permutation after removing $e_1^1$. It might happen that $u_1$ has even degree and $e_1^2$ is positioned in the middle, where a third edge $e_1^3$ should be positioned in first half to enforce the rotation.

Since permutations are enumerated for input graph, the algorithm admits time complexity higher than exponential, which makes it impractical for any graph with medium size. For our purpose, the algorithm is only applied to obstructions for the plane or the projective plane, each of which has at most 13 vertices and 22 edges, and performs well. Furthermore, the algorithm needs to be run on each graph only once since the results can be recorded for later use. This algorithm has been integrated into the new torus embedding algorithm and some results can be seen in Appendix A.

### 4.5 Improvement on non-embeddable graphs

Table 4.1 shows a trend in algorithm efficiency over the framing graph we select. Embeddings of a more complex framing graph provide more fixed structure regarding the whole graph and reduce the size of searching space.

On the other hand, a more complex framing graph may be less efficient when the input graph is not embeddable. For a graph $G$ and a framing graph $K$, $G - K$ is broken into the set of $K$-bridges $B$ to be embedded. As $K$ becomes more complex, $B$ may contain a larger number of smaller bridges. Given two bridges $B_1, B_2 \in B$ and a face $F$, it is possible that $\{F, B_1\}$ and $\{F, B_2\}$ are both compatible (Recall the definition in Section 2.5) but $\{F, B_1, B_2\}$ is incompatible. Therefore, a group of bridges, each of which is embeddable by itself, can constitute a forbidden subgraph when combined. In such cases, we have to enumerate all possibilities for embedding the bridges before a negative conclusion could be drawn. Given that the bridge embedding process calls a linear time planar embedding subroutine in each possibility, this whole running time grows to $2^{O(n)}$. However, if we check a larger bridge that contains conflicting bridges in $B$, the conflicts can be discovered in one call of the planar embedding subroutine.

One example can be seen in Figure 4.2. When the highlighted $K_5$ subdivision in
Figure 4.2: An example of framing graph selection

Figure 4.2a is selected as $K_5$, the graph has three remaining $K$-bridges each including one vertex and three edges. They share a same admissible face set containing two faces in any embedding of $K$ on the torus. The bridges are pairwise incompatible in any face and combined together they cannot be anyhow filled into the frame properly. To draw the conclusion we have to try every combination of embedding them. However, when we select any cycle as framing graph from the highlighted $K_4$ in Figure 4.2b, the planar embedding algorithm can always find a $K_{3,3}$ consisting of the three bridges above. It determines that the graph is non-toroidal in linear time.

In regarding to this aspect, NM Algorithm has an advantage that generally only a few bridges would be generated. It checks planarity immediately after removing the duplicated cycle and skip the following steps upon negative results. For a non-toroidal input graph, there is a high probability that this step could detect non-toroidality early. Given that a large portion of input graphs will be non-toroidal, this detection could be a substantial improvement in average.

This checking has been extracted from NM Algorithm as shown in Algorithm 18. It serves as an independent checking step in our algorithm and the selected cycles are no longer used for the following processing.
CHAPTER 4. A NEW TORUS EMBEDDING ALGORITHM

Algorithm 18 TorusPreprocessNM(graph $G$)

1: Find a group $\mathcal{C}$ of cycle candidates to embed around the handle
2: for each cycle $C$ in $\mathcal{C}$ do
3: Calculate $C$-bridges $\mathcal{B}$ in $G$
4: if All bridges in $\mathcal{B}$ are planar then
5: return checking passed, $G$ may be embeddable on the torus
6: end if
7: end for
8: return $G$ cannot be embedded on the torus

4.6 Component management

While the preprocessing step in Section 4.5 aims at improving performance on observing non-embeddable cases, embeddable cases may need more attention since it is both more important and more time consuming for the whole algorithm. One way to achieve such improvement is by reducing the size of input graph. For a large input graph to be emebeddable, it may have only a few small parts that are hard to embed and the remaining graph can be attached trivially. These small parts combined form a core graph of the input graph, thus reducing the problem to one with a smaller input.

Bi-connected components and tri-connected components [20] are used here to find a core graph. According to Theorem 8 below initially introduced by Battle, etc. [5], any graph with more than one non-planar bi-connected components would be non-embeddable on the torus. On the other hand, given one such component embedded on the torus, remaining components as a whole can be directly attached to the only cut vertex.

Theorem 8. [5] The genus of a graph is equal to the sum of the genera of its bi-connected components.

Similarly, a toroidal graph may contain one or two non-planar tri-connected components. If two tri-connected component coexist, it is required that their obstructions together form a bi-connected graph. These two components are then combined as the core graph. After embedding the core graph, remaining planar components can be attached by identifying and removing corresponding edges gradually. It it worth mentioning that tri-connected component calculation is non-trivial
to implement. One implementation publicly available can be found in Open Graph Drawing Framework (OGDF) [10, 17].

**Algorithm 19 TorusEmbedCoreGraph(graph G)**

**Require:** Graph G is a connected graph

1: Find bi-connected components BC of G and test their planarity
2: if all components are planar then
3: return the embedding on the plane
4: else if more than one components are non-planar then
5: return false
6: end if
7: Let BC be the only non-planar bi-connected component
8: Find tri-connected components TC of BC
9: if One component TC is non-planar then
10: return TC
11: else if Two components TC1 and TC2 are non-planar then
12: return The combination of TC1 and TC2
13: else
14: More than two non-planar tri-connected components exist
15: return false
16: end if

With above conditions considered, given a graph G and its only non-planar bi-connected component BC, we can induce a core graph H that consists of one or two tri-connected components of BC (Algorithm 19). Taking into account that embeddable graphs are sparse in general, this could be considerable reduction in running time, especially for the algorithms where search space size is exponential on input graph size.

After integrating component management and the pre-processing step in Section 4.5, the whole algorithm is shown in Algorithm 20.

### 4.7 Towards polynomial time algorithm implementation

While the algorithm introduced in this chapter has exponential time complexity, its framework and techniques are similar to that of JM Algorithm. The implementation can be, to a large extent, reused or be based on to develop an implementation of JM Algorithm. Figure 4.3 and Figure 4.4 shows schematic callings graph for the
**Algorithm 20** StartTorusEmbedNew(graph $G$)

**Require:** Graph $G$ is a connected graph

1. if $m > 3n$ then
2. return false
3. end if
4. if PlanarEmbed($G$) then
5. return the planar embedding
6. end if
7. if TorusPreprocessNM($G$) fails (Algorithm 18) then
8. return false
9. end if
10. Set $CG =$ TorusEmbedCoreGraph($G$) (Algorithm 19)
11. if $CG$ not found then
12. return false
13. end if
14. if TorusEmbedNew($CG$) (Algorithm 16) then
15. Let $\Pi(CG)$ be the embedding
16. Combine $\Pi(CG)$ and the planar embedding of $G - E(CG)$ to form an embedding $\Pi(G)$ of $G$
17. return $\Pi(G)$
18. else
19. return false
20. end if


new algorithm and JM algorithm, respectively. In both graphs, accomplished modules are colored grey and the remaining ones are colored white. Dashed rectangles in Figure 4.3 represent modules in the new algorithm that might run in exponential time. Further realizing these modules by the approaches in JM Algorithm would give a polynomial time implementation of JM Algorithm. The challenges in doing so will be explained shortly. It can be observed that our implementation covers over half of JM algorithm.

JM Algorithm consists of four major parts: 1) Projective plane tools; 2) Enumerating all embeddings of a graph; 3) Component management; and 4) Embedding extension. Among them, the first three are implemented in our algorithm and can be reused as-is.

The embedding extension part is the major difference between the new algorithm and JM Algorithm. In the new algorithm, we complete this part with a recursive embedding module (Algorithm 12) that embeds and attaches the bridges recursively, resulting in exponential running time. In JM Algorithm, this part consists of a series of conversions that finally convert the original problem to a linear number of subproblems where each of them is a 2-restricted embedding extension without singularity. For the latter problem, a module solving it by transforming into a 2-SAT problem has already been implemented in the projective plane embedding algorithm of Section 2.5.

To fully implement JM Algorithm, we need to implement the polynomial time embedding extension function. Major remaining modules include:

1. 1-singularity embedding

   For the case where a projective plane obstruction might have 1-singularity faces after embedded on the torus, further processing is required to eliminate this singularity so that the embedding extension without singularity algorithm could be applied. This elimination is achieved by dividing each singular face into multiple faces and separate the duplications. According to each particular embedding, the algorithm either finds a few paths or cuts through certain vertices to divide the singular faces.

2. Convert obstruction(in tri-connected graph embedding)
Figure 4.3: Calling graph for the new Algorithm
Figure 4.4: Calling graph for JM Algorithm

See original paper [22] for detailed description for each module
In order to eliminate all local bridges as required by the 1-singularity embedding problem, it is necessary to convert an obstruction $K$ of $G$ to another graph $K'$.

3. **Convert obstruction (in 2-separable embedding)**

A **2-separable embedding** refers to the case where the core graph is constructed from two non-planar tri-connected components. This conversion also intends to eliminate local bridges for embedding extension.

According to Claim 1, the existence of local bridges does not increase the order of time complexity. So we claim that this conversion is optional towards polynomial time complexity.

4. **Multi-face bridge arrangement**

A bridge with more than two admissible faces is a **multi-face bridge**. While 2-restricted embedding extension without singularity has been implemented in projective plane embedding, it requires that each bridge has at most two candidate faces. In the projective plane embedding algorithm, a face-arrangement module was specifically designed to fulfill this requirement and the implementation depends on analysis results for the embeddings of $K_5$ and $K_{3,3}$ on the projective plane.

In order to apply the same strategy for JM algorithm, 103 projective plane obstructions ($\text{Obst}(N_1)$) need to be analyzed and each of them may be modified by above conversions to form a framing graph $H$. Furthermore, each framing graph $H$ can have multiple embeddings in the torus. It is thus desirable to automate this process by enumerating all possibilities, which is still bounded by a constant related to the size of framing graph.

Among the above modules, module 1 to 3 share some common challenges in their implementation.

- The module consists of a few cases, each of which forms an independent subroutine and contains many details, making it both hard to fully understand and error-prone in implementation.
• The module is specifically designed for a special category of graphs and functions have to be implemented from scratch except for basic ones.

Although developing a correct implementation of module 4 is relatively easier than the others, this process of enumerating all possible arrangements can lead to large constants when many multi-face bridges exist. Further improvement may be obtainable by analyzing the embeddings of $Obst(N_1)$ on the torus. Although the total number of embeddings is huge, this analysis only need to be done once and can be conducted automatically.
Chapter 5

Computational Results

In this chapter, we show experimental results for the new algorithm.

5.1 Implementation details

NM Algorithm (Algorithm 7) and our new algorithm (Algorithm 20) (including processing steps) have been implemented in C++. Both implementations are based on the OGDF [10] library, which includes many graph data structures and useful algorithms. Besides efficient implementation of planar embedding algorithms, one major advantage of this library over others is that it contains an implementation of SPQR-Tree [17], which is used for calculating tri-connected components of a graph and fairly complex to build from scratch. In the implementation of the new algorithm, we also include a graph matching library VFLib [16] for graph homeomorphism.

5.2 Experiments environment setup

We run our implementations of on a laptop with Intel(R) Core(TM) i3-2350M CPU 2.3GHz, 8GB physical memory and operating system Windows 7. While the processor has multiple cores, only one of them is used for the experiments during whole time. This comes both for simplicity and for consistency with computational results in others’ research.

The algorithms are tested with three categories of instances. Category 1(C1)
instances are based on Delaunay triangulations of point set taken from TSPLIB [34]. These original graphs were widely used in research on planar graphs. While these graphs are planar, they can make good inputs for torus embedding by adding some additional edges.

Category 2(C2) instances are randomly generated toroidal graphs. Graphs of this category are generated with Algorithm 21. These graphs have edge number between $2n$ and $3n$ so that the generated graph is neither trivially toroidal nor trivially non-toroidal. By checking toroidality (with the new algorithm) before adding an edge, it ensures the final graph to be toroidal. They are also guaranteed to be simple, connected and non-planar.

**Algorithm 21** GenerateRandomToroidalGraph(order $n$)

1. Create a graph $G$ with $n$ vertices
2. Set $m$ as a random value between $2n$ and $3n$
3. while $G$ has less than $m$ edges or $G$ is planar do
4. Find two random vertices $u, v \in V(G)$ such that $u \neq v$, $\{u, v\} \notin E(G)$ and adding $\{u, v\}$ does not break toroidality of $G$ (checked with Algorithm 20)
5. Add edge $\{u, v\}$ to $G$
6. end while
7. Make $G$ connected by adding an edge between one fixed connected component to each of the others

Category 3(C3) instances are randomly generated graphs with edge number between $n$ and $3n$ where toroidality is unknown. It follows a same procedure as Algorithm 21 except that toroidality is not checked.

Although connectivity is ensured in all input graphs, both algorithms can deal with disconnected cases by embedding each connected component independently.

We will compare performance of our algorithm and NM Algorithm with these categories as input. All timings below are in milliseconds and any period less than one millisecond is indicated by a 0 in the results.

### 5.3 Category 1 instances

Graphs of this category are based on Delaunay triangulations of point set. These triangulations have the property that each face except for the outer one is bounded
by three edges. For one such triangulation $H$ and its outer face $F$, adding one edge $e$ that does not lie on $F$ ($e = \{u, v\} \notin V(F)$) to $H$ breaks its planarity but the resulting graph should be still toroidal. Further additional edges has a big chance of breaking its toroidality. In our experiments, we construct C1 graphs by adding one or two random edge to these triangulations.

We divide C1 graphs further into two subcategories: C1A and C1B. C1A graphs add one random edge to the triangulations and C1B graphs add two. Therefore, C1A graphs are all toroidal and C1B are mostly non-toroidal. With this construction, C1A and C1B represent two boundary cases of input: toroidal graphs that are “close” to be non-toroidal and non-toroidal graphs that are “close” to be toroidal. C1A graphs are harder to embed than other toroidal graphs considering its larger number of edges. Toroidality of C1B graphs are harder to detect because more edges could lead to trivial non-toroidality detection by pre-processing steps. As a result, they can properly represent worst case inputs for the algorithms.

We have recorded running time of both algorithms on these two subcategories as shown in Table 5.1. For each subcategory, we test 100 graphs for each graph order and show their average running time (Avg), worst case running time (Max) and the

| $|V(G)|$ | $|E(G)|$ | # | NM Algorithm | New Algorithm |
|-------|-------|---|-------------|---------------|
|       |       |   | Avg | Max | Std | Avg | Max | Std |
| 51    | 141   | 100 | 191  | 8732| 900 | 175 | 558 | 132 |
| 130   | 378   | 100 | 3541 | 70820| 11756 | 671 | 2455 | 488 |
| 225   | 623   | 100 | 72985 | 949600| 234513 | 3216 | 19400 | 4366 |
| 280   | 789   | 100 | -1   | -   | -   | 2604 | 192400 | 19313 |

(a) C1A graphs as input

| $|V(G)|$ | $|E(G)|$ | # | NM Algorithm | New Algorithm |
|-------|-------|---|-------------|---------------|
|       |       |   | Avg | Max | Std | Avg | Max | Std |
| 51    | 142   | 100 | 234  | 5830| 795 | 183 | 386 | 77 |
| 130   | 379   | 100 | 4692 | 70820| 22458 | 654 | 1878 | 345 |
| 225   | 624   | 100 | 25575 | 188500| 109473 | 2318 | 7328 | 1777 |
| 280   | 790   | 100 | -1   | -   | -   | 600 | 4811 | 1052 |

(b) C1B graphs as input

Table 5.1: Results with C1 graphs as input

1The algorithm does not end in 24 hours for at least one input graph
2The algorithm does not end in 24 hours for at least one input graph
(a) Average running time for C1A graphs
(b) Maximum running time for C1A graphs

(c) Average running time for C1B graphs
(d) Maximum running time for C1B graphs

Figure 5.1: Running time graph on C1 inputs
standard deviation(Std). Average and worst case running times are also visualized in Figure 5.1.

5.4 Random instances

Category 2 instances are randomly generated toroidal graphs with edge number between $2n$ and $3n$. By bounding the number of edges in graphs, it avoids dense graphs that are guaranteed to be non-toroidal and too sparse graphs whose embedding becomes trivial. Unlike the extreme cases in C1A instances, graphs in this category represent more general cases where embedding can be either hard or easy. More importantly, it avoids fixed structures from the base graphs, making the results more general and graph-independent. Results for this category are shown in Table 5.2 and visualized in Figure 5.2.

Category 3 graphs are also randomly generated but their toroidality are not restricted. These graphs are tested to reveal a more general view of the algorithms’ efficiency. While randomly generated graphs have big chance to be non-toroidal, this dataset puts more focus on the performance on non-toroidal graphs. The results are shown in Table 5.3 and visualized in Figure 5.3.

| $|V(G)|$ | $|E(G)|$ | # | NM Algorithm | New Algorithm |
|-------|---------|---|--------------|---------------|
|       |         |   | Avg | Max | Std | Avg | Max | Std |
| 10    | [20, 30]| 100| 7   | 299 | 30  | 29  | 228 | 31  |
| 20    | [40, 60]| 100| 163 | 4961| 636 | 35  | 112 | 14  |
| 30    | [60, 90]| 100| 31  | 630 | 82  | 46  | 114 | 16  |
| 40    | [80, 120]| 100| 72  | 4789| 477 | 64  | 142 | 25  |
| 50    | [100, 150]| 100| 557 | 53820| 5354| 77  | 191 | 29  |
| 60    | [120, 180]| 100| 53  | 2363| 246 | 93  | 197 | 36  |
| 70    | [140, 210]| 100| 131 | 11890| 1182| 115 | 321 | 51  |
| 80    | [160, 240]| 100| 17  | 412 | 59  | 128 | 362 | 55  |
| 90    | [180, 270]| 100| 307 | 12700| 1733| 151 | 387 | 67  |
| 100   | [200, 300]| 100| 836 | 35520| 4960| 178 | 519 | 90  |
| 110   | [220, 330]| 100| 1675| 112300| 13226| 187 | 571 | 83  |
| 120   | [240, 360]| 100| 650 | 12940| 2695| 221 | 343 | 78  |

Table 5.2: Results with C2 graphs as input
CHAPTER 5. COMPUTATIONAL RESULTS

(a) Average running time for C2 graphs

(b) Maximum running time for C2 graphs

Figure 5.2: Running time graph on C2 inputs
## Table 5.3: Results with C3 graphs as input

| $|V(G)|$ | $|E(G)|$ | #  | **NM Algorithm** | **New Algorithm** |
|-------|-------|----|------------------|-------------------|
|       |       |    | **Avg** | **Max** | **Std** | **Avg** | **Max** | **Std** |
| 10    | [10, 30] | 100 | 35     | 868     | 117     | 18     | 171     | 31     |
| 20    | [20, 60] | 100 | 10     | 175     | 31      | 20     | 71      | 17     |
| 30    | [30, 90] | 100 | 4      | 250     | 25      | 27     | 81      | 17     |
| 40    | [40, 120] | 100 | 16     | 1315    | 131     | 28     | 74      | 16     |
| 50    | [50, 150] | 100 | 3      | 170     | 17      | 27     | 56      | 13     |
| 60    | [60, 180] | 100 | 8      | 286     | 33      | 30     | 76      | 16     |
| 70    | [70, 210] | 100 | 4      | 174     | 19      | 35     | 96      | 16     |
| 80    | [80, 240] | 100 | 3      | 103     | 13      | 40     | 85      | 15     |
| 90    | [90, 270] | 100 | 1      | 26      | 3       | 37     | 75      | 15     |
| 100   | [100, 300] | 100 | 9      | 706     | 70      | 40     | 78      | 15     |
| 110   | [110, 330] | 100 | 5      | 383     | 38      | 40     | 87      | 17     |
| 120   | [120, 360] | 100 | 7      | 272     | 37      | 44     | 87      | 18     |
| 130   | [130, 390] | 100 | 2      | 56      | 5       | 42     | 94      | 19     |
| 140   | [140, 420] | 100 | 3      | 124     | 12      | 47     | 138     | 20     |
| 150   | [150, 450] | 100 | 2      | 3       | 1       | 49     | 103     | 20     |
| 160   | [160, 480] | 100 | 2      | 3       | 1       | 48     | 134     | 22     |
| 170   | [170, 510] | 100 | 2      | 3       | 1       | 52     | 102     | 21     |
| 180   | [180, 540] | 100 | 2      | 4       | 1       | 53     | 100     | 20     |
| 190   | [190, 570] | 100 | 3      | 8       | 1       | 65     | 209     | 30     |
| 200   | [200, 600] | 100 | 2      | 5       | 1       | 59     | 116     | 27     |
CHAPTER 5. COMPUTATIONAL RESULTS

(a) Average running time for C3 graphs

(b) Maximum running time for C3 graphs

Figure 5.3: Running time graph on C3 inputs
5.5 Analysis

From the experiment results, we see that the new algorithm has better performance over NM Algorithm on both average performance and worst case performance.

In Table 5.1 and Figure 5.1, NM Algorithm shows a dramatic growth in running time as the input graph size grows. When the input graph has order 280, NM Algorithm fails to yield a result in reasonable time. In contrast, the new algorithm handles this case pretty well by finishing within a few seconds in average and a few minutes at most. Despite that the new algorithm also has exponential running time, its growth w.r.t. input size is gentle compared with NM Algorithm.

C2 graphs have random orders and structures and yield more general results for toroidal graphs. Table 5.2 and Figure 5.2 show that NM Algorithm can have extreme cases while the new algorithm has a stable performance. The average running time for new algorithm shows a curve close to linear and its maximum running time is well below that of NM Algorithm. This trend matches with the results for C1 graphs.

Table 5.3 and Figure 5.3 show a different aspect of the algorithms’ performance. Graphs in C3 consists of more non-toroidal cases than the previous two categories. In the generated graphs, the proportion of toroidal graphs decreases exponentially with the input graph order. This matches the decreasing curve of NM Algorithm and indicates that NM Algorithm is more applicable to non-toroidal cases. For the new algorithm, its running time is still stable over input size regardless of the toroidal proportion. Therefore, its running time is less dependent on the toroidality of input graph. Although its average running time exceeds that of NM Algorithm on non-toroidal graphs, this time is still far less than a second and stays within acceptable range.

From the analysis above, we conclude that the new algorithm is more efficient and more preferable in general. As long as toroidality of input graphs are unknown, we can safely assume that both embeddable and non-embeddable graphs exist and worst cases will happen. Although NM algorithm has some advantage in non-embeddable cases, this advantage does not guarantee the worst case performance.
Chapter 6

Conclusions

In this thesis, we first reviewed existing torus embedding algorithms and analyzed their internal differences and connections. Based on this analysis, we proposed a new algorithm along with details in implementation. Finally, we showed some computational results on the new algorithm. From the discussions in Chapter 3 and Section 4.1, we see that all existing torus embedding algorithms follow a framing-filling process for embedding. The exponential time algorithms differ from each other in the selection of framing graph and filling unit. By comparing and analyzing existing algorithms, we made the conjecture that selecting more complex framing graphs would result in higher efficiency and that elementary filling units are more appropriate for recursive approaches.

Based on the conjecture, we proposed the new algorithm with projective plane obstructions as framing graph and bridges as filling units. From the computational results of Chapter 5, we see that the new algorithm has higher overall efficiency. It has significant advantage in both average and worst case running time. While NM Algorithm may run into cases where toroidality cannot be determined in reasonable time, the new algorithm has more stable performance and ends in a few minutes for the worst cases among all tested input graphs. For NM algorithm, its average running time over solvable inputs are also substantially larger than that of the new algorithm. Besides, the new algorithm has a better focus on constructing the embeddings for embeddable cases, which is more important than non-embeddable cases. The computational results match well with the conjecture.
Besides, we also demonstrated that an implementation of polynomial time algorithms can be obtained based on our work. We have implemented a few major modules for the polynomial time algorithm including a complete projective plane embedding subroutine, a system to generate all embeddings of a graph on the torus, a complete list of projective plane obstructions and a component management system. Based on this work, we can focus on the polynomial time embedding extension of one or two tri-connected components, which is unique to JM Algorithm.

In the area of torus embedding algorithms, there are still many open problems remaining. Based on the content of this thesis and other works we have done, we list some problems here and provide some suggestions for future work.

1. Complete the implementation of JM Algorithm. A polynomial time algorithm could largely reduce running time and extend the size of solvable input. To accomplish the implementation, we need to implement the polynomial time embedding extension algorithm. Challenges for remaining modules and possible approaches to solve them are listed in Chapter 4.7.

2. Find the complete set of obstructions for the torus. Although it is theoretically proven to be a finite set, there is neither an exact number for its size nor a complete set discovered yet. This complete set is important not only theoretically, it can also be used to develop even faster algorithms for torus embedding. Since our algorithm has acceptable running time, this work can be done by doing extensive search on random inputs.

3. Analysis for the embeddings of projective plane obstructions on the torus. Since the complete list of torus embeddings have been generated for these graphs, it is possible to analyze their properties in a larger scale. JM Algorithm gained the progress from some of their properties and we believe a more detailed analysis could facilitate the development of both projective plane and torus embedding algorithms.
Bibliography


naissance et construction de représentations planaires topologiques. 

cient exact algorithms on planar graphs: Exploiting sphere cut decompositions. 
*Algorithmica*, 58(3):790–810, 2010. 2

[14] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gor-
don Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, 
pages 483–484. Springer, 2002. 2

[15] JR Fiedler, John Philip Huneke, R Bruce Richter, and Neil Robertson. Com-
puting the orientable genus of projective graphs. *Journal of Graph Theory*, 
20(3):297–308, 1995. 38, 40


[17] Carsten Gutwenger and Petra Mutzel. A linear time implementation of spqr-

lications, 1994. 8, 10

ACM (JACM)*, 21(4):549–568, 1974. 2


[21] Martin Juvan, Joze Marincek, and Bojan Mohar. Embedding a graph into the 
torus in linear time. 1994. 2, 34

[22] Martin Juvan and Bojan Mohar. An algorithm for embedding graphs in the 

[23] Ken-ichi Kawarabayashi, Bojan Mohar, and Bruce Reed. A simpler linear time 
algorithm for embedding graphs into an arbitrary surface and the genus of 
2


[25] Ming Li and Paul MB Vitányi. *An introduction to Kolmogorov complexity and 


Appendix A

Enumeration of Embeddings

Table A.1 shows the number of embeddings for a few graphs on the torus. These graphs are among the 103 projective plane obstructions \((\text{Obst}(N_1))\) and each of them is a bi-connected combination of two Kuratowski graphs\((K_{3,3} \text{ and } K_5)\) (Not a complete list). It shows that a projective plane obstruction could have over 300 embeddings on the torus, making it impractical to manually calculate all of them.
## Table A.1: Number of embeddings on the torus for some projective plane obstructions

<table>
<thead>
<tr>
<th>Graph</th>
<th>Number of embeddings</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Graph 1" /></td>
<td>306</td>
</tr>
<tr>
<td><img src="image2.png" alt="Graph 2" /></td>
<td>96</td>
</tr>
<tr>
<td><img src="image3.png" alt="Graph 3" /></td>
<td>144</td>
</tr>
<tr>
<td><img src="image4.png" alt="Graph 4" /></td>
<td>144</td>
</tr>
<tr>
<td><img src="image5.png" alt="Graph 5" /></td>
<td>144</td>
</tr>
<tr>
<td><img src="image6.png" alt="Graph 6" /></td>
<td>36</td>
</tr>
<tr>
<td><img src="image7.png" alt="Graph 7" /></td>
<td>24</td>
</tr>
</tbody>
</table>
Appendix B

User Manual

In this appendix, we present a manual for our implementation of the algorithms. The implementation consists of two parts: a library named EMB that can be included into any third party program, and a program based on this library and can be used directly against inputs. We start by introducing the representations of graphs and embeddings.

B.1 Data format

The library uses strings as both input and output for embedding, hiding the internal data structures from users. This brings the advantage that users can use it without any knowledge of the actual implementation details. Also, the library can be easily integrated into other programs without generating any conflict in data structures.

Let $G$ be a graph with $n$ vertices and $m$ edges. In the string representation, each vertex $v$ is represented by a unique number $id(v) \in [0, n)$, named its index. The way in which vertices are indexed is called a labelling of the graph. Each edge $e = \{u, v\}$ is represented by $id(u) id(v)$ (separated by a space) where $id(u) \leq id(v)$. The representation of $G$ is $n m id(u_1) id(v_1) id(u_2) id(v_2) \ldots$, called the graph string of $G$. For example, the graph string of $K_5$ in Figure B.1a is:

\[ K_5: 5 \ 10 \ 0 \ 1 \ 0 \ 2 \ 0 \ 3 \ 0 \ 4 \ 1 \ 2 \ 1 \ 3 \ 1 \ 4 \ 2 \ 3 \ 2 \ 4 \ 3 \ 4 \]

Note that an unlabelled graph can have multiple representations depending on the
labelling. Figure B.1b and Figure B.1c show two labelling of $K_{3,3}$ that lead to different graph strings:

$K_{3,3}$ Labelling 1: 6 9 0 3 0 4 0 5 1 3 1 4 1 5 2 3 2 4 2 5
$K_{3,3}$ Labelling 2: 6 9 0 1 0 3 0 5 1 2 1 4 2 3 2 5 3 4 4 5

![Figure B.1: Labelled $K_5$ and $K_{3,3}$](image)

An embedding $\Pi$ of a graph $G$ in an orientable surface is represented by its rotation system and then transformed to a string representation. The rotation of vertex $v$ is represented with its index followed by the indices of its neighbors, in the order of the rotation and placed within a pair of brackets. The rotation system is represented with the concatenation of all vertices' rotations, called the embedding string of $\Pi$.

![Figure B.2: Embeddings of $K_5$ and $K_{3,3}$ on the torus](image)

For example, the embedding string of one embedding of the $K_5$ in Figure B.1a on the torus is:
APPENDIX B. USER MANUAL

0:[4 1 2 3] 1:[4 3 2 0] 2:[4 3 0 1] 3:[4 0 2 1] 4:[0 3 2 1]

And the embedding is shown in Figure B.2a.

The embedding string of one embedding of the $K_{3,3}$ in Figure B.1b on the torus is:

0:[5 4 3] 1:[4 3 5] 2:[5 4 3] 3:[1 0 2] 4:[0 2 1] 5:[0 2 1]

And the embedding is shown in Figure B.2b.

Given an embedding $\Pi$ of $G$ on the projective plane, its representation should include both the rotation system and the $+1/ -1$ signature for every edge. In the embedding string of $\Pi$, the signature of an edge $e = \{u, v\}$ is shown in the neighbor list of both $v$ and $u$. If $e$ has a $-1$ signature, $u$ will appear as $\text{id}(u) -$ in the representation of $v$’s rotation and vice versa. If $e$ has a $+1$ signature, no additional action need to be taken and no ‘+’ symbol is added.

(a) An embedding of $K_5$

(b) An embedding of $K_{3,3}$

Figure B.3: Embeddings of $K_5$ and $K_{3,3}$ on the projective plane

For example, the embedding string of one embedding of the $K_5$ in Figure B.1a on the projective plane is:

0:[1 2 3 4] 1:[2 0 4 3-] 2:[0 1 4- 3] 3:[4 0 2 1-] 4:[0 3 1 2-]

And the embedding is shown in Figure B.3a.

The embedding string of one embedding of the $K_{3,3}$ in Figure B.1b on the projective plane is:

0:[4- 3 5 ] 1:[4 3 5-] 2:[5 4 3-] 3:[1 0 2-] 4:[0- 2 1 ] 5:[0 2 1-]

And the embedding is shown in Figure B.3b.
B.2 The EMB library

The EMB library includes implementations of the new algorithm (Algorithm 20), NM Algorithm (Algorithm 7) for torus embedding and the algorithm for projective plane embedding introduced in Section 2.5. The implementation is in C++ and a uniform interface is provided for above algorithms:

```cpp
bool embed(string g, string & emb, int alg);
```

The function inputs a graph string \( g \) and embed with an algorithm indicated by \( alg \). It returns \text{true} \) if the input graph is embedded and an embedding string is written to \( emb \). Otherwise, \text{false} \) is returned and \( emb \) will be set to an empty string. Valid values of the \( alg \) parameter are:

1: Embed \( g \) on the torus with the new algorithm

2: Embed \( g \) on the torus with NM Algorithm

3: Embed \( g \) on the projective plane with the algorithm described in Section 2.5

All other values are considered invalid and will terminate its execution.

The EMB library is based on two other libraries: OGDF(v. 2012.07 Sakura) [10] and VFLib(v. 2.0.6) [16], both of which are publicly accessible online. To use the EMB library requires the user code to add \textit{Additional Include Directories} and \textit{Additional Library Directories} and link against all three libraries in compilation.

The EMB library comes with a folder \texttt{Embs}, which contains files for all torus embeddings of \( K_{3,3}, K_5 \) and projective plane obstructions (\( Obst(\mathbb{N}_1) \)). These files exempts the new algorithm from regenerating them for each run. To make use of these files, the folder \texttt{Embs} has to be copied to the parent folder of the user’s executable file. For example, given an executable file \texttt{root/test/embed.out} that uses the EMB library, the folder \texttt{Embs} should to be copied to \texttt{root}.

In summary, using the library requires five steps:

1. Include “embedder.h” in the user code

2. Add calls to the function \textbf{bool embed(string g, string & emb, int alg)};
3. Add paths to the libraries as **Additional Include Directories** and **Additional Library Directories**

4. Link against the libraries in compilation

5. Copy folder **Embs** to the parent folder of generated executable file

A workable code example is provided in Listing B.1.

```cpp
#include <iostream>
#include <string>
#include "embedder.h"
using namespace std;

int main()
{
    string g = "5 10 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4";
    string emb = "";
    if (embed(g, emb, 1))
        cout << emb;
    else
        cout << "Not toroidal";
    return 0;
}
```

**Listing B.1: Example code for using the EMB library**

An example for compilation is shown in Listing B.2. An executable file **emb.out** is generated.

```
g++ -o emb.out -O0 -std=c++0x -I $(OGDF_DIR) -I $(VF_DIR)/include -I $(EMB_DIR)/include -L $(OGDF_DIR)/_release -L $(VF_DIR)/lib -L $(EMB_DIR)/_release -lEMB -lOGDF -lvf -lpthread
```

p.s. $(OGDF_DIR), $(VF_DIR), $(EMB_DIR) are the paths to OGDF library, VF library and EMB library, respectively.

**Listing B.2: Example for compilation with the EMB library**

### B.3 The program

A program **Embedder** has been set up both as a directly usable tool and a demo for the library. As mentioned above, it is recommended to copy the folder **Embs** to
**Embedder**'s parent folder. After that, **Embedder** can be run against input files directly. The input file consists of multiple lines, each of which is a graph string. **Embedder** processes the file line by line, generating an embedding or announcing it to be unembeddable. A brief manual is provided ("[]" indicates an argument is optional) in Listing B.3

**Usage:**

```
Embedder algId inFileName [outFileName] [inPath] [outPath]
```

- `algId`: choose the algorithm to apply
  - 1: Embed on the torus with the new algorithm
  - 2: Embed on the torus with NM Algorithm
  - 3: Embed on the projective plane

- `inFileName`: input file name
- `outFileName`: output file name (Default: `inputFileName+.emb`)
- `inPath`: input file path (Default: ".")
- `outPath`: output file path (Default: ".")

Input graphs from `${inPath}/${inFileName}` and output to `${outPath}/${outFileName}`

**Example:**

```
Embedder 1 a.txt a.emb ../in out
```

**Explanation:**

- `algId=1`, `inFileName="a.txt", outFileName="a.emb", inPath="../in", outPath= "out"
- Input graphs from "../in/a.txt", output results to "out/a.emb"

Listing B.3: Manual for Embedder

Listing B.4 shows a sample input file **a.in**. After running the command “**Embedder 3 a.in a.emb**”, the output file **a.emb** is shown in Listing B.5.

```
5 10 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4
10 17 0 1 0 3 0 7 1 2 1 6 2 3 2 7 3 6 3 8 3 9 4 7 4 8 4 9 5 7 5 8 5 9 6 7
```

Listing B.4: a.in
Graph Id: 1
Graph to embed:
5 10 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4
The graph is embeddable on the projective plane.
0:[1 4 2- 3-] 1:[2 0 3- 4-] 2:[0- 3 1 4-] 3:[4 2 0- 1-] 4:[0 3 1- 2-]

Graph Id: 2
Graph to embed:
10 17 0 1 0 3 0 7 1 2 1 6 2 3 2 7 3 6 3 8 3 9 4 7 4 8 4 9 5 7 5 8 5 9 6 7
The graph is NOT embeddable on the projective plane.

Listing B.5: a.emb