Evaluating algorithms in knot theory via random sampling

Mai Shimodaira

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2022

Abstract

The unknotting problem is the problem of recognising mathematical knots that can be unknotted into a single loop. Since it was first solved by Haken in 1961, various algorithms have been devised. However, most literature focuses on developing algorithms and on the mathematical aspects of the unknotting problem, and there are very few empirical studies have been published. Our aim was to bridge the gap between theory and practice by using the open-source software, *Regina*, to empirically evaluate existing algorithms relating to knot theory. In particular, we tested the following knot algorithms: Burton and Ozlen's unknot recognition algorithm using normal surface theory, Reidemeister moves, and polynomial knot invariants Jones and HOMFLY polynomials. To achieve this, we used an algorithm provided by Chapman and Schaeffer to develop an almost-uniform knot diagram generator that can generate large knot diagrams. In this paper, we also present a new theorem and an algorithm for testing whether a knot diagram is prime. The algorithm was used to develop a prime knot diagram generator in Python.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Mai Shimodaira*)

Acknowledgements

I would like to express my deepest gratitude to my project supervisor, John Longley, for introducing me to this fascinating topic and for his guidance and invaluable advice throughout this project. I would also like to thank my family and friends for their endless support and encouragement.

Table of Contents

1 Introduction **1**

2 Background **4**

2.1 Knots in the 3-sphere 4

2.2 Planar maps and knot diagrams 4

2.3 Prime knots and prime knot diagrams 5

3 Regina's Algorithms Relating to Knot Theory **7**

3.1 Burton and Ozlen's unknot recognition algorithm using normal surface theory 8

3.2 Brute-force search using Reidemeister Moves 10

3.3 Polynomial Knot Invariants 11

4 Random Sampling of Knot Diagrams **12**

4.1 Generating random balanced sequences 13

4.2 Binary trees from balanced sequences 13

4.3 Random blossom trees from binary trees 14

4.4 Closing blossom trees 15

4.5 Link shadows from rooted 4-valent planar maps 17

4.6 Knot shadows by rejection sampling 17

4.7 Knot diagrams from knot shadows 17

4.8 Implementation 18

4.8.1 Random balanced sequences 18

4.8.2 Blossom trees 18

4.8.3 Closing Blossom Trees 20

4.8.4 Representing knot diagrams by classical Gauss Codes 20

5 Random Sampling of Prime Knot Diagrams **21**
5.1 Theorem for recognising composite knot shadows 22
5.2 Algorithm for Sampling Prime Knot Diagrams 26
5.2.1 Generating knot shadows 26
5.2.2 Finding possible well-bracketed sequences s and t 27
5.2.3 Finding a 2-edge cut (x, y) 27

6 Experimental Results and Discussions **30**
6.1 Almost uniform knot diagram generators 30
6.1.1 Correctness 30
6.1.2 Efficiency 31
6.2 Unknot recognition algorithm using normal surface theory 32
6.2.1 Experimental results using the general knot diagram generator 33
6.2.2 Experimental results using the prime knot diagram generator . 35
6.3 Reidemeister Moves 36
6.4 Polynomial Knot Invariants 37
6.4.1 Jones Polynomial 37
6.4.2 HOMFLY Polynomial 38

7 Conclusions **39**

Bibliography **41**

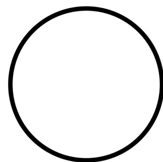
Chapter 1

Introduction

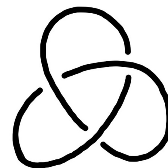
In the mathematical field of knot theory, a knot can be thought of as a flexible and elastic strand with both ends attached together at a single point. The simplest example of a mathematical knot is the *unknot* or equivalently the *trivial knot*, which is a single closed loop. A projection of the unknot onto a plane is illustrated in Figure 1.1a. We call the diagram a *knot diagram*.

There are two fundamental problems in knot theory. The first is the *equivalence problem* - given two knots, are they *equivalent*? In other words, can one knot be smoothly deformed into the other without making illegal moves such as cutting and glueing? The second problem, which is also the main focus of this paper, is a special case of the equivalence problem called the *unknotting problem* - given a knot, is it equivalent to the unknot? A famous example is Haken's Gordian knot with 141 crossings that seems intractable at first sight but can be properly untangled [11]. However, in contrast to knots with loose ends that appear in everyday life, not all mathematical knots can be unknotted. No matter how hard we try, the trefoil knot illustrated in Figure 1.1b cannot be manipulated into the unknot. We call these types of knots *non-trivial knots*.

In fact, the unknotting problem has already been solved in principle. Haken provided



(a) The unknot.



(b) Trefoil knot.

Figure 1.1: Knot diagrams of the unknot and the trefoil knot.

the first algorithm in 1961 by studying the space surrounding the knot and searching for surfaces called the *normal surfaces* [20]. However, it has not been implemented due to its extreme computational complexity. Haken's original algorithm has since then been refined and developed, and numerous algorithms have been devised. We now also know that the unknotting problem lies in the complexity classes NP [22] and co-NP [25] [27]. Thus, the question now is whether there exists a polynomial time algorithm for the unknotting problem. An algorithm announced by Lackenby in 2021 could potentially provide a quasi-polynomial time algorithm that runs slower than polynomial time but faster than exponential time [28]. However, most of the existing algorithms run in exponential time, which reflects the difficulty in tackling the unknotting problem in practice.

Currently, there is a wide range of computer programs for studying and experimenting with knots, such as *Regina* [9], *SnapPy* [14] and *Knotscape* [2]. Yet, most literature focuses on developing algorithms and on the mathematical aspects of the unknotting problem, and we could find very few empirical studies that have been published. Our aim is to fill the gap by undertaking an empirical evaluation of the knot algorithms implemented in *Regina*.

One of *Regina*'s knot algorithms is Burton and Ozlen's unknot recognition algorithm [10], which is based on the work by Haken. A significant result is that their algorithm appears to exhibit empirically polynomial-time performance, despite it being exponential time in theory. A major limitation of their experiment, however, is that they only tested on knot diagrams up to 12 crossings. Thus, it was left unclear whether the same behaviour could be observed with much larger knots. To overcome this problem, we generate random samples of large knot diagrams and feed them to *Regina*. In particular, it is preferable to sample with distribution as close to uniform as possible to avoid bias towards particular types of knots.

The PhD thesis of Chapman [12] provides an algorithm for sampling knot diagrams almost uniformly based on Schaeffer's planar map sampling algorithm [31]. However, the algorithm was implemented mainly for the purpose of studying the mathematical properties of knot diagrams. Therefore, we aim to bridge the gap by applying the random sampling algorithm to the evaluation of sophisticated knot algorithms implemented in *Regina*.

There are three main objectives to this project. The first objective is to develop a knot diagram generator in Python using Chapman's algorithm [12]. A bijection between *rooted link shadows* and a family of trees called the *blossom trees* (see Figure 1.2)

enables us to almost uniformly sample knot diagrams [31]. We discuss the details in Chapter 4. The second aim is to construct a *prime* knot diagram generator to reproduce the experiment done by Burton and Ozlen more closely. We say that a knot diagram is *composite* if there is a pair of edges whose removal disconnects the graph. Otherwise, the knot diagram is *prime*. To develop the generator, we present a new theorem and an algorithm for testing whether a knot diagram is prime. The third goal is to use both generators to empirically evaluate various knot algorithms in *Regina*, with the main focus being Burton and Ozlen's unknot recognition algorithm.

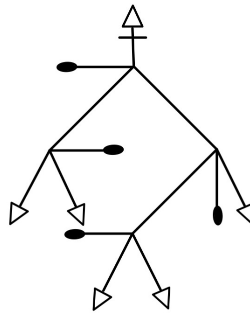


Figure 1.2: A blossom tree.

The remainder of this paper is structured as follows. Chapter 2 briefly introduces the basic concepts of knots. Chapter 3 studies *Regina*'s knot algorithms. Chapter 4 examines Chapman's algorithm for almost uniform sampling of knot diagrams and provides implementation details. Chapter 5 presents a new theorem and an algorithm for sampling prime knot diagrams. Chapter 6 discusses the experimental results and finally, conclusions are presented in Chapter 7.

Chapter 2

Background

We first briefly introduce the basic concepts of knots needed for the rest of this paper. Our mathematical definitions are somewhat informal but are sufficient for our purposes. We assume that the reader has basic knowledge of graph theory.

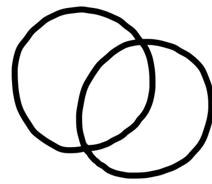
2.1 Knots in the 3-sphere

We consider knots embedded in 3-dimensional space. In particular, it is convenient to work with knots lying in the *3-sphere* that consist of all points in $\mathbb{R}^4$ equidistant from some origin. We denote this space by S^3. It is a *3-manifold*, meaning that locally, it resembles Euclidean 3-dimensional space. We can also think of S^3 by adding a single point at infinity to $\mathbb{R}^3$ with some suitable topology [13]. A knot is an embedding of the circle S^1 into this space, avoiding the point at infinity. The main benefit of working in S^3 over $\mathbb{R}^3$ is that S^3 possesses some nice properties that allow us to use the normal surface theory for recognising unknots in Section 3.1.

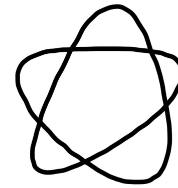
When studying a knot, the first step is to represent it in a simpler form that ideally preserves all its structures and properties. The most popular method involves projecting the knot onto a plane or (almost equivalently) onto a sphere [24], which we describe next.

2.2 Planar maps and knot diagrams

A projection of a knot on a plane can be described as a finite, connected graph with self-loops and multiple edges between pairs of vertices, where each vertex represents a single crossing of the knot. We call such embedding of graphs on a plane *planar maps*.

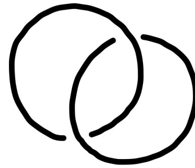


(a) A link shadow.

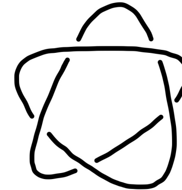


(b) A knot shadow.

Figure 2.1: Examples of link and knot shadows.



(a) A link diagram.



(b) A knot diagram.

Figure 2.2: Examples of link and knot diagrams.

Since every vertex represents a crossing, every vertex has degree 4, and so the projected graph is a 4-valent planar map. We call this map a *link shadow*. In particular, a link shadow is called a *knot shadow* if it is made of one strand only. This can be tested by following the strand and always taking the path straight ahead at each vertex. If we can pass through all the edges, then it is a knot shadow. Examples of link and knot shadows are shown in Figure 2.1. Furthermore, a link shadow is called a *link diagram* if it has additional crossing information on every vertex, where one of the strands represents the over strand, and the other represents the under strand. Similarly, a *knot diagram* is a knot shadow with crossing information. Both link and knot diagrams are given in Figure 2.2. Note that we often measure the size of a knot by the number of crossings in a knot diagram.

Technically speaking, we regard our planar maps as living not just on a plane but on a sphere by adding a point at infinity to the plane.

2.3 Prime knots and prime knot diagrams

In this section, we define particular types of knots called *prime knots* and *prime knot diagrams*. First, a *composition* of two knots can be obtained by removing a small region from each knot and sticking the two knots together without creating new crossings, as

shown in Figure 2.3.

Definition 1. *(Prime knots). A knot is a composite knot if it is a composition of two non-trivial knots. Otherwise, the knot is called a prime knot.*

Definition 2. *(Prime Knot Diagrams). Let G be a knot diagram with more than one vertex. We say that G is composite if there exists a pair of edges in G whose removal disconnects the graph. Otherwise, G is prime.*

Remark. Prime knots have prime knot diagrams, but not every prime knot diagram is guaranteed to represent a prime knot.

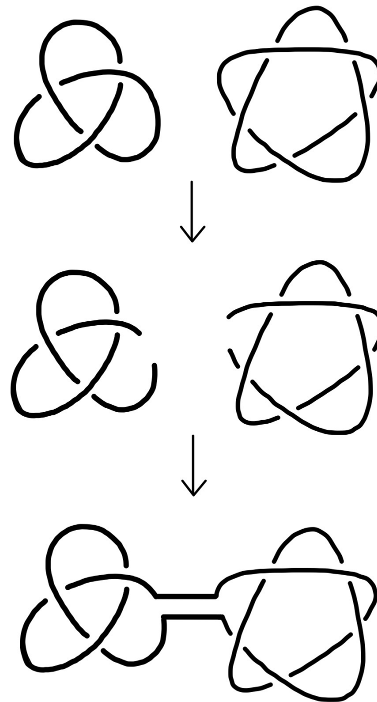


Figure 2.3: Composition of two knots.

Chapter 3

Regina's Algorithms Relating to Knot Theory

In this chapter, we look at *Regina*'s various algorithms relating to unknot recognition: Burton and Ozlen's algorithm using normal surface theory, brute-force search using Reidemeister moves, and two polynomial knot invariants Jones and HOMFLY polynomials. Our primary focus is the algorithm using normal surface theory.

Today, a wide variety of computer programs related to knot theory exist. *SnapPea* was created by Weeks in the 1990s to provide tools for experimentation with hyperbolic 3-manifolds and knots [4] [33]. Its kernel was later used to develop another program for studying hyperbolic structures, called *SnapPy* [14]. Alternatively, there are interactive software such as *Knotscape* [2] and *KnotPlot* [1], that allows one to easily draw knots. However, they lack tools for studying the unknotting problem.

Our project uses an open-source software package called *Regina*, which is designed for studying low-dimensional topology [9]. Its core strength is working with triangulated 3-manifolds and normal surfaces. In particular, it can compute knot algorithms, including Burton and Ozlen's unknot recognition algorithm, Reidemeister moves, and polynomial knot invariants Jones and HOMFLY polynomials. It has a user-friendly graphical interface and has options to interfere closely with the source code and carry out large experiments using its Python scripting tool. Furthermore, *SnapPea* and *SnapPy* can be used in conjunction with *Regina*. The complete documentation and the source code can be found in [9]. In the following sections, we study *Regina*'s algorithms relating to unknot recognition.

3.1 Burton and Ozlen's unknot recognition algorithm using normal surface theory

In this section, we first briefly study Haken's unknot recoginition algorithm and then focus on Burton and Ozlen's algorithm, which is built on the work by Haken.

The normal surface theory has played a central role in the study of the unknotting problem and, more generally, in the field of knot theory. The idea was developed and used by Haken to provide the first solution to the unknotting problem in 1961 [20]. Later, Haken used it to also solve the more general equivalence problem [21] with significant contributions by Hemion [23] and Matveev [5]. The major problem of both algorithms is their extreme computational complexity [24]. Over the years, Haken's original unknot recognition algorithm has been refined and developed by numerous researchers, including Hass et al., who showed that the unknot recognition problem lies in the class NP [22], and Burton and Ozlen, whose algorithm has been implemented in *Regina* [10]. The major strength of the latter algorithm is its sophisticated simplification tools that appear to lead to a polynomial time behaviour when run in practice, even though it is exponential time in theory [10]. In addition, it is guaranteed to provide a conclusive result, unlike many other algorithms that have been implemented [10].

We first give a very brief overview of Haken's unknot recognition algorithm. Interested readers should refer to [5] and [20] for comprehensive accounts of the algorithm and, more generally, the normal surface theory. The main idea is to find a disc embedded in S^3 with the knot as its boundary. Such a disc can only exist if the knot represents the trivial knot. First, a useful property of knots, given by the famous Gordon–Luecke theorem, is that they can be fully described by their exteriors [19]. Therefore, it is common to study the *knot exterior* $\overline{K}$ of the knot K, which can be obtained by thickening it slightly to create what is known as a *tubular neighbourhood* of K and removing its interior from S^3. The resulting space is a 3-manifold.

As with other algorithms based on normal surface theory, we start by triangulating $\overline{K}$. A *triangulation* of a 3-manifold can be constructed by decomposing the space into non-overlapping tetrahedra that are placed in an orderly fashion and fill the entire space [6]. Note that all 3-manifolds can be triangulated [29]. In particular, the knot exterior $\overline{K}$ is a *compact* 3-manifold, implying we can always triangulate the knot exterior of any given knot nicely by a finite number of tetrahedra [13]. Finding a triangulation of a given knot can be done in small polynomial time [10].

The next step is to search for a disc called the *esssential disc* by considering *normal*

surfaces. A *normal surface* in the knot exterior $\overline{K}$ is a union of surfaces that are embedded in the tetrahedra of $\overline{K}$ in a particular way. Each tetrahedron can have none or any of the seven possible triangles and quadrilaterals that are shown in Figure 3.1. Once all possible normal surfaces are found, we look for a normal surface that is an essential disc. If such a surface is found, then the knot is equivalent to the unknot. Otherwise, the knot is a non-trivial knot.

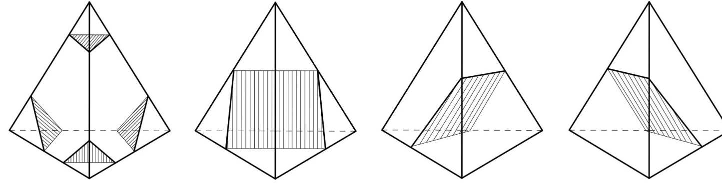


Figure 3.1: A diagram illustrating the possible triangles and quadrilaterals in a tetrahedron.

As aforementioned, Burton and Ozlen's unknot recognition algorithm [10] is based on Haken's algorithm using normal surface theory. However, what differentiates their algorithm from other existing unknot recognition algorithms is its sophisticated simplification techniques. Once a triangulation of the knot exterior $\overline{K}$ is obtained, it first reduces the number of tetrahedra while preserving the underlying 3-manifold. This is achieved by combining a series of simplification techniques such as *edge collapsing* and *local edge* and *vertex moves*. The full details can be found in [8]. This is a crucial step as many algorithms later on in the process have running times that depend heavily on the number of tetrahedra [8]. Although these simplification tools do not necessarily give the smallest possible number of tetrahedra, it is remarkably successful in practice.

After reducing the number of tetrahedra as much as possible, the algorithm follows Haken's algorithm and searches for a disc embedded in S^3 with the knot as its boundary. It looks for a connected normal surface that satisfies a number of constraints by applying integer and linear programming. This part of the algorithm is the root cause of exponential time in the number of tetrahedra. However, a remarkable result by Burton and Ozlen is that in practice, the algorithm runs like polynomial time, at least for up to 50 tetrahedra [10]. Their experiment consisted of testing all 2977 prime knots up to 12 crossings. They found that the simplification techniques eliminate the need to go through some of the computationally expensive steps while searching for a normal surface, which is likely to be the main reason for the polynomial-time behaviour.

A limitation of their experiment, however, is that they only tested on small knots,

and it was left unclear whether the same result could be observed for knots with greater than 50 tetrahedra. As a starting point, Burton and Ozlen tested two knots with approximately 70 tetrahedra, and their results were consistent with the polynomial time behaviour. The major bottleneck is that to test on knots beyond 50 tetrahedra, we require much larger knots, namely knots greater than 12 crossings. One possible solution is to use the prime knot table of the prime knots, which can be obtained from [9] up to 19 crossings. However, the number of prime knots increases exponentially as the number of crossings increases and thus, testing on all prime knots of size greater than 12 would be computationally expensive and impractical. Even if we were to only randomly sample knots from the knot table, we would still require a huge amount of data. As an example, *Regina*'s knot table indicates that there are approximately 294 million prime knots of 19 crossings, requiring about 10 GB of data [9]. To overcome this issue and build on the work by Burton and Ozlen, we create a generator that can almost uniformly sample knot diagrams and prime knot diagrams of any given size. We discuss the algorithms in Chapters 4 and 5.

3.2 Brute-force search using Reidemeister Moves

Another approach to recognising the unknot involves making local modifications to the knot diagram. There are three types of moves called the Reidemesiter moves. The Type I Reidemeister move twists or untwists a small section of the knot. The Type II Reidemeister move adds or removes two crossings at once. Lastly, the Type III Reidemeister move shifts a strand over a crossing. The three types of moves are depicted in Figure 3.2.

A knot can have very different knot diagrams, depending on the choice of projection. A powerful theorem by Reidemeister states that two knot diagrams represent the same knot if and only if there is a sequence of Reidemesiter moves that takes us from one diagram to the other [30]. This implies that the original knot is unchanged by Reidemeister moves. Furthermore, a given knot diagram represents the unknot if and only if it is related to the unknot by a sequence of Reidemeister moves. However, it should be noted that with this approach, one cannot prove that a non-trivial knot cannot be unknotted since the algorithm will not terminate. There is also no known method for efficiently finding a sequence to take us to the unknot, and it is often the case that we need to increase the number of crossings in the knot diagram first.

The problem can be overcome if we know an upper bound for the number of

Reidemesiter moves required to reduce a knot to the unknot. We could then terminate the program once we have exhausted all possible moves. If the knot is still knotted, then it must be a non-trivial knot. Lackenby showed that a knot with n crossings requires at most $(236n)^{11}$ Reidemeister moves to show that it is the unknot [26]. However, the upper bound is still too large to be useful.

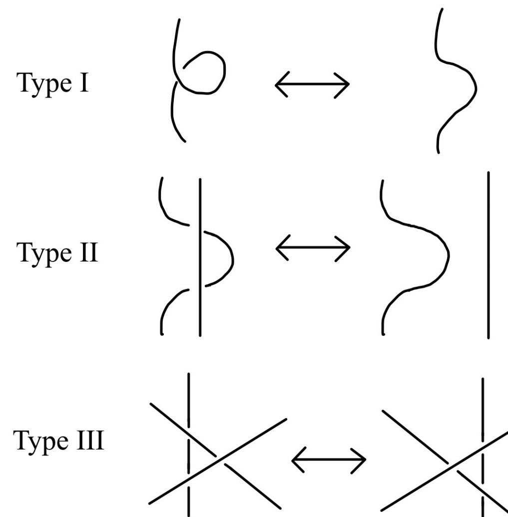


Figure 3.2: Three types of Reidemeister moves.

3.3 Polynomial Knot Invariants

A *knot invariant* is a quantity of knots that is kept the same for knots that are equivalent, and thus, they are unchanged by Reidemeister moves. In particular, we consider the *polynomial* knot invariants Jones and HOMFLY polynomials, which are polynomials associated with knots. A major difference between the two is that Jones polynomials have one variable, whereas HOMFLY polynomials have two. Note that *Regina* can also compute Kauffman polynomials, but it is not an invariant since it can be altered by Type I Reidemeister moves [6].

The two polynomials can be used to distinguish knots to a certain extent. By definition, trivial knots have Jones and HOMFLY polynomials equal to 1 [6]. If two knots have different polynomials, then they are not equivalent. However, two knots with the same polynomial do not necessarily imply that they are equivalent. A major unsolved problem in knot theory is whether there exists a non-trivial knot with Jones polynomial 1. Currently, it has been verified that no such non-trivial knots exist for knots up to 24 crossings [32].

Chapter 4

Random Sampling of Knot Diagrams

As we discussed in Chapter 3, experimenting on all prime knots of a given size is computationally expensive and impractical. Instead, we developed a generator in Python that can sample knot diagrams almost uniformly. The algorithm is based on Chapman's algorithm [12], which relies on the planar map sampling algorithm provided by Schaeffer [31]. Note that the algorithm has already been implemented in Schaeffer's *PlanarMap* software. However, it is written in C, and we found it easier to work with our own code for developing the prime knot diagram generator, which we discuss in Chapter 5.

We first give a summary of Chapman's algorithm [12] for sampling a knot diagram of size n. The algorithm is as follows. 1) Generate a random *balanced sequence*. 2) Obtain the binary tree represented by the bracket sequence. 3) Construct a random *blossom tree* from the binary tree by placing an extra leaf to the root node and adding a *bud* to each internal node. 4) *Close* the blossom tree to produce a rooted 4-valent planar map. 5) Remove the orientation and the choice of *root edge* to obtain a link shadow. 6) Check whether the link shadow represents a knot shadow. Repeat steps 1 to 5 until we obtain a knot shadow. 7) Convert the knot shadow into a knot diagram by adding crossing information.

Steps 2, 5, and 6 are deterministic algorithms, while 1, 3, 4 and 7 involve some randomness. In this chapter, we go through each step in depth, followed by the implementation details.

4.1 Generating random balanced sequences

The first goal is to generate a *balanced sequence* that can later be used to represent a binary tree in Section 4.2. This is achieved by uniformly sampling a *bracket sequence* and turning it into a *balanced sequence* using the Cycle Lemma given by Dvoretzky and Motzkin [18].

A *bracket sequence* is a sequence consisting of open brackets '(' and closed brackets ')'. For example, (())) and ())())()(are both bracket sequences. Note that we are particularly interested in bracket sequences with n open and $n+1$ closed brackets since we later use the open and closed brackets to represent the internal nodes and leaves of a binary tree, respectively. We now give a definition of a *balanced* sequence.

Definition 3. *(Balanced sequence). A bracket sequence $x_1x_2...x_{2n+1}$ of n open brackets and $n+1$ close brackets is balanced if for every $1 \leq i \leq 2n$, the number of open brackets is at least the number of closed brackets in $x_1x_2...x_{i-1}x_i$.*

In other words, a balanced sequence is a well-bracketed sequence with an extra closed bracket at the end. Next, we state the Cycle Lemma provided by Dvoretzky and Motzkin [18], which says that a bracket sequence of n open brackets and $n+1$ closed brackets can be rotated into precisely one balanced sequence.

Theorem 1. *(The Cycle Lemma [18]). Let S be a bracket sequence $x_1x_2...x_{2n+1}$ consisting of n open brackets and $n+1$ closed brackets. Then there exists a unique $1 \leq j \leq 2n+1$ such that the sequence $x_jx_{j+1}...x_{2n+1}x_1x_2...x_{j-1}$ is balanced.*

We apply the method provided by [15] to find the balanced sequence from a random bracket sequence. The algorithm is as follows. Given a bracket sequence $x_1x_2...x_{2n+1}$, iterate over each element of the sequence from left to right. Let *partial sum* $= 0$. If the element is an open bracket, increment *partial sum* by one. If it is a closed bracket, decrement *partial sum* by one. The unique balanced sequence is $x_jx_{j+1}...x_{2n+1}x_1x_2...x_{j-1}$, where $1 \leq j \leq 2n+1$ and *partial sum* at position $j-1$ is at its minimum. As an example, suppose we generate a bracket sequence)(())())())((((). Then $j = 12$ and its balanced sequence is ((())(())())()).

4.2 Binary trees from balanced sequences

We now obtain a random binary tree from the balanced sequence generated in Section 4.1. Unless stated otherwise, all binary trees in this paper are complete binary trees, in

which every node has either zero or two children.

First, note that a binary tree can be expressed as a bracket sequence of n open brackets and $n+1$ closed brackets traversed in depth-first order, where the open and closed brackets represent the internal nodes and leaves, respectively. The binary tree represented by the balanced sequence ((())(())())()) is depicted in Figure 4.1. It is easy to see that binary trees are fully determined by their balanced sequences. Therefore, by the Cycle Lemma [18], we can always find a unique binary tree from a uniformly sampled bracket sequence. In other words, the balanced sequence obtained in Section 4.1 in fact represents a unique binary tree.

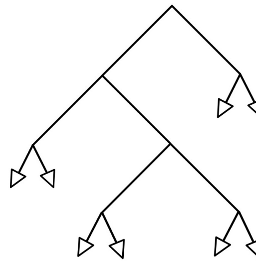


Figure 4.1: A binary tree ($n = 7$) represented by the balanced sequence ((())(())())()).

4.3 Random blossom trees from binary trees

Once we have a binary tree, our next step is to construct a random *blossom tree* by adding a *root leaf* and placing a *bud* to each internal node.

A *blossom tree* is a binary tree where the root node has an extra leaf, which we call the *root leaf*, and every internal node has an additional child node called the *bud* that itself has no children. In other words, a blossom tree of n internal nodes has n buds, $n+2$ leaves and $3n+2$ edges in total, and every node except the leaves and buds have three children.

Any binary tree can be converted into a blossom tree by attaching a roof leaf to the root node and placing a bud to one of the three possible positions on each internal node with equal probability, as shown in Figure 4.2. An example of a blossom tree built from the binary tree presented in Figure 4.1 is given in Figure 4.3. The black oval-shaped nodes and the triangle-shaped nodes represent the leaves and buds, respectively. We use the term *fringe nodes* to refer to the leaves and buds in a blossom tree. A fringe node is *proper* if it is not the root leaf.

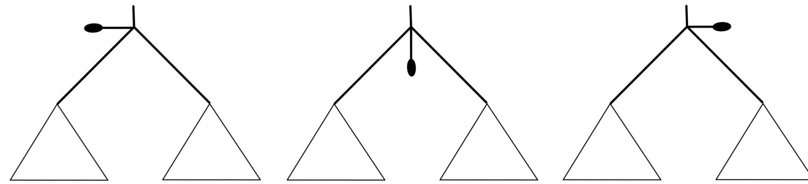


Figure 4.2: Three possible positions to place a bud.

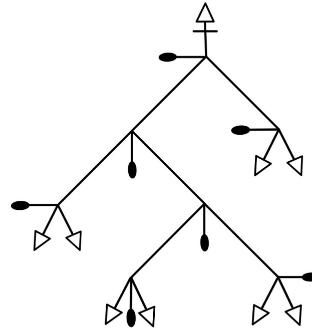


Figure 4.3: A blossom tree ($n = 7$) constructed from the binary tree shown Figure 4.1.

4.4 Closing blossom trees

The fourth step of the algorithm is to *close* the blossom tree and obtain its *closure*, which essentially means connecting all the n buds and $n+2$ leaves by edges. The resulting graph is a so-called *rooted* link shadow, that is, a 4-valent planar map with a designated choice of *root edge* and a choice of orientation of that edge. In this section, we study the algorithm for closing blossom trees and state a remarkable result by Schaeffer, which guarantees a *uniform* rooted link shadow.

We build a closure of a blossom tree by traversing the outer fringe nodes in anti-clockwise direction, starting from the root. If a bud is followed immediately by a leaf, we connect these two nodes by an edge without adding any crossings to the graph. We call these new edges between the fringe nodes *closure edges*. We continue creating closure edges in this manner until there are two unmatched leaves left. This will always be the case since every blossom tree has exactly two more leaves than buds. The partial closure of the blossom tree depicted in Figure 4.3 is given in Figure 4.4. The closure edges are represented by the dashed lines between the fringe nodes. Note that the graph has two unmatched leaves. Lastly, to complete the closure, we join these two leaves by an edge, choosing either of the two directions with equal probability. The resulting graph is a rooted link shadow. A completed closure of the tree is given in Figure 4.5.

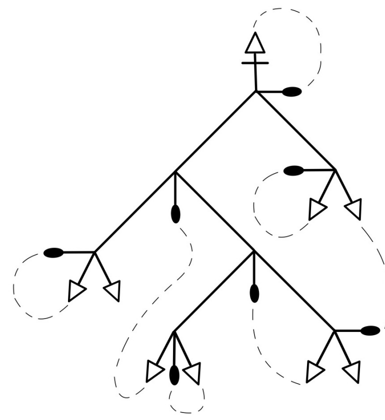


Figure 4.4: The partial closure of the blossom tree depicted in Figure 4.3.

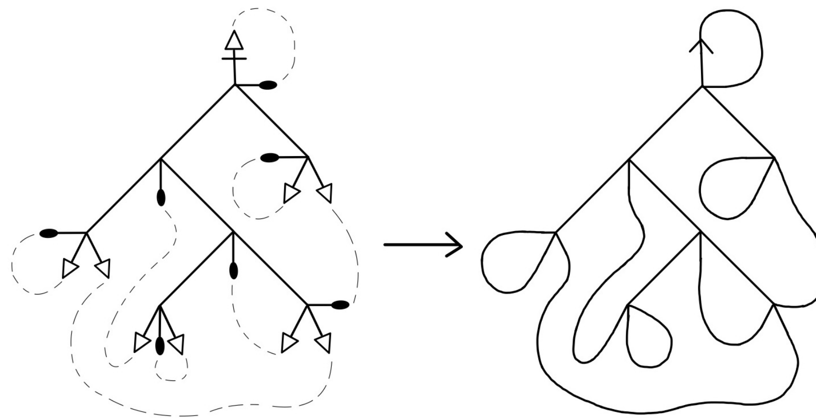


Figure 4.5: A completed closure of the blossom tree depicted in Figure 4.3.

A remarkable result by Schaeffer is that closing the tree in this way provides a bijection between blossom trees and rooted 4-valent planar maps [31]. Since the algorithms described in Sections 4.1, 4.2 and 4.3 allows us to uniformly sample blossom trees, Schaeffer's result implies that we can uniformly sample rooted 4-valent planar maps by closing uniformly sampled blossom trees.

We now define a particular group of blossom trees, called *balanced* blossom trees.

Definition 4. *(Balanced blossom tree). A blossom tree G is balanced if the root leaf is one of the last two remaining leaves in the partial closure of G.*

It is easy to see that any blossom tree can be turned into a balanced blossom tree by taking one of the last two leaves to be the new root. Whether the tree is balanced or not does not make any difference to the closure of a blossom tree, but we see later in Chapter 5 that balanced trees come in useful when testing if a knot diagram is diagrammatically prime.

4.5 Link shadows from rooted 4-valent planar maps

In order to study unrooted link shadows, we remove the orientation on the root leaf of the unrooted 4-valent map obtained in Section 4.4. Note that this process breaks the uniformity of our generator since it introduces symmetry. Simply put, a generator is *almost uniform* over a set M if there exists a subset S of M with $|S| \ll |M|$ such that the generator can sample uniformly from $M - S$ but not from M. Chapman showed that the probability of a random link shadow being symmetric tends exponentially to zero as the size of the graph increases [12]. Hence, our generator can almost uniformly sample link shadows. Although we cannot sample knot diagrams uniformly, Chapman's result suggests that we can safely sample diagrams without any concerns about uniformity, particularly for large knots.

4.6 Knot shadows by rejection sampling

Following the creation of a link shadow, Chapman's algorithm checks to see if it represents a knot shadow by following the strand and testing whether it is made of one strand only. If not, we keep generating link shadows until we obtain a valid knot shadow. A major problem with this rejection sampling approach is that the proportion of link shadows representing a knot shadow becomes exponentially small as the knot size becomes large [12]. Diao et al. take a different approach by converting any link shadow into a knot shadow, which resolves this issue [16]. Their method involves merging the components of the graph while keeping the number of nodes the same. We chose Chapman's rejection sampling method since it is easier to implement, and their experiments suggest that we can sample a valid knot shadow relatively fast for sizes at least up to 60. In fact, we found that knot shadows as large as 150 crossings take only a few seconds to sample.

We check if a link shadow represents a valid knot shadow by following the strand and taking the path straight ahead at each vertex. If we pass through all the edges, then the shadow represents a knot shadow.

4.7 Knot diagrams from knot shadows

The last step of the algorithm is to convert the knot shadow obtained in Section 4.6 into a knot diagram. This is achieved by assigning the under and over strands to each of the

n internal nodes with equal probability. The resulting diagram is a knot diagram of size n.

4.8 Implementation

The almost uniform knot diagram algorithm is implemented in Python. In this section, we give details on how we represent graphs and knots and some main data structures and techniques used to develop an efficient generator.

4.8.1 Random balanced sequences

We express a bracket sequence as a list consisting of integers 1 and -1 representing the open and closed brackets, respectively. To generate a random bracket sequence of n open and $n+1$ closed brackets, we first construct a list of size $2n+1$ of all -1's. Next, we randomly choose an index i, $0 \leq i \leq 2n$, with equal probability. If the i-th element of the list is -1, then we replace it with 1. We repeat the process until we have exactly n open brackets in the list. Its balanced sequence is obtained using the method described in Section 4.1. Recall that the balanced sequence also represents a unique binary tree.

4.8.2 Blossom trees

A blossom tree is constructed from a balanced sequence by a depth-first traversal of its binary tree and attaching a bud to every internal node. In this subsection, we provide two different encoding methods to represent a blossom tree of n internal nodes. In both methods, we start by labelling each node with a unique number between 0 and $n-1$ in depth-first order, as shown in Figure 4.6. The leaves and buds are characterised by the negative numbers -1 and -2, respectively. We use the term *node value* to refer to the label assigned to a node.

The first method is a 1-dimensional list that contains all the internal nodes and the proper fringe nodes in depth-first order, starting from the root node. For example, the blossom tree given in Figure 4.6 can be encoded as $[\texttt{0}, -\texttt{2}, \texttt{1}, \texttt{2}, -\texttt{1}, -\texttt{2}, -\texttt{1}, -\texttt{2}, \texttt{3}, \texttt{4}, -\texttt{2}$ $-\texttt{1}, -\texttt{1}, -\texttt{2}, -\texttt{1}, \texttt{5}, -\texttt{2}, -\texttt{1}, \texttt{6}, -\texttt{1}, -\texttt{1}, -\texttt{2}]$. The second method is a 2d-dimensional list A. We first define the *edge root* to be the following. Label each of the four edges incident to an internal node by a number between 0 and 3 in the anticlockwise direction, starting from the edge above the node, as illustrated in Figure 4.7. The *edge root* of an edge is a tuple (m,n), where m is the node value of the internal node and n is the value

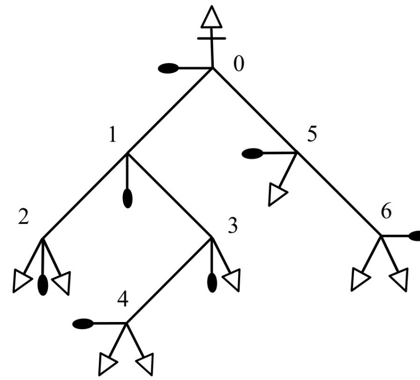


Figure 4.6: Labelling each node of a blossom tree with a number between 0 and $n-1$ in depth-first order.

assigned to the edge. For example, in Figure 4.6, we can say that the edge root $(0,3)$ is connected the edge root $(5,0)$. We can now describe the second encoding method. For $0 \leq i \leq n-1$ and $0 \leq j \leq 3$, the element $A[i][j]$ is either a tuple or a negative number. If $A[i][j] = (m,n)$, then the edge root (i,j) is connected to the edge root (m,n). Otherwise, $A[i][j] = -1$ and $A[i][j] = -2$ means that the edge root (i,j) is connected to a leaf and bud, respectively. Using this second method, the blossom tree illustrated in Figure 4.6 can be represented by the following:

$$\begin{bmatrix} -1 & -2 & (1,0) & (5,0) \\ (0,2) & (2,0) & -2 & (3,0) \\ (1,1) & -1 & -2 & -1 \\ (1,3) & (4,0) & -2 & -1 \\ (3,1) & -2 & -1 & 1 \\ (0,3) & -2 & -1 & (6,0) \\ (5,3) & -1 & -1 & -2 \end{bmatrix}$$

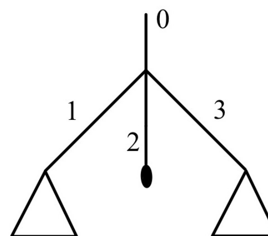


Figure 4.7: The diagram illustrates the values assigned to each of the four edges incident to an internal node.

4.8.3 Closing Blossom Trees

To close a blossom tree using the method given in Section 4.4, we create a list `fringe_nodes` containing of all the fringe nodes in depth-first order, starting from the root leaf. Whenever we see an unmatched bud followed by an unmatched leaf, we join the two nodes and remove them from `fringe_nodes`. We also update the 2-dimensional list `blossom_tree_2d` describing the blossom tree and store the closure edge information in `closure_edges`. We continue in this way until we have created exactly n closure edges, and `fringe_nodes` has two leaves left. The last step is to connect these leaves and randomly pick one of the two with equal probability to be the new root leaf.

4.8.4 Representing knot diagrams by classical Gauss Codes

To use our knot diagram generator in *Regina*, we need to encode our 2-dimensional representation of knot diagrams. There are five different codes that *Regina* accepts: Regina's own knot signature, classical and oriented Gauss codes, Dowker-Thistlethwaite codes and planar diagram codes. In particular, we use the classical Gauss codes since it is one of the most common encoding methods in the literature and the simplest to encode.

The algorithm for obtaining a classical Gauss code of a knot diagram of size n is as follows. First, we associate each crossing with a unique value between 1 and n. Then, starting from the crossing with value 1, we follow the strand, taking the path straight ahead and keeping track of the node values whenever we encounter a crossing. We store its negative value if it is an under strand, and its positive value if it is and over strand. As an example, the trefoil knot illustrated in Figure 4.8 has a Gauss code $[1,-2,3,-1,2,-3]$.

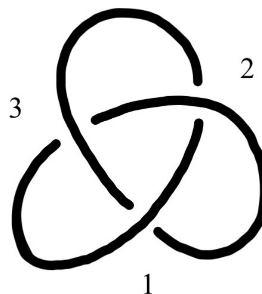


Figure 4.8: The trefoil knot.

Chapter 5

Random Sampling of Prime Knot Diagrams

To reproduce Burton and Ozlen's experiment [10] as closely as possible, we developed a prime knot diagram generator. In this chapter, we present a new theorem for checking whether a knot diagram is composite directly from a blossom tree. The full algorithm for generating random prime knot diagrams is provided in Section 5.2.

At present, there are several methods for generating prime knot diagrams. Diao et al. [16] obtain prime knot shadows from a knot shadow with n nodes by decomposing the graph using an algorithm given by Dowker and Thistlethwaite [17]. However, a limitation of this approach is that it is not guaranteed to give a prime knot shadow of size n. Furthermore, it is unclear whether we can even sample diagrams almost uniformly. Chapman takes a different approach by first generating a simple *quadrangulation* using Brinkmann and McKay's `plantri` [3] [7]. A *quadrangulation* is a planar map in which every face is bounded by exactly four edges. Chapman then obtains its *dual* graph by swapping the roles of vertices and faces and uses the fact that the dual of a simple quadrangulation is equivalent to a prime link shadow. An example of a dual graph of a quadrangulation is given in Figure 5.1. The white circles and the dashed lines represent the vertices and edges of the quadrangulation, respectively.

We present a new approach to generating prime knot diagrams. An advantage of our method is that we can make use of blossom trees and the already created knot diagram generator without building prime knot diagrams from scratch. The main idea is to use the properties of blossom trees to find two distinct edges x and y whose removal disconnects the given knot shadow. If no such edges exist, then it is prime.

We first give a brief summary of our algorithm for sampling a prime knot diagram of

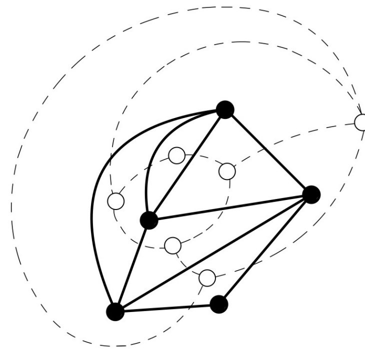


Figure 5.1: The dual graph of a quadrangulation.

size n. The full details are given in Section 5.2. The algorithm is as follows. 1) Obtain a knot shadow from a balanced blossom tree using the method described in Chapter 4. 2) Find all possible *well-bracketed sequences* s and t. 3) For each bud, search for a *2-edge cut* (x, y) by considering the following three cases: only s is non-empty, only t is non-empty, and s and t are both non-empty. 5) If x and y are found, then the knot shadow is composite. Repeat steps 1 to 4 until we find a prime knot shadow. 6) Convert the prime knot shadow into a prime knot diagram by adding the crossing information to each node.

5.1 Theorem for recognising composite knot shadows

In this section, we present a new theorem for checking whether a link shadow is composite. We first define what it means for a balanced blossom tree to have a *splitting*.

Definition 5. *(Splitting). Let B be a balanced blossom tree. B has a splitting if there exists a subtree X of B and a proper subtree Y of X such that the strings s and t that form the proper fringe of $X - Y$ are both well-bracketed.*

Remark. Y may consist of a single bud or a leaf. Furthermore, s or t may be empty but cannot both be empty, since then $X = Y$.

Figure 5.2 illustrates the three different ways in which we can have X and Y. We use x and y to denote the edges incident to the roots of the subtrees X and Y, respectively. These edges are represented by the thick edges above each subtree.

We are now ready to prove the following theorem which allows us to check if a given link shadow is composite. The proof is based on an initial idea and a sketch proof provided by John Longley.

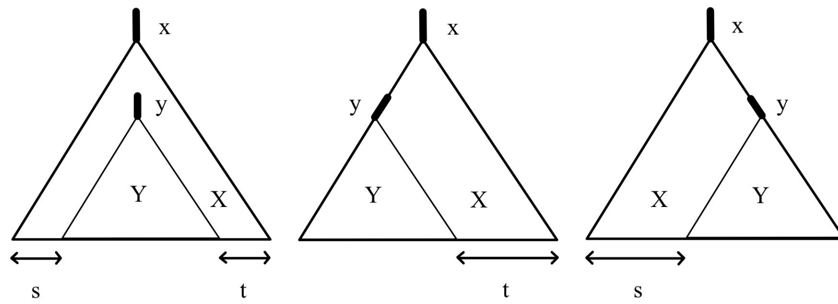


Figure 5.2: Three different ways in which we can have X and Y.

Theorem 2. *Let B be a balanced blossom tree. The closure of B, G(B), is composite if and only if B has a splitting.*

Proof. Suppose B has a splitting with a subtree X of B and a proper subtree Y of X. Let $x \in B - X$ and $y \in B - Y$ be the edges incident to the root of X and Y, respectively. Then removing x and y would disconnect $X - Y$ from $G(B)$, since s and t are well-bracketed.

In the opposite direction, suppose $G(B)$ is composite. Then there exists a pair of distinct edges $x, y \in G(B)$ such that removing x and y disconnects $G(B)$. Let $G'(B)$ be the graph that results from removing x and y from $G(B)$. First, note that every edge in $G(B)$ is either an edge in B called the *tree edge*, or a closure edge between two fringe nodes, as shown in Figure 5.3. However, a closure edge can be replaced by one of the two tree edges incident to its end vertices. Thus, we can assume that both x and y are tree edges. Now, let X and Y be the subtrees connected to x and y, respectively. There are two cases to consider: y is within X, or y lies outside X.

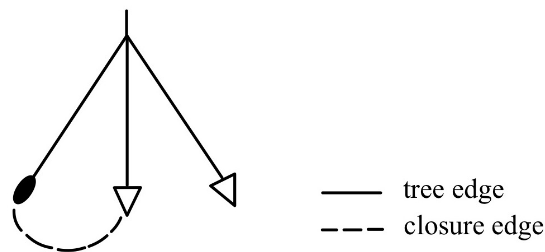


Figure 5.3: Tree edges and closure edges of a part of a tree.

Case 1: Suppose first that y lies in X. Clearly, Y is a proper subtree of X since x and y are distinct. Moreover, there cannot be any closure edge connecting $X - Y$ and $B - X - Y$ since removing x and y disconnects $G(B)$. Therefore, if s is non-empty, then it must be well-bracketed. The same holds for t. Hence, B has a splitting.

Case 2: Suppose now that y lies outside X. Since x and y are distinct, $X \cup Y$ is empty. There are three subcases to consider: both X and Y consist of a single fringe node only, Y is a single fringe node and X contains more than one node, and both X and Y consist of more than one node.

Subcase 2.1: Suppose both X and Y consist of a single fringe node only. If these two nodes are not connected by a closure edge, then clearly, $G'(B)$ is connected since both nodes must be attached to some fringes nodes in $B - X - Y$. If the two nodes are joined by a closure edge, then x and y represent the same edge so $G'(B)$ is still connected.

Subcase 2.2: Suppose Y is a single fringe node, and X contains more than one node. If Y is connected to a node outside X by a closure edge, then X must have at least one leaf that is connected to a node in $B - X - Y$, since every blossom tree has more leaves than buds (See Figure 5.4). Note that since B is balanced, there cannot be any closure edge between two leaves that do not contain the root leaf. If Y is connected to a node in X, then we can replace y by y' and Y by Y', where y' is the edge in X that is connected to y by a closure edge, and Y' is the subtree that is connected to y' (See Figure 5.5). Since y' is contained in X, this case is equivalent to *case 1*.

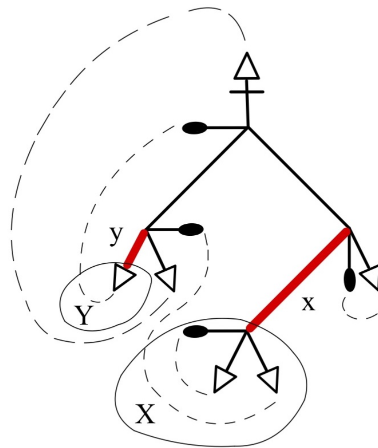


Figure 5.4: Y connected to a node outside X by a closure edge.

Subcase 2.3: Lastly, suppose both X and Y consist of more than one node. Then each of these subtrees have one extra leaf which must be connected to the root leaf or a bud in $B - X - Y$ (See Figure 5.6). As aforementioned, these two leaves cannot be connected by a closure edge, since B is balanced. Therefore, $G'(B)$ must be connected.

Thus, the only possibility is *case 1*. Y must be a proper subtree of X, and s and t that form the proper fringe of $X - Y$ must be well-bracketed.

□

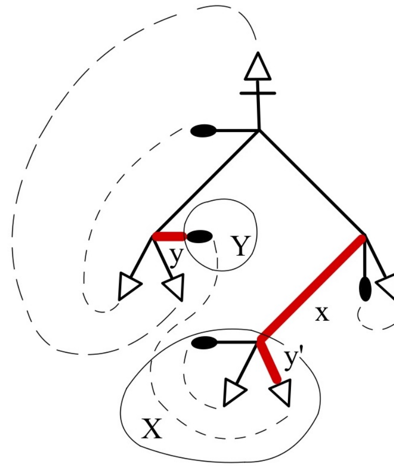


Figure 5.5: Y connected to a node in X by a closure edge.

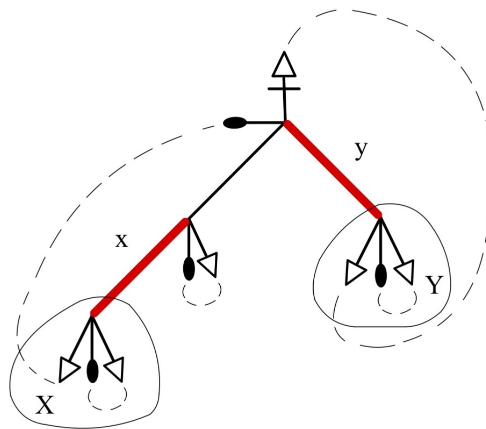


Figure 5.6: Both X and Y consist of more than one node.

Note that if we redefine the term *splitting* by also allowing s and t to include the last closure edge between the last two remaining leaves, then the theorem holds for any blossom tree. An example of this case is illustrated in Figure 5.7. The subtree Y is the single bud inside X. We can observe that s contains the last closure edge, and t is empty. The graph is composite since removing the edges x and y disconnect the graph. The proof remains the same, except we need an additional argument to show that y cannot lie outside X: if the last closure edge lies within the tree X or Y, then there is always a bud inside the tree that is connected to a leaf outside it.

For the rest of the paper, we use our first definition of *splitting* for two reasons. Firstly, if s and t are allowed to be well-bracketed sequences only, then we only need to look at the n buds compared to the $n+1$ closure edges when using the second definition. Secondly, it turns out that we can efficiently find the edge x if we know the path to a

bud. We present the following lemma to prove this fact.

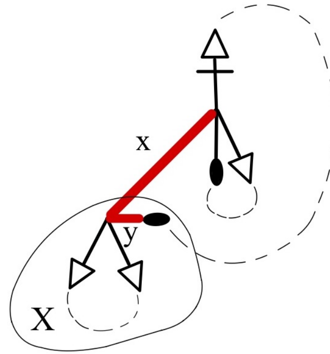


Figure 5.7: An unbalanced blossom tree with the last closure edge inside X.

Lemma 1. *Suppose a balanced blossom tree B has a splitting with subtree X of B and a proper subtree Y of X. Let x and y be the corresponding edges that disconnect the graph, and s be the usual well-bracketed sequence that lies within X. If s is non-empty, then the left child of the root of X is a single bud.*

Proof. Since s is non-empty and well-bracketed, the leftmost fringe node of X must be a bud. Let r denote the root node of X. Aiming for a contradiction, suppose that the left child of r has children. Then the left child of r together with its children is itself a tree. Any subtree of a blossom tree contains exactly one more leaf than buds, so s is not well-bracketed. □

Remark. The above lemma holds only when the blossom tree is balanced.

5.2 Algorithm for Sampling Prime Knot Diagrams

We now present our algorithm for generating random prime knot diagrams directly from a balanced blossom tree.

5.2.1 Generating knot shadows

The first step is to generate a knot shadow by constructing a balanced blossom tree and obtaining its closure using the algorithm described in Chapter 4. We also require a list of *addresses* of all the fringe nodes. An *address* of a node k is the path from the root node to k. It can be described by a string of three letters 'l', 'm' and 'r', representing left, middle and right child nodes, respectively. The addresses are obtained by a depth-first traversal of the tree.

5.2.2 Finding possible well-bracketed sequences s and t

The next step is to search for all possible well-bracketed sequences s and t. Recall that a blossom tree of n internal nodes has precisely n buds, and every well-bracketed sequence starts with a bud. Therefore, we only need to look at the n buds in turn and find all possible well-bracketed sequences that start from that bud. This is more efficient than considering all pairs of edges x and y, and then searching for the well-bracketed sequences. Most importantly, Lemma 1 shows that if s is non-empty, then the left child of the root of a subtree X is always a single bud, and so we can easily identify the unique edge x if we know the path to that bud.

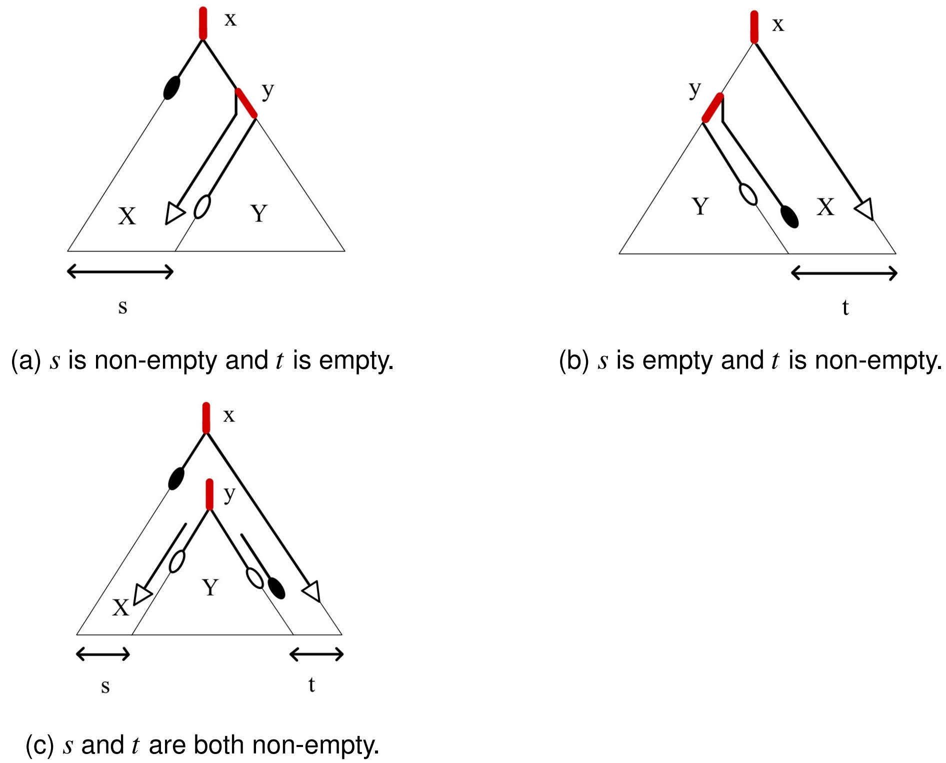
There are several requirements for s and t to be one of the possible sequences. The first condition is that both sequences must be well-bracketed. This implies that a valid sequence represents either a closure edge or a set of consecutive closure edges. The second condition is that if s is non-empty, then the path from x to the first node of s must always follow the left path since X is a tree. Similarly, if t is non-empty, then the path from x to the last leaf of t must always follow the right path. A pseudo-code for finding all possible s and t is given in Algorithm 1.

5.2.3 Finding a 2-edge cut (x, y)

The third step is to search for a pair of edges (x, y) whose removal disconnects the graph. Such a pair is called a *2-edge cut*. To find these edges, we look at each bud in turn and consider the following three cases: only s is non-empty, only t is non-empty, and s and t are both non-empty. Let $a_{bud} = b_1 b_2 ... b_m$ be the address of the bud, where m is some non-negative integer and each b_i represents either 'l', 'm' or 'r'.

5.2.3.1 s is non-empty and t is empty

Suppose s is non-empty but t is empty. Then using Lemma 1, the address of x is $\alpha(x) = b_1 b_2 ... b_{m-1}$. To find y, we look at the addresses of the last leaf of s and the fringe node immediately next to it. Let these two addresses be denoted by a_1 and a_2, respectively. Since both X and Y are trees, we must have $a_1 = \alpha(x) r_k m r_p$ and $a_2 = \alpha(x) r_{k+1} l_q$, for some non-negative integers k, p and q. If both conditions are satisfied, then the address of y is given by $\alpha(y) = \alpha(x) r_{k+1}$, and the pair ($x$, y) is a 2-edge cut. A diagram illustrating the first case is depicted in Figure 5.8a. We use the white oval-shaped nodes to describe a fringe node that could be a leaf or a bud.



(a) s is non-empty and t is empty.

(b) s is empty and t is non-empty.

(c) s and t are both non-empty.

Figure 5.8: Diagrams illustrating the three possible cases.

5.2.3.2 s is empty and t is non-empty

If s is empty and t is non-empty, then we cannot use Lemma 1 to find x. Instead, we first find the edge y. Let a_1 be the address of the fringe node immediately before the bud. If there is a path $y_1y_2...y_k$ such that $a_1 = y_1y_2...y_kr_p$ and $a_{bud} = y_1y_2...y_{k-1}mr_q$ for some non-negative integers k, p and q, then y is the edge with the address $y_1y_2...y_k$.

Suppose we found y. Our next step is to search for x. Let a_3 be the address of the last leaf of t. Since X is a tree, x must be an edge such that $\alpha(y) = \alpha(x)l_p$ and $a_3 = \alpha(x)r_q$ for some non-negative integers p and q. If both x and y are found, and both X and Y are trees, then the pair (x, y) is a 2-edge cut.

5.2.3.3 Both s and t are non-empty

Suppose now that both sequences are non-empty. Then using Lemma 1, the address of x is $\alpha(x) = b_1b_2...b_{m-1}$. We require four addresses to find y: the last leaf of s, the fringe node immediately next it, the first bud of t and the fringe node immediately before it, denoted by a_1, a_2, a_3 and a_4, respectively. y is an edge such that $a_2 = \alpha(y)l_p$ and $a_4 = \alpha(y)r_q$ for some positive integers p and q. Furthermore, the address of y must be

strictly longer than that of x since Y is a proper subtree of X. If both x and y are found, and both X and Y are trees, then the pair (x, y) is a 2-edge cut.

Algorithm 1 Finding possible well-bracketed sequences s and t

```
for each bud i do
    bud1_add ← address of bud i
    c1 ← closure edge from bud i
    c1_last_leaf_add ← address of the last leaf of c1
    if the last string of bud1_add is 'l' then
        c1 is a possible s
    end if
    if the last string of c1_last_leaf_add is 'r' then
        c1 is a possible t
    end if
    for each bud j = i+1 to total number of buds do
        bud2_add ← address of bud j
        c2 ← closure edge from bud j
        if c2 comes immediately after c1 & the last leaf of c2 is not the root leaf then
            c2_last_leaf_add ← address of the last leaf of c2
            if the last string of bud2_add is 'l' then
                c3 = c1 + c2 is a possible s
            end if
            if the last string of c2_last_leaf_add is 'r' then
                c3 = c1 + c2 is a possible t
            end if
        end if
    end for
end for
```

Chapter 6

Experimental Results and Discussions

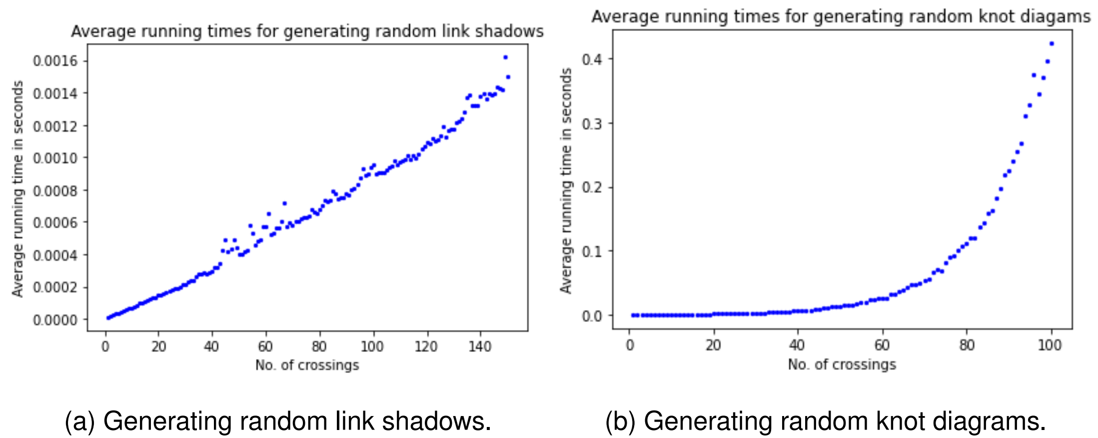
6.1 Almost uniform knot diagram generators

The knot diagram sampling algorithms studied in Chapter 4 and Chapter 5 were implemented in Python to be used for evaluating various knot algorithms in *Regina*. In this section, we consider the correctness and the efficiency of the generators.

6.1.1 Correctness

Several approaches were taken to ensure the correctness of the general knot diagram generator. First, we generated samples of relatively large knot diagrams up to $n = 20$ crossings and checked by hand that the outputs represented 4-valent planar maps of size n. To test whether a link shadow is a valid knot shadow, we followed the strand, taking the path straight ahead at each vertex and checking that we passed through all the edges. We were also able to test its correctness in *Regina* since it raises an error if a Gauss code is not in the correct format or does not represent a knot diagram. No errors were raised in all the samples and experiments we conducted.

In contrast, testing the correctness of the prime knot diagram generator was far more challenging. It is easy to verify that two edges disconnect a graph if we are provided with the edges. However, proving that a knot diagram is prime requires one to look at all combinations of edges and show that they do not disconnect the graph. Besides checking the correctness by hand, one effective test was to use the fact that a balanced blossom tree can be presented in two different ways, depending on which of the last two remaining leaves we choose to be the new root. We generated a large sample of prime knot diagrams of sizes ranging from $n = 3$ to $n = 12$, and for each sample, we made



(a) Generating random link shadows.

(b) Generating random knot diagrams.

Figure 6.1: Average running times for generating random link shadows and knot diagrams from the general knot diagram generator.

sure that these two *conjugate* blossom tree representations outputted identical results. Of course, there are more reliable and perhaps efficient methods to test its correctness than by hand. An inefficient but simplest method would be a brute-force search of all pairs of edges and checking whether they disconnect the graph. If no such edges exist, then the diagram is prime.

6.1.2 Efficiency

We measured the efficiency of the generators by obtaining the experimental running times. In the first experiment, we sampled 2000 link shadows of each size between $n = 1$ and $n = 150$. The average running times are shown in Figure 6.1a. The graph suggests that the running time increases linearly with the number of crossings in the diagram. However, this may not remain true in the limit, since there is a very slight hint of curvature. Nevertheless, all the steps from sampling a binary tree to creating a blossom tree and closing the tree can be done immediately, even for large diagrams. For example, on average, it takes just 0.0015s to sample a link shadow of $n = 150$.

In the second experiment, we generated 1000 general knot diagrams of each size between $n = 1$ and $n = 100$. The smaller sample size compared to the first experiment is due to the difficulty in generating knot diagrams. Figure 6.1b describing the average running times displays a clear exponential curve. The result can be explained by the fact that the probability of a random link shadow representing a knot shadow decreases exponentially with size. This is evident in Figure 6.2a which shows the average number of link shadows that were rejected per sample. The result is also in line with Chapman's

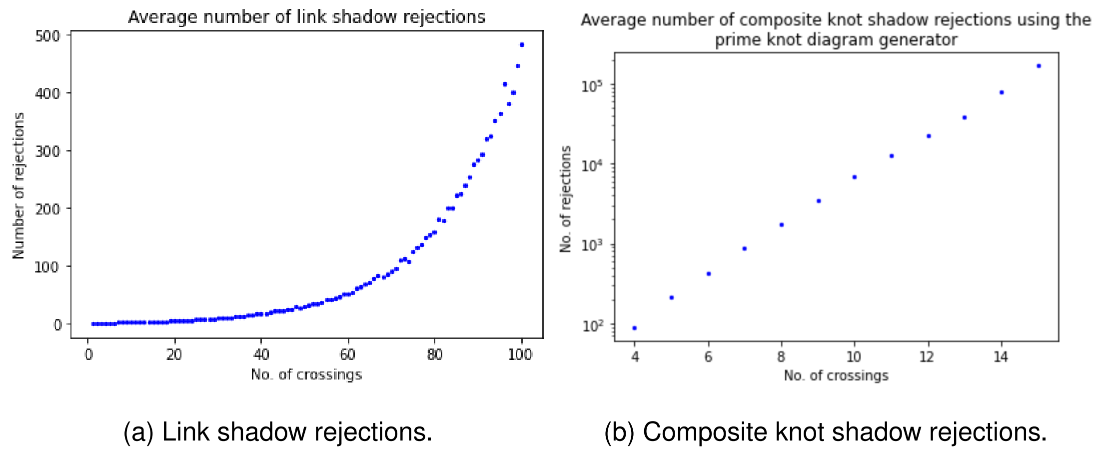
result [12]. Even so, the generator can successfully find a valid knot shadow extremely fast. For example, a knot diagram of $n = 100$ can be attained in approximately 0.42s on average. This is a remarkable result since it implies we can carry out a variety of experiments in knot theory on knot diagrams as large 150 crossings very efficiently.

In the third experiment, we generated approximately 80 prime knot diagrams of each size between $n = 4$ and $n = 15$. The smaller knot size is due to the difficulty in finding knot shadows that are prime. Checking whether a given knot shadow is prime took less than one second in all the samples, even with the prime knot shadows, where our generator had to conclusively verify that no 2-edge cuts exit. This indicates that the extreme running times were mainly due to the number of composite knot shadow rejections. This can be observed from Figure 6.2b. The y-axis is plotted on a logarithmic scale. Note that we obtained the number of composite knot shadow rejections rather than the running times to gain a better understanding of the difficulty in obtaining prime knot diagrams. The graph reveals that the average number of composite knot shadow rejections required to obtain a prime knot shadow increases exponentially with the number of crossings. Furthermore, the growth rate is much higher than that of Figure 6.2a. In other words, it is much more challenging to find a prime knot shadow than it is to find a knot shadow. Knot diagrams as small as 12 crossings required approximately 22,858 knot shadow rejections before obtaining a prime knot shadow, which reflects the difficulty in obtaining a prime diagram. As an initial exploration, we also obtained a prime knot diagram of size 20 and found that it required approximately 2 million rejections.

The result clearly shows that it is extremely difficult to generate prime knot diagrams, and therefore, our prime knot diagram generator is less useful for studying and evaluating knot algorithms. One possible solution is to use the method provided by Diao et al. [16], which reduces the given knot shadow of size n into smaller prime knot shadows. Although this does not guarantee a prime knot shadow of n crossings, we can at least obtain prime knot shadows from every knot shadow without wasting it.

6.2 Unknot recognition algorithm using normal surface theory

The next core part of the project was to use our knot diagram generators to evaluate the unknot recognition algorithm developed by Burton and Ozlen [10]. We first provide the



(a) Link shadow rejections.

(b) Composite knot shadow rejections.

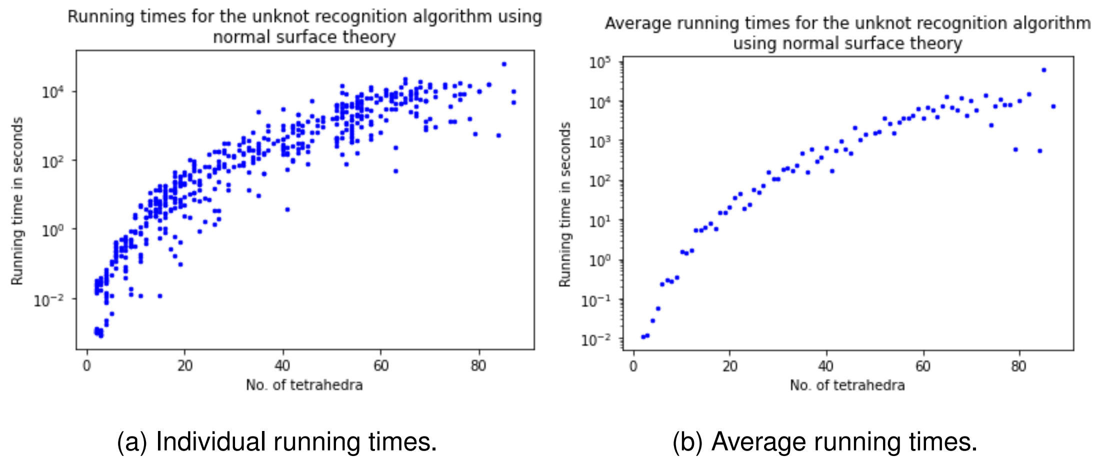
Figure 6.2: Average number of rejections per sample.

results obtained using the general knot diagram generator. Then, in Subsection 6.2.2, we discuss the results from the prime knot diagram generator.

6.2.1 Experimental results using the general knot diagram generator

As an initial experiment, we tested the algorithm on only the trivial knots. We took 3000 samples of knot diagrams of various sizes up to 100 crossings and found that all except a very few had either 2 or 3 tetrahedra. The maximum number of tetrahedra we observed was 4. The small number of tetrahedra meant each knot took less than one second to prove that it is equivalent to the unknot. This was as expected since the unknotting problem lies in the complexity class NP and we can verify that a knot is the unknot in polynomial time [22].

In the next experiment, we tested the algorithm on only the non-trivial knots. As previously mentioned in Section 3.1, a limitation of Burton and Ozlen's experiment is that they only investigated knots up to 50 tetrahedra. The problem was overcome by sampling large knot diagrams from our generator. Firstly, to test how large a number of tetrahedra we can generate, we sampled 400 knot diagrams of 150 crossings. Figure 6.4 describes the frequency of the numbers of tetrahedra. The results indicate that there were 193 knot diagrams with over 50 tetrahedra and 11 diagrams with over 100 tetrahedra, with the largest being 143. This is nearly three times the maximum number of tetrahedra obtained by Burton and Ozlen. Furthermore, all the samples required less than one second to find a triangulation and compute the number of tetrahedra. However, it was computationally infeasible to test Burton and Ozlen's unknot recognition algorithm on



(a) Individual running times.

(b) Average running times.

Figure 6.3: Individual and average running times for the unknot recognition algorithm using normal surface theory.

knot diagrams with greater than 90 tetrahedra. The largest sample we were able to test was 87. This can be explained by the fact obtaining the complement and a triangulation of a knot can be done in small polynomial time, but the rest is exponential time in theory [10].

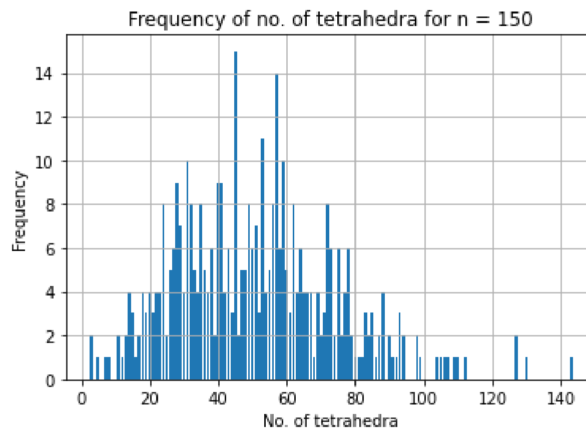
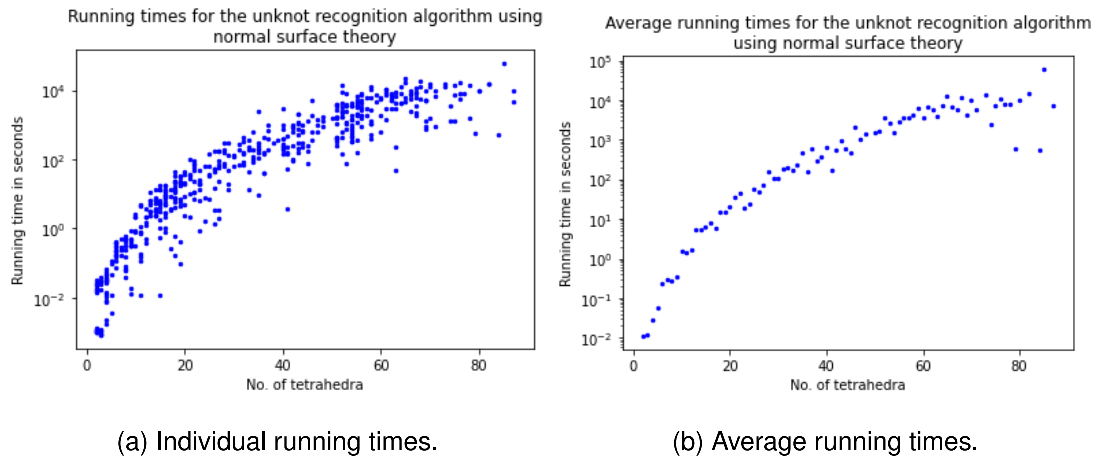


Figure 6.4: Frequency of the numbers of tetrahedra for n = 150 (sample size = 400).

The aim of the next experiment was to reproduce and extend the experiment done by Burton and Ozlen. Their data set consisted of only the prime knots, whereas our experiment consisted of general non-trivial knot diagrams. Nevertheless, our results are consistent with theirs and show a polynomial-time behaviour up to 50 tetrahedra. This can be seen in Figure 6.5. Note that the y-axes are plotted on a logarithmic scale. We can also see that the polynomial time behaviour continues from 50 up to 87 tetrahedra. Figure 6.5b describing the average running times has several outliers around



(a) Individual running times.

(b) Average running times.

Figure 6.5: Individual and average running times for the unknot recognition algorithm using normal surface theory.

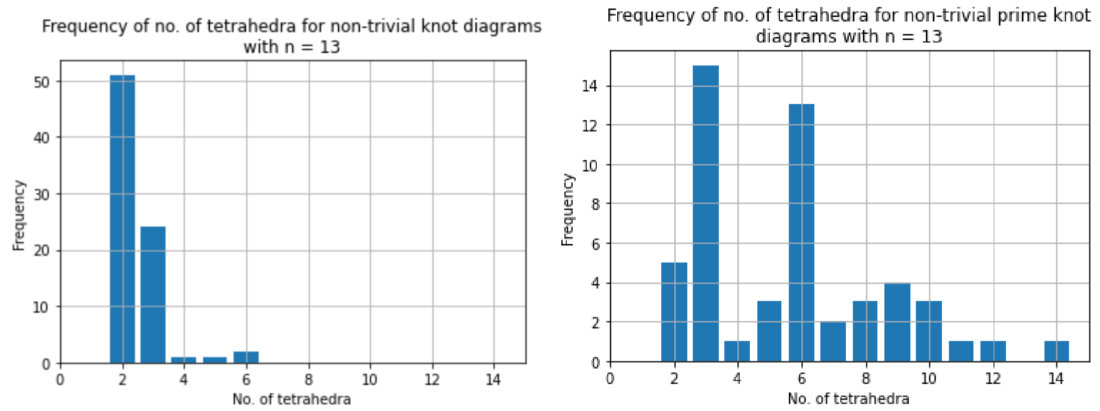
80 tetrahedra. This is due to the small sample size for larger knot diagrams. Figure 6.5a also shows a high variability in the running times. For example, a knot diagram with 69 tetrahedra had a running time of 5226s, but another sample of the same number of tetrahedra took just 774s to solve. Further study of the algorithm is required to explain these differences in the results. Nevertheless, the results clearly indicate a polynomial time behaviour at least up to around 80 tetrahedra.

6.2.2 Experimental results using the prime knot diagram generator

In this section, we evaluate Burton and Ozlen's unknot recognition algorithm using our prime knot diagram generator. Our aim was to reproduce their experiment as closely as possible by testing on prime knot diagrams.

Firstly, we generated approximately 80 prime knot diagrams of each size between $n = 4$ and $n = 15$, and computed the number of tetrahedra of each knot diagram. The results were striking. Although our sample size was rather small, we found that we could generate a larger number of tetrahedra using prime knot diagrams compared to general knot diagrams of the same size. As an example, Figure 6.6 shows the frequency of the numbers of tetrahedra we observed for $n = 13$. Clearly, there is a wider variety of numbers of tetrahedra in Figure 6.6b than in Figure 6.6a. Furthermore, we obtained a maximum of 14 tetrahedra with non-trivial prime knot diagrams, compared to just 6 using general non-trivial knot diagrams. Similar results were observed for all the sizes n.

As aforementioned, Burton and Ozlen tested on all 2977 prime knots up to 12



(a) Non-trivial general knot diagrams with $n = 13$ (sample size = 79).

(b) Non-trivial prime knot diagrams with $n = 13$ (sample size = 79).

Figure 6.6: Frequency of the numbers of tetrahedra for non-trivial general and prime knot diagrams for $n = 13$.

crossings. As an experiment, we generated 3000 random non-trivial knot diagrams of 12 crossings. However, we could not find any knot diagrams as close to 50 tetrahedra. In fact, the maximum we obtained was just 8, and the majority had either 2 or 3 tetrahedra. This suggests that there may be a property of prime knots that causes them to have a larger number of tetrahedra. This could also suggest the reason why we found slightly larger tetrahedra with prime knot diagrams.

In the next experiment, we tested the algorithm using our generated prime knot diagrams. The empirical running times were very similar to that obtained using the general knot diagram generator. Unfortunately, we were not able to fully reproduce Burton and Ozlen's experiment due to the difficulty in obtaining prime knot diagrams. However, the results suggest that prime knots and prime knot diagrams may have a larger number of tetrahedra compared to general knot diagrams. This is an interesting topic for future work.

6.3 Reidemeister Moves

As we discussed in Section 3.2, a given knot diagram represents the unknot if and only if it is related to the unknot by a sequence of Reidemeister moves. However, it cannot be used to prove that a given knot is a non-trivial knot, since the program will not terminate. Hence, we tested the algorithm on knots equivalent to the trivial knots and found the empirical running times for transforming the knots into the unknot. The

results are shown in Figure 6.7. The data set consists of 400 samples of trivial knot diagrams up to 100 crossings. Clearly, it becomes difficult to generate large trivial knot diagrams as the size increases. The graph shows that turning the knot into the unknot can be done extremely fast. This was as expected since the unknotting problem lies in the complexity class NP [22].

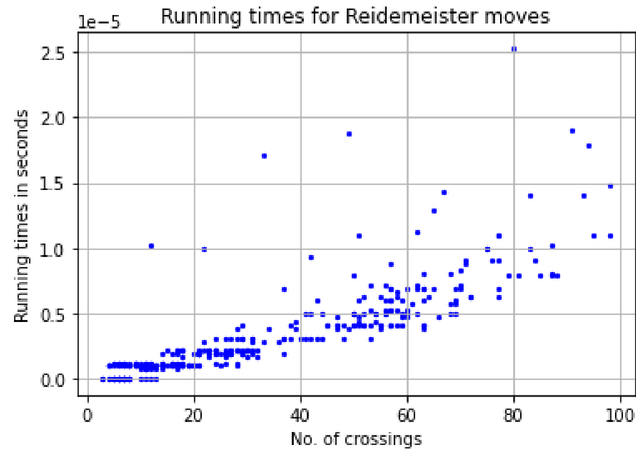


Figure 6.7: Running times for the experiment using Reidemeister moves.

6.4 Polynomial Knot Invariants

The final part of the experiments investigated the polynomial invariants Jones and HOMFLY polynomials. Prior to carrying out both experiments, we generated a large sample of non-trivial knot diagrams up to $n = 150$. This was achieved using Burton and Ozlen's unknot recognition algorithm. We then used these knot diagrams to obtain the empirical running times for computing the Jones and HOMFLY polynomials.

6.4.1 Jones Polynomial

Figure 6.8 shows the individual running times for computing the Jones polynomial. Clearly, computing the Jones polynomial is much faster than the algorithm using normal surface theory. For example, finding the Jones polynomial of a knot diagram of $n = 141$ took approximately 0.0060s, on average. As we discussed in Section 3.3, it has been verified up to 24 crossings that there are no non-trivial knots with Jones polynomial equal to 1 [32]. Therefore, up to 24 crossings, computing the Jones polynomial is a much faster method for unknot recognition. During the experiment, we also looked for

counterexamples of the Jones conjecture by testing on large knot diagrams. However, of the 10000 samples we tested, none were found.

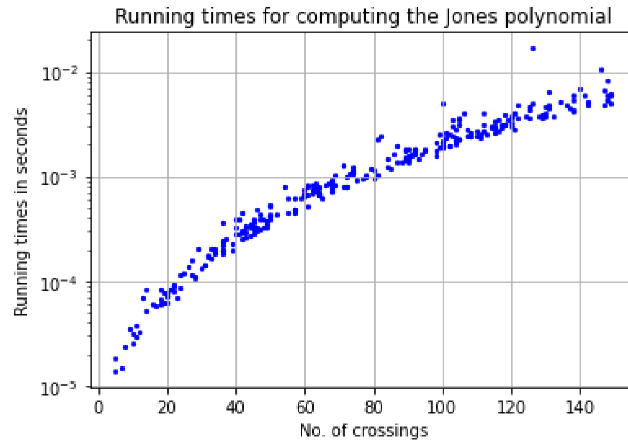


Figure 6.8: Individual running times for computing the Jones polynomial.

6.4.2 HOMFLY Polynomial

Regina can also compute HOMFLY polynomials. As with the Jones polynomial, computing the HOMFLY polynomial can be done very quickly. For example, finding the HOMFLY polynomial of a knot diagram of $n = 141$ took approximately 0.0145s, on average. However, the graph shows that HOMFLY polynomials take slightly longer to compute than Jones polynomials. This is clear since HOMFLY polynomial is a generalised version of Jones polynomials and requires working with two variables instead of one.

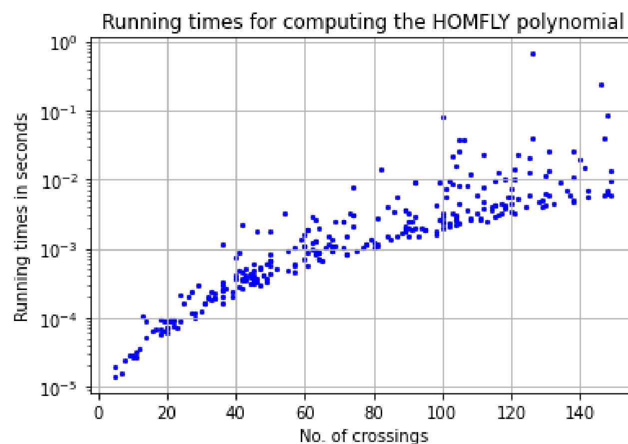


Figure 6.9: Individual running times for computing the HOMFLY polynomial.

Chapter 7

Conclusions

The aim of this project was to empirically evaluate *Regina*'s various knot algorithms, with the main focus being Burton and Ozlen's unknot recognition algorithm using normal surface theory. To achieve this, we used the algorithms provided by Chapman and Schaeffer to develop an almost uniform knot diagram generator. Furthermore, we presented a new theorem and an algorithm for testing whether a knot diagram is prime. The main advantage of our method was that we could make use of blossom trees and the already created knot diagram generator without building prime knot diagrams from scratch. Both the general and prime knot diagram generators were implemented in Python. Our experiments showed that we could sample general knot diagrams very efficiently, even with knot diagrams as large as 150 crossings.

Using the knot diagram generator, we reproduced and extended Burton and Ozlen's experiment. We found that the polynomial-time behaviour continues above 50 tetrahedra, even though the algorithm is exponential time in theory. However, it was computationally infeasible to test on knot diagrams with greater than 90 tetrahedra.

Our aim was also to use the prime knot diagram generator to reproduce their experiment more closely. However, we could only sample prime knot diagrams up to 15 crossings due to the difficulty in finding knot shadows that were prime. Nevertheless, the experiment suggested that prime knot diagrams have a slightly higher number of tetrahedra compared to general knot diagrams of the same size. We believe that there are some properties of prime knots and prime knot diagrams that cause a higher number of tetrahedra. This is an interesting topic for future work.

Lastly, we used the knot diagram generator to obtain empirical running times for simplifying knots into the unknot using Reidemeister moves and computing the polynomial knot invariants Jones and HOMFLY polynomials. We tested on knot diagrams

up to 100 crossings and observed that all three algorithms could be computed much faster than the unknot recognition algorithm using normal surface theory. However, they cannot be used as an unknot recognition algorithm on their own.

There are several areas for further research. Firstly, we were unable to test the correctness of the prime knot diagram generator on large samples. One possible method would be a brute-force search of all pairs of edges and checking whether they disconnect the graph. Secondly, we could follow the method provided by Diao et al. [16], which reduces a given knot shadow of size n into smaller prime knot shadows. Although this does not guarantee a prime knot shadow of n crossings, we can at least obtain prime knot shadows from every knot shadow without wasting it. Lastly, another future research could be developing a *link* diagram generator. In contrast to knots, a *link* can consist of more than one strand. The generator can then be used to carry out similar experiments in *Regina*. Note that we have to encode the links using planar diagram codes since Gauss codes are only defined for knots.

Bibliography

[1] The KnotPlot Site. Available at https://knotplot.com/. Online; accessed August 2022.

[2] Knotscape. Available at https://pzacad.pitzer.edu/ jhoste/ HosteWebPages/ kntscp.html. Online; accessed August 2022.

[3] Plantri and fullgen. Available at https://users.cecs.anu.edu.au/ bdm/ plantri/. Online; accessed August 2022.

[4] Topology and Geometry. Available at https://www.geometrygames.org/. Online; accessed August 2022.

[5] *Algorithmic Topology and Classification of 3-Manifolds*. Springer, 2007.

[6] Colin Adams. *The knot book: an elementary introduction to the mathematical theory of knots*. American Mathematical Society, 2004.

[7] G. Brinkmann and B. D. McKay. Fast generation of some classes of planar graphs. *Electronic Notes in Discrete Mathematics*, 3:28–31, 1999.

[8] Benjamin A. Burton. Computational topology with Regina: Algorithms, heuristics and implementations. Technical Report arXiv:1208.2504, arXiv, 2013. arXiv:1208.2504 [cs, math] type: article.

[9] Benjamin A. Burton, Ryan Budney, William Pettersson, et al. Regina: Software for low-dimensional topology. Available at http://regina-normal.github.io/., 1999–2021. Online; accessed June 2022.

[10] Benjamin A. Burton and Melih Ozlen. A fast branching algorithm for unknot recognition with experimental polynomial-time behaviour. *arXiv:1211.1079 [cs, math]*, October 2014.

[11] Mick Burton. How do you construct haken's gordian knot? Available at https://mickburton.co.uk/2015/06/05/how-do-you-construct-hakens-gordian-knot/. Online; accessed July 2022.

[12] Harrison Chapman. A diagrammatic theory of random knots, 2017. PhD thesis, The University of Georgia.

[13] Peter Cromwell. *Knots and Links*. Cambridge University Press, 2004.

[14] Marc Culler, Nathan M. Dunfield, Matthias Goerner, and Jeffrey R. Weeks. SnapPy, a computer program for studying the geometry and topology of 3-manifolds. Available at http://snappy.computop.org. Online; accessed August 2022.

[15] Nachum Dershowitz and Shmuel Zaks. The Cycle Lemma and Some Applications. *European Journal of Combinatorics*, 11:35–40, 1990.

[16] Yuanan Diao, Claus Ernst, and Uta Ziegler. Generating large random knot projections. In *Physical and Numerical Models in Knot Theory*, volume Volume 36 of *Series on Knots and Everything*, pages 473–494. WORLD SCIENTIFIC, 2005.

[17] C. H. Dowker and Morwen B. Thistlethwaite. Classification of knot projections. *Topology and its Applications*, 16:19–31, 1983.

[18] A. Dvoretzky and Th Motzkin. A problem of arrangements. *Duke Mathematical Journal*, 14:305–313, 1947. Publisher: Duke University Press.

[19] C. McA Gordon and J. Luecke. Knots are determined by their complements. *Journal of the American Mathematical Society*, 2:371–415, 1989.

[20] Wolfgang Haken. Theorie der Normalflächen. *Acta Mathematica*, 105(3):245–375, 1961.

[21] Wolfgang Haken. Some results on surfaces in 3-manifolds. Studies in Modern Topology. *Math. Assoc. America*, page 39–98, 1968.

[22] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger. The computational complexity of knot and link problems. *Journal of the ACM*, 46:185–211, 1999.

[23] Geoffrey Hemion. On the classification of homeomorphisms of 2-manifolds and the classification of 3-manifolds. *Acta Mathematica*, 142:123–155, 1979.

[24] Jim Hoste. The enumeration and classification of knots and links. In *Handbook of Knot Theory*, pages 209–232. Elsevier, 2005.

[25] Greg Kuperberg. Knottedness is in NP, modulo GRH. *Advances in Mathematics*, 256:493–506, 2014.

[26] Marc Lackenby. A polynomial upper bound on Reidemeister moves. Technical report, arXiv, 2014. arXiv:1302.0180 [math] type: article.

[27] Marc Lackenby. The efficient certification of knottedness and Thurston norm. *Advances in Mathematics*, 387:107796, 2021.

[28] Marc Lackenby. Unknot recognition in quasi-polynomial time. Available at `https://people.maths.ox.ac.uk/lackenby/quasipolynomial-talk-oxford-compressed.pdf`., 2021. Online; accessed June 2022.

[29] Edwin E. Moise. Affine Structures in 3-Manifolds: V. The Triangulation Theorem and Hauptvermutung. *Annals of Mathematics*, 56:96–114, 1952. Publisher: Annals of Mathematics.

[30] Kurt Reidemeister. Elementare Begründung der Knotentheorie. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 5:24–32, 1927.

[31] Gilles Schaeffer. Random sampling of large planar maps and convex polyhedra. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing - STOC '99*, pages 760–769. ACM Press, 1999.

[32] Robert E. Tuzun and Adam S. Sikora. Verification Of The Jones Unknot Conjecture Up To 24 Crossings. 2020.

[33] Jeff Weeks. Chapter 10 - Computation of Hyperbolic Structures in Knot Theory. In William Menasco and Morwen Thistlethwaite, editors, *Handbook of Knot Theory*, pages 461–480. Elsevier Science, 2005.