2017

The programmatic manipulation of planar diagram codes to find an upper bound on the bridge index of prime knots

Genevieve R. Johnson
University of Northern Iowa

THE PROGRAMMATIC MANIPULATION OF PLANAR DIAGRAM CODES TO FIND AN UPPER BOUND ON THE BRIDGE INDEX OF PRIME KNOTS

An Abstract of a Thesis
Submitted
in Partial Fulfillment
of the Requirement for the Degree
Master of Arts

Genevieve R. Johnson
University of Northern Iowa
December 2017

ABSTRACT

The “bridge index” of a knot is the least number of maximal overpasses taken over all diagrams of the knot. A naïve method to determine the bridge index of a knot is to perform Reidemeister moves on diagrams of the knot, and this method quickly becomes tedious to implement by hand. In this paper, we introduce a sequence of Reidemeister moves which we call a “drag the underpass” move and prove how planar diagram codes change as Reidemeister moves are performed. We then use these results to programatically perform Reidemeister moves using Python 2.7 to calculate an upper bound on the bridge index of prime knots with three through twelve crossings. We conclude with discussions of how our results compare to the literature and future work related to these calculations.

THE PROGRAMMATIC MANIPULATION OF PLANAR DIAGRAM CODES TO FIND AN UPPER BOUND ON THE BRIDGE INDEX OF PRIME KNOTS

A Thesis
Submitted
in Partial Fulfillment
of the Requirement for the Degree
Master of Arts

Genevieve R. Johnson
University of Northern Iowa
December 2017

This Study by: Genevieve R. Johnson

Entitled: THE PROGRAMMATIC MANIPULATION OF PLANAR DIAGRAM CODES TO FIND AN UPPER BOUND ON THE BRIDGE INDEX OF PRIME KNOTS

Has been approved as meeting the thesis requirement for the Degree of Master of Arts.

Date	Dr. Theron J. Hitchman, Chair, Thesis Committee
Date	Dr. Adrienne M. Stanley, Thesis Committee Member
Date	Dr. Bill Wood, Thesis Committee Member
Date	Dr. Patrick Pease, Interim Dean, Graduate College

To my husband Cody. Thank you for your unconditional support and being the best rubber duck a developer could ask for.

ACKNOWLEDGEMENTS

Dr. Theron Hitchman, thank you for nurturing my love of mathemtics. I will always be grateful to you for your support and kindness.

Dr. Adrienne Stanley and Dr. Bill Wood, thank you for serving on my thesis review committee. I greatly appreciate your support.

Dr. Radmila Sazdanovic, thank you for providing clarity when planar diagram codes did not obviously match diagrams. The knowledge you shared helped tremendously.

Cale Beirman and Chris Dibbern, thank you for performing code reviews. Your feedback on my programming is much appreciated.

A special thank you to my family. Mom, Dad, Charlene, Bennett, John, Cecelia, Rosalyn, Carl, and Mariel, your unwavering support means the world to me.

And to my husband, Cody. I am forever thankful for your patience and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 What is a knot?

Formally, a **knot** is a simple closed polygonal curve in $\mathbb{R}^3$. However, knots are usually thought of and drawn as smooth curves. [6]

The simplest knot is the unknotted circle, pictured in Figure 1.1, which is called the **unknot** or the **trivial knot**.

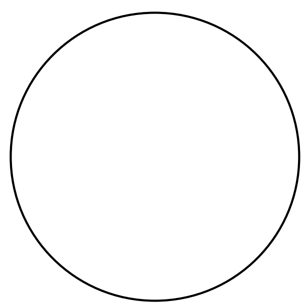


Figure 1.1: A diagram of the unknot

Common methods for depicting knots are regular projections and diagrams. In a **regular projection**, a knot is mapped from $\mathbb{R}^3$ onto a plane such that no three points on the knot project to the same point and no vertex projects to the same point as any other point on the knot [6]. Sections of a knot which pass over or under each other in $\mathbb{R}^3$ are depicted as double points in a regular projection. In a knot **diagram**, the double points of a projection are called **crossings**. Corresponding to each crossing of a diagram, in $\mathbb{R}^3$ there are two sections of the knot called an overpass and an underpass. The underpass is broken in diagrams to indicate which section passes over the other in $\mathbb{R}^3$.

Figure 1.2: A projection (left) and a diagram (right) of the trefoil

A **labeled diagram** is a knot diagram for which the edges are sequentially labeled with natural numbers. To create a labeled diagram:

1. Select a starting point on an edge of the diagram.
2. Label the edge containing the starting point 1.
3. Unless already specified, select a direction to travel the diagram. This direction is the called the **orientation** of the knot.
4. From the starting point, traverse the diagram following the orientation of the knot and label each edge with the natural number one greater than the label of the preceeding edge until all of the edges have been labeled.

Note that a diagram of a knot can have different labelings depending on the starting point selected and the orientation of the knot, as illuatrated in Figure 1.3.

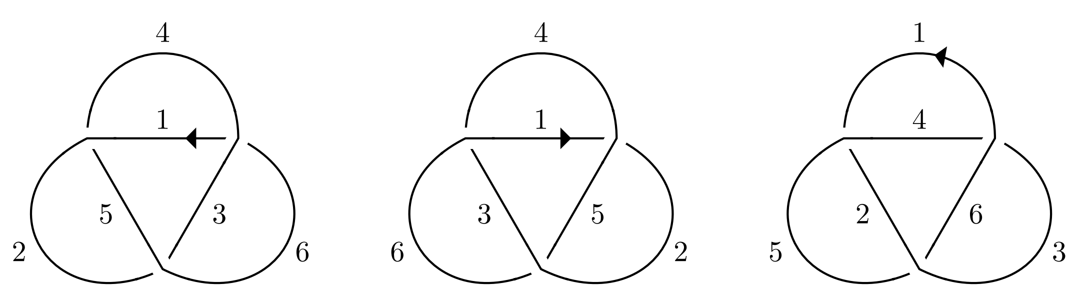


Figure 1.3: Three labelings of a diagram of the trefoil

1.2 The bridge index

In a knot diagram, an **underpass** is a section of the knot which goes under at least one crossing and does not go over a crossing. Conversely, an **overpass** is a section of the knot which goes over at least one crossing and does not go under a crossing. A **bridge**, or "maximal overpass", is an overpass which cannot be made any longer - going further from either end would result in traveling under a crossing. An underpass, overpass and bridge are illustrated by the red sections in Figure 1.4. Note that each crossing in a knot diagram must have some bridge that crosses over it.

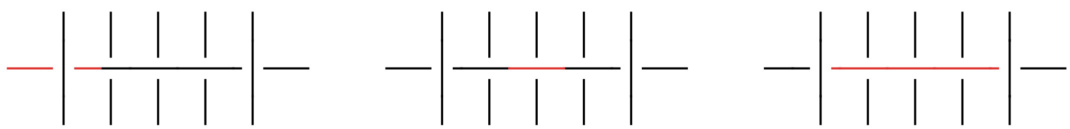


Figure 1.4: An underpass, an overpass and a bridge

The number of maximal overpasses in a knot diagram is the "bridge number" of the diagram. Different diagrams of the same knot can have different bridge numbers. The **bridge index of K**, denoted as $b(K)$, is the least bridge number of all diagrams of knot K [9]. The bridge index is sometimes refered to as the *bridge number*.

There is no known general method or algorithm for computing the bridge index of an arbitrary knot. As such, various methods have been used to compute bridge indexes. Schubert [10] introduced a projection which is now referred to as *Schubert normal form* and completely classified knots with bridge index 2, which includes many knots with less than 11 crossings. Musick [7] found the bridge index for all 11-crossing prime knots using a combination of methods. Musick presented the knots with bridge index two in two bridge form. For the knots with bridge index three, he notes that none of these are rational knots and exhibit a three bridge presentation. For knots with bridge index four, he uses a theorem which says that if a Montesinos knot K has $r \geq 3$ rational tangles, excluding integer tangles, then K has bridge index r. While this project was in progress, Blair et al [3] proved that the Wirtinger number of a knot equals its bridge index and used this finding to determine the bridge index of prime knots with up to 14 crossings. The bridge indexes from these and other sources are available in the table of knot invariants curated by Cha and Livingston [4].

Knots with a bridge index of 2, called *two-bridge knots*, are arguably the most well-understood class of knots. Below are several interesting facts about two-bridge knots:

- The two-bridge knots are exactly the rational knots [1].
- The bridge index of the composition of two knots is one less than the sum of the bridge indexes of the two knots, i.e., $b(K_1 \# K_2) = b(K_1) + b(K_2) - 1$ [10]. This implies that all two-bridge knots are prime.
- The number of distinct two-bridge knots of n crossings is at least $(2^{n-2} - 1)/3$ [5].

Though two-bridge knots are well understood, knots for which $b(K) > 2$ are not yet well understood. For example, classifying all of the knots with $b(K) = 3$ is an open problem.

1.3 Planar diagram codes

A **planar diagram (PD) code** is a numerical representation of a knot diagram comprised of 4-tuples. Each element of a 4-tuple corresponds to an edge in the knot diagram and each 4-tuple corresponds to a crossing.

The set of 4-tuples of a PD code representing a given labeled knot diagram is generated as follows. For each crossing we include the 4-tuple of edge labels involved beginning with the incoming under-edge and proceeding counter-clockwise around the crossing.

Note that a knot diagram may have multiple corresponding planar diagram codes depending on the the labeling of the diagram.

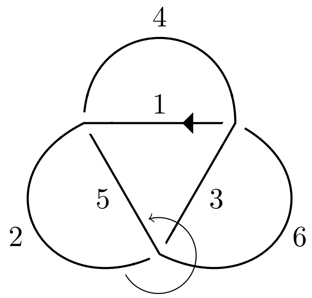


Figure 1.5: A labeled diagram of the trefoil with PD code $(2, 6, 3, 5)(4, 2, 5, 1)(6, 4, 1, 3)$

Given a planar diagram code, we can construct a diagram of the corresponding knot. As an example, we will construst a diagram of the trefoil from the planar diagram code we found above in Figure 1.5, $(2, 6, 3, 5)(4, 2, 5, 1)(6, 4, 1, 3)$. Begin by drawing a labeled crossing based on each tuple in the planar diagram code as in Figure 1.6. Note that it is not necessary for the crossings to be drawn in the same order as the correspoding tuples in the planar diagram code. Then connect the segments with matching labels as in Figure 1.7, being careful not to cross any edges which have already been connected.

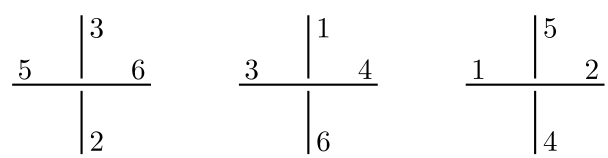


Figure 1.6: Crossings constructed from the PD code $(2, 6, 3, 5)(4, 2, 5, 1)(6, 4, 1, 3)$

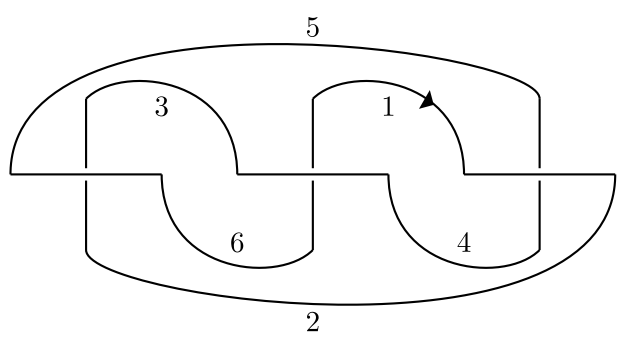


Figure 1.7: Diagram constructed from the PD code $(2, 6, 3, 5)(4, 2, 5, 1)(6, 4, 1, 3)$

Note that we are able to determine the orientation of a knot from a planar diagram code because the first element in each tuple corresponds to an edge that we follow under a strand to begin an undercross.

1.4 Reidemeister moves

There are three operations (and their inverses) which can be performed on a knot diagram without altering the corresponding knot. These operations, called **Reidemeister moves**, are local changes to the knot diagram. That is, the diagram remains unchanged except for the change depicted in Figure 1.8.

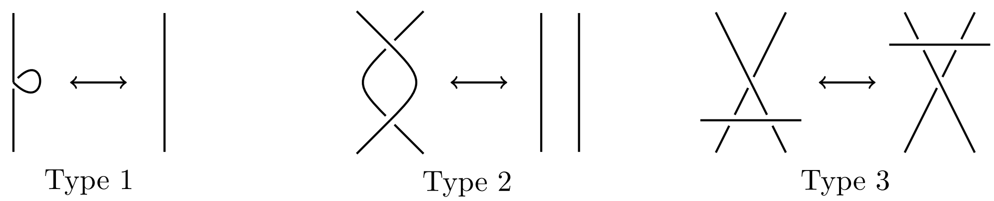


Figure 1.8: Reidemeister moves

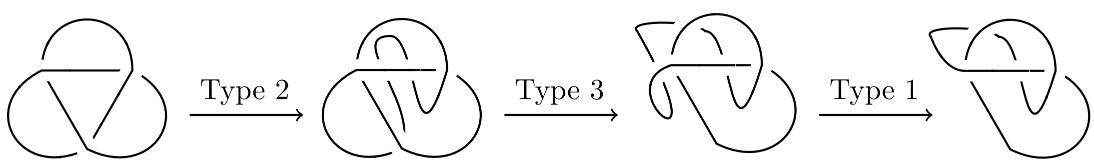


Figure 1.9: Reidemeister moves performed consecutively

As illustrated in Figure 1.9:

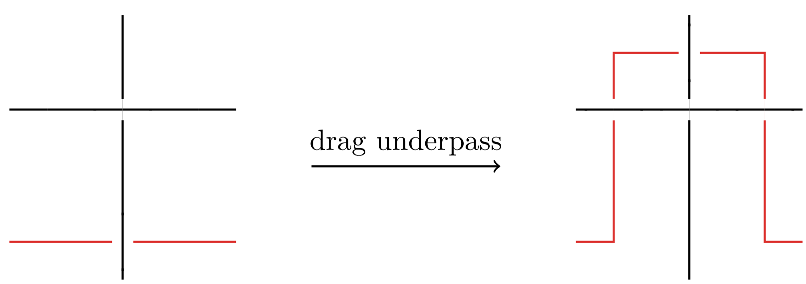
- a Reidemeister move of type 1 adds or removes one twist in a knot,
- a Reidemeister move of type 2 adds or removes two crossings by sliding one strand over or under another strand, and
- a Reidemeister move of type 3 changes the organization of three adjacent crossings by sliding a strand from one side of a crossing to the other.

Theorem 1.1. *Two knot diagrams belonging to the same knot, up to planar isotopy, can be related by a sequence of the three Reidemeister moves.*

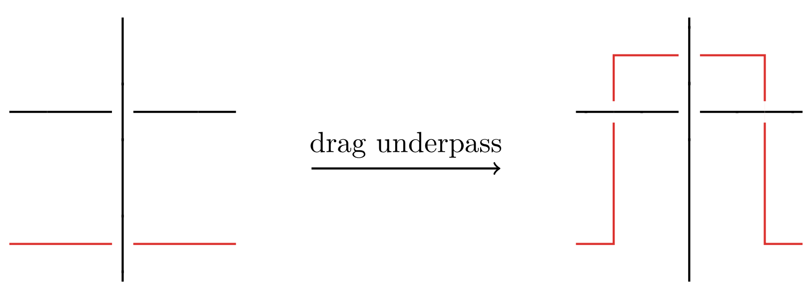
Proofs of this theorem were published independenly by Reidemeister in 1927 [8] and by Alexander and Briggs in 1926 [2].

1.5 "Drag the underpass" move

We introduce a method of altering a knot diagram called the "drag the underpass" move. To perform a "drag the underpass" move, drag the underpass strand of a crossing along the overpass strand to the opposite side of an adjacent crossing as illustrated in Figure 1.10. Throughout this paper we will focus on "drag the underpass" moves for which the overpass strand we drag the underpass along forms an overpass at the adjacent crossing, and this will be discussed further in sections 2.3 and 3.3.



Dragging an underpass along a segment which becomes an underpass



Dragging an underpass along a segment which remains an overpass

Figure 1.10: Knot segments before and after a "drag the underpass" move

The “drag the underpass” move is equivalent to a Reidemeister move of type 2 followed by a Reidemeister move of type 3, as illustrated in Figure 1.11.

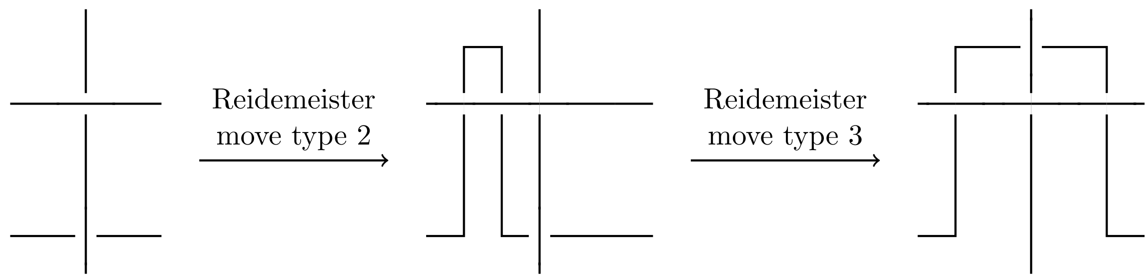


Figure 1.11: A “drag the underpass” move is a series of Reidemeister moves

CHAPTER 2
MANIPULATING PLANAR DIAGRAM CODES

In this chapter we present how planar diagram codes change as Reidemeister moves and "drag the underpass" moves are performed. We will begin by identifying how a PD code is structured when a crossing can be eliminated by a Reidemeister move of type 1 and present an algorithm for altering PD codes when such a move is performed. We will then identify how a PD code is structured when two crossings can be eliminated by a Reidemeister move of type 2 and present an algorithm for altering PD codes when such a move is performed. We will finish by identifying how a PD code is structured when a "drag the underpass" move can be performed and introduce an algorithm for altering PD codes when a "drag the underpass" move is performed.

2.1 Identifying and simplifying Reidemeister moves of type 1

A knot is said to be "simplifiable" by a Reidemeister move of type 1 if a crossing of the knot can be eliminated by performing a Reidemeister move of type 1 as discussed in Chapter 1.

Proposition. *A planar diagram code tuple indicates that a knot can be simplified by a Reidemeister move of type 1 if and only if the tuple contains at most three unique values.*

Proof. We begin by showing that if a knot can be simplified by a Reidemeister move of type 1, then the PD code tuple corresponding to the crossing to be eliminated contains at most three unique values.

Consider all possible diagrams of a knot segment which can be simplified by a Reidemeister move of type 1, which are depicted in Figure 2.1.

Note that for each of the crossings depicted in Figure 2.1, the corresponding PD code tuple includes the value of the loop segment, b, twice.

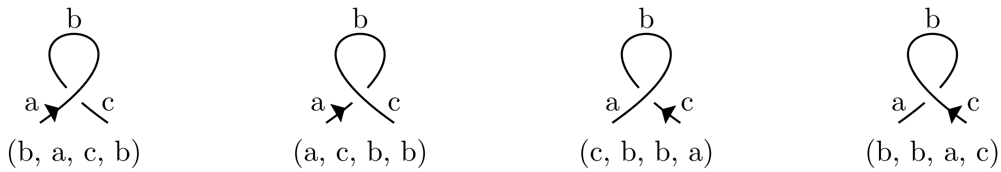


Figure 2.1: Knot segments simplifiable by a Reidemeister move of type 1 and the corresponding planar diagram code tuple

Hence if a knot can be simplified by a Reidemeister move of type 1, then the PD code tuple corresponding to the crossing to be eliminated contains at most three unique values.

We will finish by showing that every PD code tuple that has at most three unique values corresponds to a crossing which can be eliminated by a Reidemeister move of type 1.

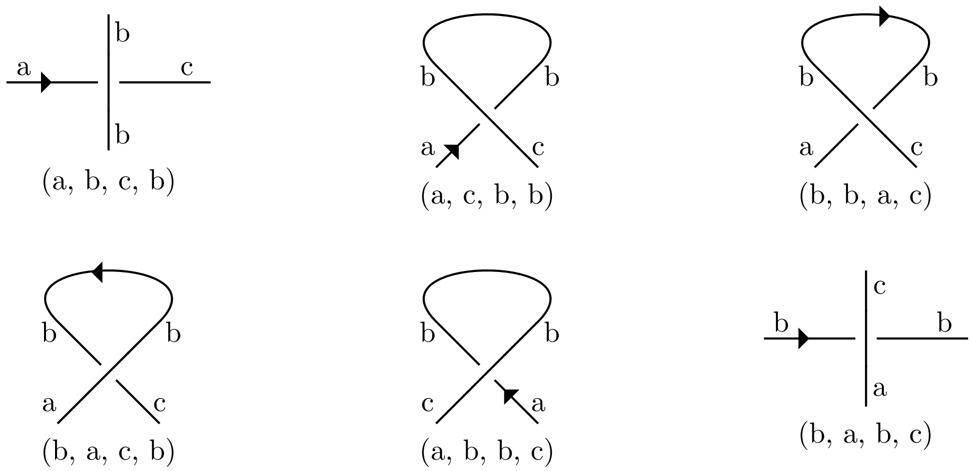


Figure 2.2: Diagrams corresponding to planar diagram codes with at most 3 unique values

Let us consider the diagrams that correspond to PD code tuples which contain at most three unique values, $(a, b, c, b), (a, c, b, b), (b, b, a, c), (b, a, c, b), (a, b, b, c)$ and (b, a, b, c), with $b \neq a, c$. Notice in Figure 2.2 that for the tuples (a, b, c, b) and (b, a, b, c) we cannot connect the two segments labeled b without crossing over or under either segment a or segment c. This indicates that these two tuples are not valid PD code tuples. All of the other crossings can be eliminated by un-twisting motion, which is a Reidemeister move of type 1. Hence every PD code tuple with at most three unique elements corresponds to a crossing of a knot which can be eliminated by a Reidemeister move of type 1. □

Proposition. *When a knot is simplified by a Reidemeister move of type 1, the planar diagram code of the knot changes as follows:*

1. *The tuple corresponding to the crossing that is eliminated by performing the Reidemiester move is removed from the planar diagram code.*
2. *Let b denote the PD code value of the edge that formed the loop which was eliminated and let m denote the maximum value of elements in the PD code before the Reidemeiseter move was performed.*

   Apply a function to each element of every PD code tuple, where the function to apply is determined by the value of b.

   If $1 < b < m$, apply the function f where f is defined as:

$$f(x) = \begin{cases} x & x < b \\ x - 2 & x > b \end{cases}$$

   If $b = 1$, apply the function g where g is defined as:

$$g(x) = \begin{cases} x - 2 & x > 2 \\ m - 2 & x = 2 \end{cases}$$

   If $b = m$, apply the function h where h is defined as:

$$h(x) = \begin{cases} x & x \leq b - 2 \\ 1 & x = b - 1 \end{cases}$$

Proof. Note that in the definitions of f, g, and h it is not necessary to consider the case $x = b$. The label of each edge in a knot diagram is included in the corresponding PD code exactly twice, and b was included twice in the tuple which was already removed from the PD code. Hence the case $x = b$ never arises.

We will find how a PD code changes when a knot is simplified through a Reidemeister move of type 1 by considering how the labels of edges change in the corresponding knot diagrams.

Begin by recognizing that when a knot diagram is simplified by a Reidemeister move of type 1, the edge labeled b is merged with the edges immediately before and after it (i.e., the edges $b - 1$ and $b + 1$), as illustrated in Figure 2.3.

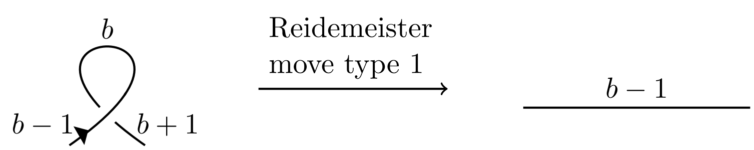


Figure 2.3: The merging of edges from a Reidemeister move of type 1

We now consider how the labels change depending on the value of b so that the resulting diagram has a valid labeling.

Case $1 < b < m$. The edges which originally had a label less than b remain unchanged and the edges which originally had a label greater than b have their labels decreased by 2 due to the edges that were merged together. As an example, consider the label changes in Figure 2.4.

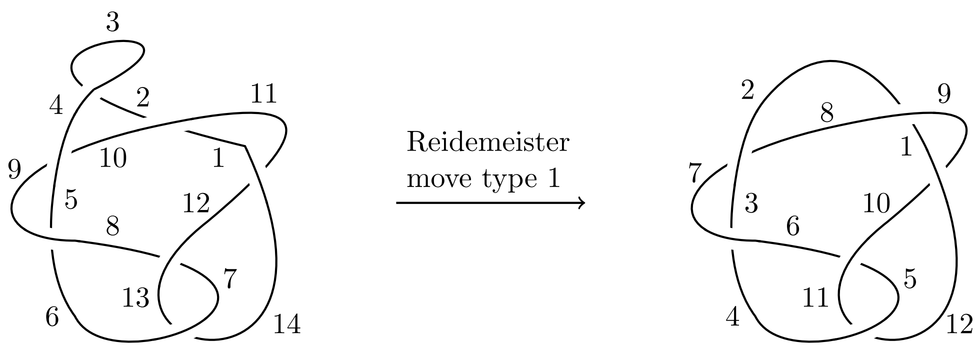


Figure 2.4: A diagram with $1 < b = 3 < m$ simplified by a Reidemeister move of type 1

Case $b = 1$**.** If we apply the label changes discussed in the case $1 < b < m$ so that $x \to x$ for all $x < b$ and $x \to x - 2$ for all $x > b$, then the edge which results as a merger of the edges originally labeled 2 and m is labeled 0 and $m - 2$, as illustrated in Figure 2.5. This labeling is problematic as the least label must be 1 and each edge may only have one label.

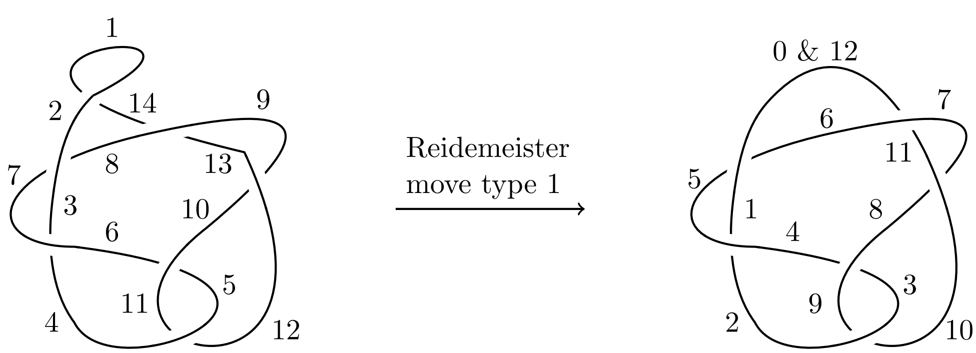


Figure 2.5: A knot diagram with $b = 1$ naïvely simplified by a Reidemeister move of type 1

The labeling can be fixed by adjusting the function applied to each element of the PD code so that 2 and $m - 2$ are mapped to the same value, $x \to x - 2$ for all $x > 2$ and $x \to m - 2$ if $x = 2$, as illustrated in Figure 2.6.

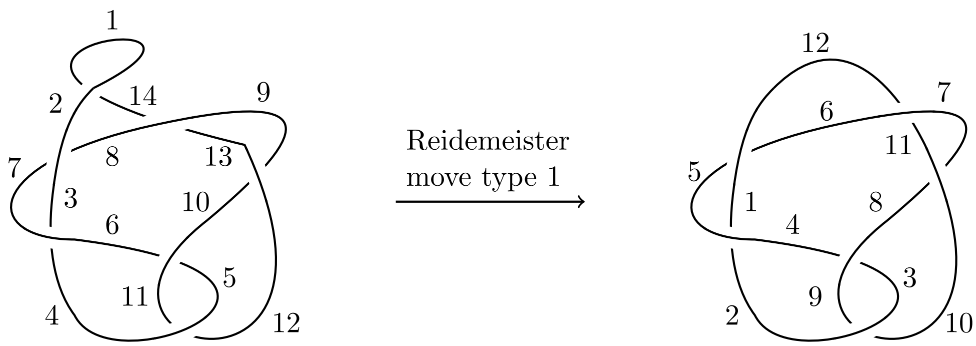


Figure 2.6: A knot diagram with $b = 1$ correctly simplified by a Reidemeister move of type 1

Case $b = m$. Consider what happens if we apply the label changes discussed in the case $1 < b < m$ so that $x \to x$ for all $x < b$ and $x \to x - 2$ for all $x > b$. As illustrated in Figure 2.7, the edge which results as a merger of the edges originally labeled 1 and $m - 1$ is labeled 1 and $m - 1$. This is not a valid labeling as each edge may only have one label and $m - 1$ is greater than the maximum allowed label, which is $m - 2$.

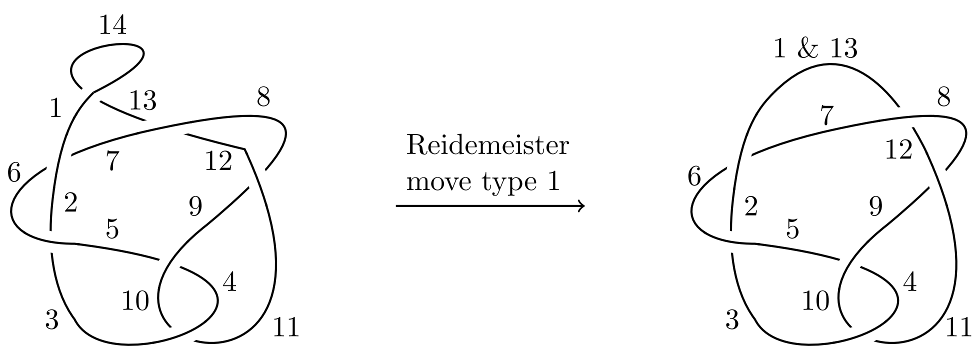


Figure 2.7: A diagram with $b = m$ naïvely simplified by a Reidemeister move of type 1

This can be fixed by adjusting the function applied to each element of the PD code so that 1 and $m-1$ are both mapped to 1 and all other labels remain the same, $x \to x$ for all $x \le m-2$ and $x \to 1$ for $x = m-1$, as illuatrated in Figure 2.8.

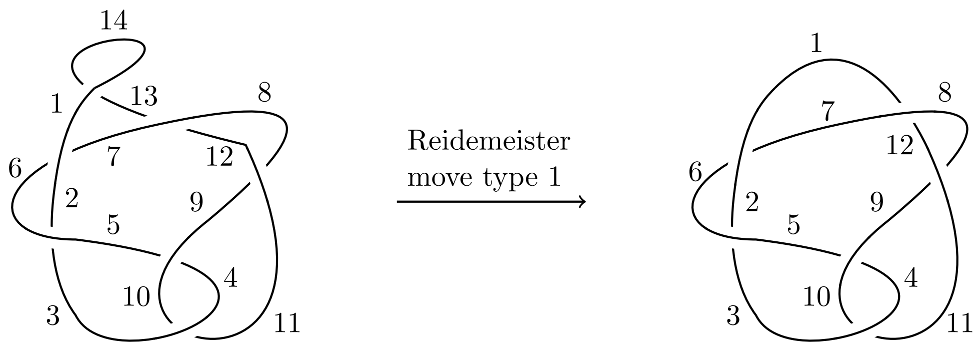


Figure 2.8: A diagram with $b = m$ correctly simplified by a Reidemeister move of type 1

□

2.2 Identifying and simplifying Reidemeister moves of type 2

A knot is said to be "simplifiable" by a Reidemeister move of type 2 if two crossings of the knot can be eliminated by performing a Reidemeister move of type 2.

Proposition. *A knot can be simplified by a Reidemeister move of type 2 if and only if a pair of tuples in a planar diagram code of the knot satisfy one of the following configurations of elements, where m is the maximum value in the PD code:*

1. (a, γ, β, b) *and* (β, γ, c, d) *with* $|a - \beta| = 1$

2. (a, b, β, γ) *and* (β, c, d, γ) *with* $|a - \beta| = 1$

3. $(m, b, 1, \beta)$ *and* $(1, a, 2, \beta)$

4. $(m, \beta, 1, b)$ *and* $(1, \beta, 2, a)$

Proof. We begin by showing that if a knot can be simplified by a Reidemeister move of type 2, then the PD code tuples corresponding to the crossings to be eliminated are of one of the four forms stated above.

First, consider the PD code tuples of a knot that can be simplified by a Reidemeister move of type 2 which are obtained by choosing a starting point that is not in the loop. Depending on the orientation of the knot, there are eight possible diagrams. These are depicted in Figure 2.9, supposing that we traverse the knot from a_1 toward a_2 and then from b_1 toward b_2.

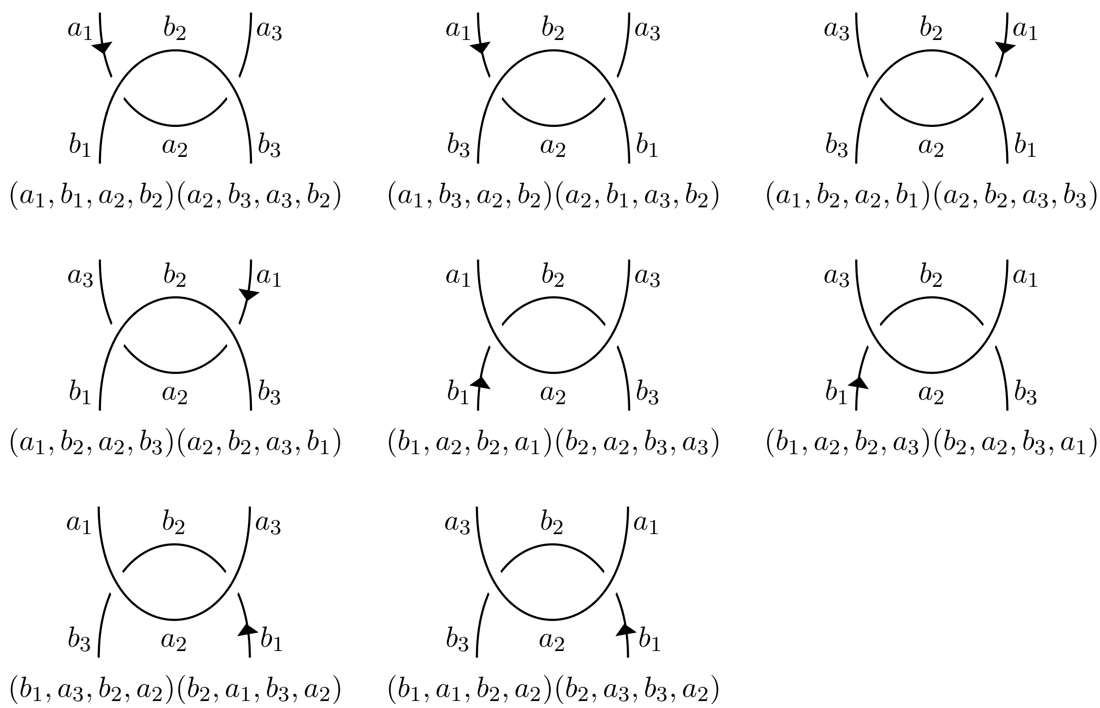


Figure 2.9: Knot segments which can be simplified by a Reidemeister move of type 2 and the corresponding PD code tuples

Note that the pairs of PD code tuples corresponding to these diagram segments have one of two forms - (a, γ, β, b) and (β, γ, c, d), or (a, b, β, γ) and (β, c, d, γ). Also note that since a and β are consecutive arcs of the knot, by the construction of the PD code $\beta = (a + 1) \bmod m$. Also by the construction of the PD code, $a_1 < a_2$ and $b_1 < b_2$, so $a \neq m$. Hence $\beta = a + 1$ and $|a - \beta| = 1$.

Second, consider the PD code tuples corresponding to the section of a knot which can be simplified by a Reidemeister move of type 2 where the knot diagram is labeled such that the edge labeled 1 is one of the arcs segments involved in the Reidemeister move. There are eight such possible diagrams, and these are illustrated in Figure 2.10 and Figure 2.11, supposing that we traverse the knot from 1 toward 2 and then from b_1 toward b_2.

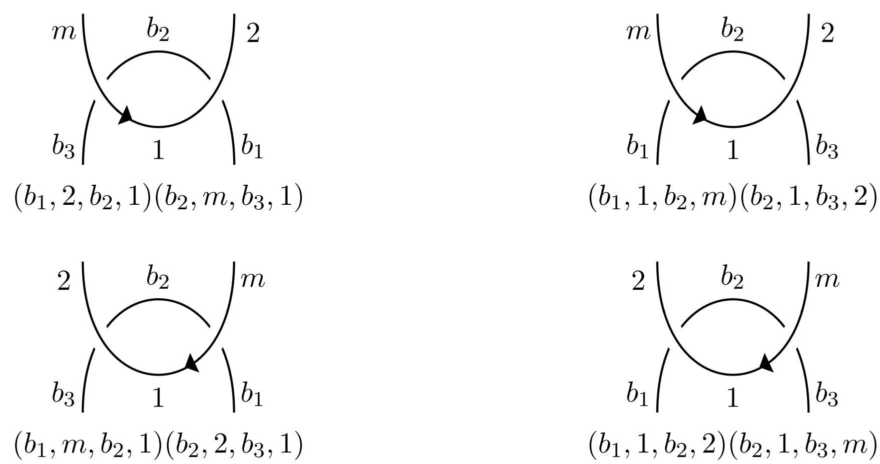


Figure 2.10: Diagrams of knot segments which can be simplified by a Reidemeister move of type 2 with a starting point in the arc forming an overpass

Note that in the cases that the knot edge labeled 1 forms an overpass, as in Figure 2.10, the PD code tuple pairs corresponding to these knot segments have the form (a, γ, β, b), (β, γ, c, d) or (a, b, β, γ), (β, c, d, γ) with $|a - \beta| = 1$.

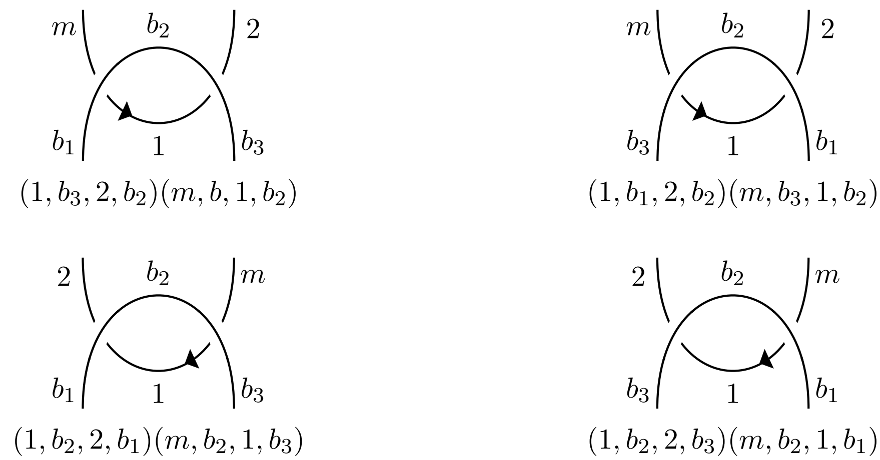


Figure 2.11: Diagrams of knot segments which can be simplified by a Reidemeister move of type 2 with a starting point in the arc forming an underpass

In the cases that the edge labeled 1 dead-ends into a strand creating an underpass, as depicted in Figure 2.11, the PD code tuples corresponding to this segment of the knot have the form $(m, b, 1, \beta)$ and $(1, a, 2, \beta)$, or $(m, \beta, 1, b)$ and $(1, \beta, 2, a)$, where m is the maximum value in the PD code.

Hence if a knot can be simplified by a Reidemeister move of type 2, then the PD code tuples corresponding to the crossings to be eliminated are of one of the following forms, where m is the maximum value in the PD code:

1. (a, γ, β, b) and (β, γ, c, d) with $|a - \beta| = 1$
2. (a, b, β, γ) and (β, c, d, γ) with $|a - \beta| = 1$
3. $(m, b, 1, \beta)$ and $(1, a, 2, \beta)$
4. $(m, \beta, 1, b)$ and $(1, \beta, 2, a)$

We finish by showing that every pair of PD code tuples with one of the forms enumerated below corresponds to a segment of a knot which can be simplified by a Reidemeister move of type 2:

1. (a, γ, β, b) and (β, γ, c, d) with $|a - \beta| = 1$

2. (a, b, β, γ) and (β, c, d, γ) with $|a - \beta| = 1$

3. $(m, b, 1, \beta)$ and $(1, a, 2, \beta)$

4. $(m, \beta, 1, b)$ and $(1, \beta, 2, a)$

Let us consider the diagrams that correspond to PD codes of the form $(a, \gamma, \beta, b)(\beta, \gamma, c, d)$ and $(a, b, \gamma, \beta)(\gamma, c, d, \beta)$ with $|a - \beta| = 1$.

Figure 2.12: Knot segment diagrams with PD codes of the form $(a, \gamma, \beta, b)(\beta, \gamma, c, d)$ and $(a, b, \gamma, \beta)(\gamma, c, d, \beta)$ with $|a - \beta| = 1$

From inspection of Figure 2.12 we see that these knot segments can be simplified by a Reidemeister move of type 2.

Now let us consider the diagrams the correspond to PD code tuples of the form $(m, b, 1, \beta)(1, a, 2, \beta)$ and $(m, \beta, 1, b)(1, \beta, 2, a)$.

Figure 2.13: Knot segment diagrams with consecutive PD code tuples of the form $(m, b, 1, \beta)(1, a, 2, \beta)$ and $(m, \beta, 1, b)(1, \beta, 2, a)$

We see from inspection of the diagrams in Figure 2.13 that these knot segments can also be simplified by a Reidemeister move of type 2.

Therefore, two tuples in a planar diagram code indicate that the knot can be simplified by a Reidemeister move of type 2 if and only if the tuples are of one of the forms:

1. (a, γ, β, b) and (β, γ, c, d) with $|a - \beta| = 1$
2. (a, b, β, γ) and (β, c, d, γ) with $|a - \beta| = 1$
3. $(m, b, 1, \beta)$ and $(1, a, 2, \beta)$
4. $(m, \beta, 1, b)$ and $(1, \beta, 2, a)$

□

Proposition. *When a knot is simplified by a Reidemeister move of type 2, the planar diagram code of the knot changes as follows:*

1. *The PD code tuples corresponding to the pair of crossings that are eliminated are removed.*
2. *Every element in the remaining PD code tuples is adjusted by applying* $g \circ f$ *where* n_1, n_2 *denote the PD code values of the arcs that were eliminated from the knot with* $n_1 < n_2$*,* f *is defined by*

$$f(x) = \begin{cases} x & x < n_1 \\ x - 1 & x > n_1, n_1 = 1 \\ x - 2 & x > n_1, n_1 > 1 \end{cases}$$

g is defined by

$$g(x) = \begin{cases} x \bmod (m-2) & x < k \\ (x-1) \bmod (m-2) & x > k, k = 1 \\ (x-2) \bmod (m-2) & x > k, k > 1 \end{cases}$$

with

$$k = \begin{cases} n_2 - 1 & n_1 = 1 \\ n_2 - 2 & n_1 > 1 \end{cases}$$

and m is the maximum value of elements in the PD code tuples before this Reidemeister move was performed.

Proof.

By definition, each tuple in a planar diagram code has a one-to-one relationship with a crossing in the corresponding knot. Hence when two crossings are eliminated from a knot by a Reidemeister move of type 2, the corresponding tuples are removed from the planar diagram code.

Now consider how the labels of a knot diagram change when a Reidemeister move of type 2 is performed.

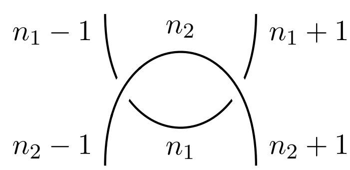


Figure 2.14: A labeled diagram segment simplifiable by a Reidemiester move of type 2

As illustrated in Figure 2.14, when a knot is simplified by a Reidemeister move of type 2 the edges n_1 and $n_1 + 1$ are merged into edge $n_1 - 1$ and the edges n_2 and $n_2 + 1$ are merged into edge $n_2 - 1$. Without loss of generality, let $n_1 < n_2$. Then the labels of edges originally labeled less than n_1 are not affected, the labels of edges originally labeled greater than $n_1 + 2$ decrease by 2 (since edges n_1 and $n_1 + 1$ are merged into edge $n_1 - 1$) and the edges originally labeled greater than $n_2 - 1$ are decreased by an additional 2 (since edges n_2 and $n_2 + 1$ are merged into edge $n_2 - 1$).

When adjusting the labels of edges, we need the resulting labeled diagram to satisfy the following criteria:

- each edge must have exactly one label
- the edges must be labeled sequentially beginning with 1 and ending with $m - 4$ (which is the number of edges in the new diagram)

Case $n_1 \neq 1$. Consider a knot diagram that can be simplified by a Reidemeister move of type 2 with $n_1 \neq 1$.

Suppose we apply h to each label where h is defined as:

$$h(x) = \begin{cases} x & x < n_1 \\ x - 2 & n_1 < x < n_2 \\ x - 4 & x > n_2 \end{cases}$$

Note that it is not necessary to consider the cases $x = n_1$ and $x = n_2$ in the definition of h. The label of each edge in a knot diagram is included in the corresponding PD code exactly twice, and n_1 and n_2 were included once in both of the tuples that were eliminated from the PD code. Hence the cases $x = n_1$ and $x = n_2$ never arise.

$h(1) = 1$, $h(m) = m - 4$, and for x and $x + 1$ in the same case range, $h(x + 1) = (x + 1) - y = x - y + 1 = (x - y) + 1 = h(x) + 1$. Hence h is strictly increasing within each case range and it is easy to see that h covers $[1, m - 4]$.

We have $h(n_1 - 1) = n_1 - 1$ and $h(n_1 + 1) = (n_1 + 1) - 2 = n_1 - 1$, so $h(n_1 - 1) = h(n_1 + 1)$, and we have $h(n_2 - 1) = (n_2 - 1) - 2 = n_2 - 3$ and $h(n_2 + 1) = (n_2 + 1) - 4 = n_2 - 3$, so $h(n_2 - 1) = h(n_2 + 1)$. Hence the labels of the edges that are merged together map to the same value, confirming that each edge has exactly one label.

Hence for $n_1 \neq 1$, h results a valid labeled diagram and PD code. This can be verified with the example in Figure 2.15.

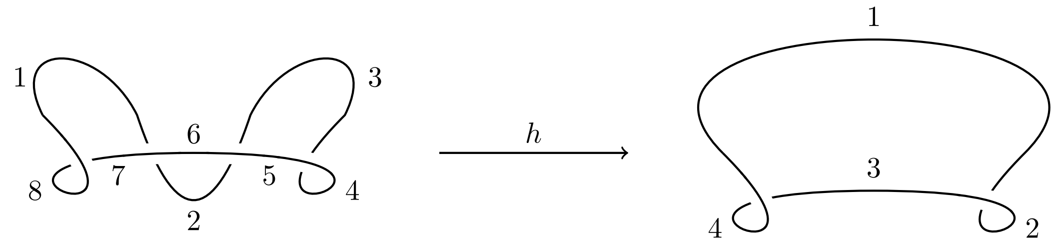


Figure 2.15: Simplify a diagram with $n_1 \neq 1$ by a Reidemeister move of type 2 using h

Case $n_1 = 1$. If we apply h to a PD code for which $n_1 = 1$, we quickly see that $h(n_1 + 1) = h(2) = 0$, which is not a valid label for a knot diagram. Also, $h(n_1 - 1) = h(m) = m - 4$, and so $h(n_1 + 1) \neq h(n_1 - 1)$, resulting in two different labels for the egde that is the result of edges on either side of edge n_1 being merged together. This is illustrated in Figure 2.16.

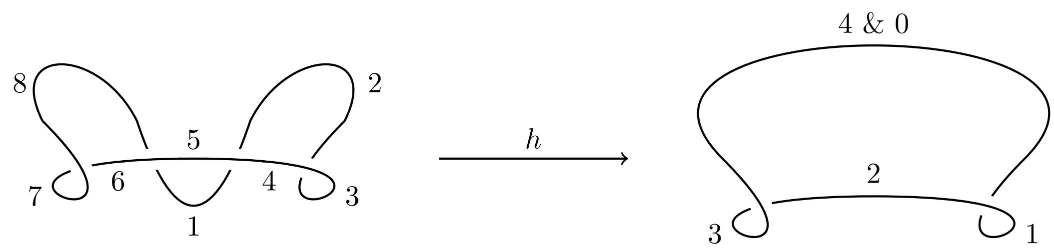


Figure 2.16: A diagram with $n_1 = 1$ naïvely simplified by a Reidemeister move of type 2 using h

We can fix both of these issues by defining a new function, j, such that $j(2) = h(m)$ and $j(x) = h(x)$ for all $x \neq 2$. Hence j is defined as:

$$j(x) = \begin{cases} m - 4 & x = 2 \\ x - 2 & 2 < x < n_2 \\ x - 4 & x > n_2 \end{cases}$$

$\square$

2.3 Identifying and preforming “drag the underpass” moves

Let (a, b, c, d) denote the crossing to be dragged and (e, f, g, h) the crossing directly before or after (a, b, c, d) which we will drag (a, b, c, d) underneath.

There are eight configurations for the crossings (a, b, c, d) and (e, f, g, h) depending on the orientation of the knot and which element the two tuples have in common. These eight configurations can be sub-divied into two sets - those with the segments *ac* and *eg* perpendicular to each other as in Figure 2.17 and those with segments *ac* and *eg* parallel to each other as in Figure 2.18.

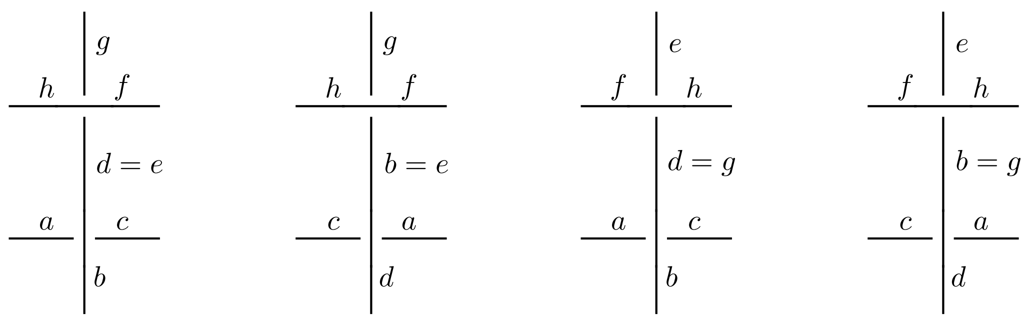


Figure 2.17: Knot segments with *ac* perpendicular to *eg*

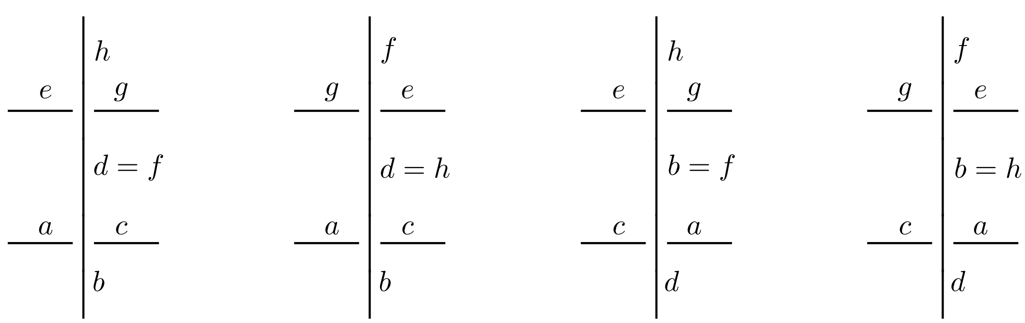
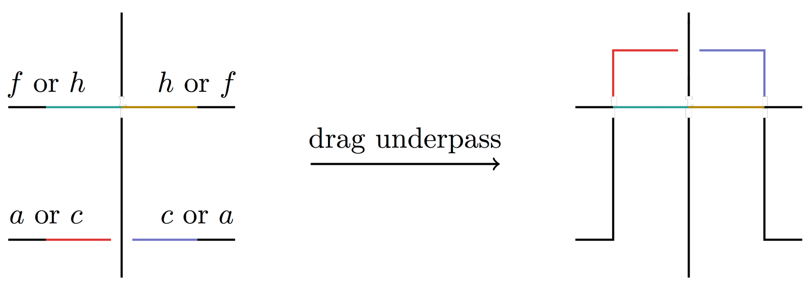


Figure 2.18: Knot segments with *ac* parallel to *eg*

We perform "drag the underpass" moves only for crossings arranged such that ac is perpendicular to eg, which as illustrated in 2.17 occurs when $d = e$, $b = e$, $d = g$ or $b = g$. The rational for this restriction is detailed in section 3.3, and for the remainder of this paper we will focus our discussion on these cases.

A "drag the underpass" move adds three crossings to the knot diagram and removes one crossing, resulting in four new edges in the diagram. This is illustrated in Figure 2.19, where the colored portions of the diagram on the left are the sections of the original diagram that become new edges after dragging the underpass.



Cases $d = e$, $b = e$, $d = g$, and $b = g$

Figure 2.19: Four edges added to a knot diagram by dragging (a, b, c, d) under (e, f, g, h)

Proposition. *When performing a "drag the underpass" move for which the crossing denoted by the PD code tuple (a, b, c, d) is dragged underneath the crossing denoted by the PD code tuple (e, f, g, h) and the crossings (a, b, c, d) and (e, f, g, h) are oriented such that one of the following is true: $d = e$, $b = e$, $d = g$, or $b = g$, then the change to the knot diagram can be expressed through the PD code by altering the tuples (a, b, c, d) and (e, f, g, h) as detailed in Appendix A and applying the function k to each element of all the other tuples in the PD code where the value of y is determined as discussed in the remark* *on page 26 and k is defined as*

$$k(x) = \begin{cases} x & x \leq min(a,y) \\ x+2 & min(a,y) < x \leq max(a,y) \\ x+4 & x > max(a,y) \end{cases}$$

Proof. The general idea is as follows, for which we assume $|f-h| = 1$. When (a,b,c,d) is dragged under (e,f,g,h), the crossing (a,b,c,d) is eliminated and three new crossings are added to the knot resulting in four new edges, as illustrated in Figure 2.19.

The labels of edges in the diagram remain unchanged until we reach the first new edge, so $x \to x$ for all $x \leq min(a,f,h)$.

We then have the first pair of new edges, labeled $min(a,f,h)+1$ and $min(a,f,h)+2$. The following edge, which was originally labeled $min(a,f,h)+1$, is now labeled $min(a,f,h)+3$ which is equal to the original label plus 2. We continue re-labeling the diagram by adding 1 to the label of the previous edge. Similarly, the labels of edges after the second pair of new edges are increased by an additional 2, and so
$x \to x+2$ for all x satisfying $min\,(a,f,h) < x \leq max\,(a,f,h)$ and
$x \to x+4$ for all $x > max\,(a,f,h)$.

Now consider how the value of $min(a,f,h)$ affects the labels of the new pairs of edges. If $min(a,f,h) = a$, then the labels of the first pair of new edges are $a+1$, $a+2$ and the labels of the second pair of new edges are $(y+2)+1 = y+3$ and $(y+2)+2 = y+4$, where $y = min(f,h)$. If $min(a,f,h) = f$, then the labels of the first pair of new edges are $f+1$ and $f+2$ and the labels of the second pair of new edges are $(a+2)+1 = a+3$ and $(a+2)+2 = a+4$. If $min(a,f,h) = h$, then the labels of the first pair of new edges are $h+1$ and $h+2$ and the labels of the second pair of new edges are $(a+2)+1 = a+3$ and $(a+2)+2 = a+4$.

Remark. *For the vast majority of labeled diagrams, $|f-h| = 1$ and the label changes described above result in a valid diagram labeling. However, if $f = 1$ and the knot is* *oriented such that* $h = m$*, or if* $f = m$ *and the knot is oriented such that* $h = 1$ *(i.e., if* $|f - h| \neq 1$*), then applying the label changes exactly as described above results in an invalid labeling similar to the invalid labelings considered in sections 2.1 and 2.2.*

To avoid these invalid labelings, in the label changes described above we replace $min(a, f, h)$ *with* $min(a, y)$*,*

$$y = \begin{cases} min\,(f, h) & |f - h| = 1 \\ m & |f - h| \neq 1 \end{cases}$$

Careful consideration needs to be given to how the three tuples which replace (a, b, c, d) in the PD code, which correspond to the three new crossings, are expressed and how tuple (e, f, g, h) is changed. There are three factors to consider:

- how (a, b, c, d) and (e, f, g, h) are oriented to each other (i.e., which elements of the (a, b, c, d) and (e, f, g, h) are equal),
- the order in which edges a, e, y are traveled, and
- the orientation of the knot.

Given these three factors, there are 48 cases to consider which are detailed in Appendix A.

Note that 48 is the result of the combinatorial caculation $4 * 3! * 2$ based on the three factors we must consider:

4 - the number of ways (a, b, c, d) and (e, f, g, h) may be oriented to each other. Recall that for this project we restrict ourselves to “drag the underpass” cases for which one of the following is true: $d = e$, $b = e$, $d = g$, or $b = g$,

3! - the number of ways to choose the order in which edges a, e, and y are traveled, and

2 - the number of options for the orientation of the knot. □

Example 2.1. *Consider the labeled diagram of* 6_1 *in Figure 2.20 which has the PD code* $(1,9,2,8)(3,7,4,6)(5,10,6,11)(7,3,8,2)(9,1,10,12)(11,4,12,5)$. *We will drag the underpass* $(a,b,c,d) = (3,7,4,6)$ *along segment* 6 *underneath the overpass* $(e,f,g,h) = (5,10,6,11)$.

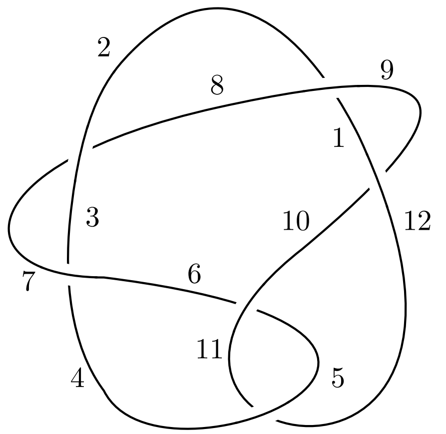


Figure 2.20: A labeled diagram of 6_1

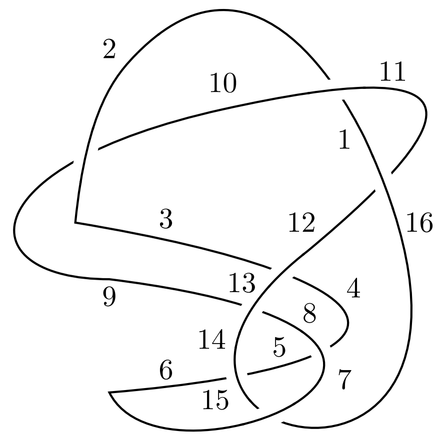


Figure 2.21: A labeled diagram of 6_1 after dragging an underpass

This is the case $d = g$, $a < e < y$ *and* $y = f$, *so from Appendix A we have that the tuples in the PD code change as follows:*

$(3,7,4,6) \to (3,13,4,12)(4,8,5,7)(5,14,6,3)$

$(5,10,6,11) \to (8,13,9,14)$

and we apply the function k *to each element of all the other tuples where* k *is defined as*

$$k(x) = \begin{cases} x & x \leq 3 \\ x+2 & 3 < x \leq 10 \\ x+4 & x > 10 \end{cases}$$

Hence the PD code after dragging the underpass is:
$(1, 11, 2, 10)(3, 13, 4, 12)(4, 8, 5, 7)(5, 14, 6, 15)(8, 13, 9, 14)(9, 3, 10, 2)(11, 1, 12, 16)(15, 6, 16, 7)$, *which can be verified with the labeled knot diagram in Figure 2.21.*

CHAPTER 3
COMPUTING AN UPPER BOUND ON THE BRIDGE INDEX

We produced a program using Python 2.7 that alters planar diagram codes to mimic the performance of Reidemeister moves of type 1 and 2 and “drag the underpass” moves on knot diagrams to naïvely find an upper bound on the bridge index of prime knots, and in this chapter we will detail how the program works. The source code of this program may be downloaded from https://doi.org/10.5281/zenodo.999014 and is included in Appendix C.

Remark. *Throughout the rest of this paper we use the term “bridge” to mean a maximal overpass which may have crossings added to it when a sequence of “drag the underpass” moves has been completed.*

The program is restricted from performing Reidemeister moves of type 1 and 2 which would add crossings to the knot. “Drag the underpass” moves which add crossings may be performed, but we restrict the edges to which we may add crossings. This restriction on “drag the underpass” moves is discussed in detail in section 3.3.

3.1 Data structures

The program handles data using two custom object types - a `Knot` and a `Crossing`. The key attributes of the `Knot` object type are described in Table 3.1, and the key attributes of the `Crossing` object type are described in Table 3.2.

Knot attribute	**Description**
name	a string to identify the knot in the output file
crossings	a list of `Crossing` objects, one per crossing in the knot diagram
free_crossings	a list of `Crossing` objects corresponding to the crossings in the knot diagram which are not covered by a bridge
bridges	a dictionary of key:value pairs where keys are integers and values are lists of the PD code label of the first and last edges of a bridge

Table 3.1: Attributes of the `Knot` object type

Crossing attribute	**Description**
bridge	the key of the element in `Knot.bridges` which corresponds to the bridge covering this crossing
pd_code	the PD code tuple corresponding to this crossing formatted as a list

Table 3.2: Attributes of the `Crossing` object type

3.2 Processing a planar diagram code

Each PD code in the input file is processed similar to a depth-first search, with the PD code in the input file being the root and each choice of a bridge "T" a branch. The general process is as follows:

A `Knot` object is created and then simplified by Reidemiester moves of type 1 and 2 until no more moves are possible.

```
knot = create_knot_from_pd_code(ast.literal_eval(row['pd_notation']),
  row['name'])
knot.simplify_rm1_rm2_recursively()
```

If there are no crossings remaining, the knot is equivalent to the unknot and the program moves on to process the next PD code. Otherwise, the program continues by creating a list of all combinations of bridge pair choices that form a “T”. Refer to section 3.3 for details about how bridges are chosen and why we restrict ourselves to selecting bridges that form a “T”.

```
if knot.free_crossings != []:
  base_knot_name = row['name']
  directory = 'knot_trees/' + base_knot_name
  knot.list_bridge_ts(directory, 0)
  ...
else:
  write_output(knot, outfile_name)
```

For each choice of bridge “T”, sequences of “drag the underpass” moves and simplifications by Reidemeister moves of type 1 and 2 are performed until no more moves are possible. Refer to section 3.4 for an explanation of how sequences of “drag the underpass” moves to perform are identified and completed and to section 3.5 for an explanation of how bridges that are simplified away are handled. When no more moves are possible, if all of the crossings are covered by a bridge then the number of bridges is stored in an output file and the processing of this branch is complete. Otherwise, if there is at least one crossing not covered by a bridge, then a list of bridge choices which form a “T” with an existing bridge is created. The process of dragging underpasses, simplifying by Reidemeister moves of types 1 and 2, and generating a list of bridge choices is repeated until all crossings are covered by a bridge and all the combinations of bridge “T” choices have been considered.

```
while knot.free_crossings != []:
  try:
    # Drag underpasses & simplify until no moves are possible.
    args = knot.find_crossing_to_drag()
    knot.drag_crossing_under_bridge_resursively(*args)
    knot.simplify_rm1_rm2_recursively()
  except:
    break
if knot.free_crossings == []:
  write_output(knot, outfile_name)
```

```
else:
  knot.list_bridge_ts(subdir, depth + 1)
  more_to_process = True
```

After all bridge "T" choices for the knot have been considered, the output file is searched for the minimum number of bridges that was recorded and this number is returned as our result.

```
find_minimum_computed_bridge_index()
```

3.3 Choosing overpasses to designate as bridges

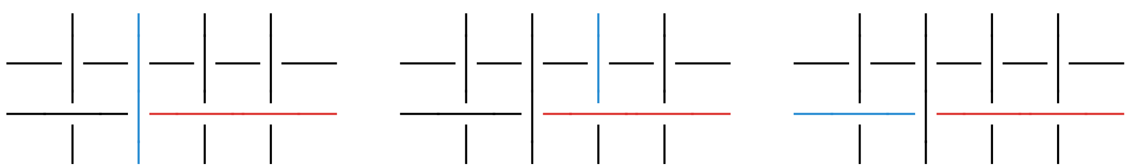
As mentioned in section 2.3, the program only performs a "drag the underpass" move for adjacent crossings (a, b, c, d) and (e, f, g, h) if one of the following is true: $d = e$, $b = e$, $d = g$, or $b = g$. As illustrated in Figure 3.1, these are the only arrangements of (a, b, c, d) and (e, f, g, h) for which performing a single "drag the underpass" move results in fewer crossings which are not covered by a bridge. We refer to a crossing which is not covered by a bridge as a **free crossing** and a crossing which is covered by a bridge a **bridge crossing**.

To improve the chances to being able to perform a sequence of "drag the underpass" moves to reduce the number of free crossings after designating a new bridge, we designate an overpass as a bridge only if it shares a crossing with an existing bridge and the shared crossing is not at the end of both the overpass and the bridge. This results in selecting overpasses which form a "T" with an existing bridge, as illustrated in Figure 3.2.

Cases $d = e$, $b = e$, $d = g$, and $b = g$

Cases $d = f$, $b = f$, $d = h$, and $b = h$

Figure 3.1: Bridge designations (denoted by red segments) necessary for new crossings to be covered by a bridge when dragging (a, b, c, d) under (e, f, g, h)



Good choice: The shared crossing is at the end of the bridge (red) but not the end of the overpass (blue)

Good choice: The shared crossing is at the end of the overpass (blue) but not the end of the bridge (red)

Poor choice: The shared crossing is at the end of the bridge (red) and the end of the overpass (blue)

Figure 3.2: Segment arrangements to consider when designating an overpass as a bridge

The method `Knot.list_bridge_ts()` generates a list of all "good" choices for overpasses to designate as a bridge based on the bridges that have already been designated. If no bridges have been designated, the list of overpasses to designate as bridges is each pair of overpasses that intersect at a crossing.

```
for a,b in itertools.combinations(self.free_crossings, 2):
  if list(set(a.pd_code).intersection(b.pd_code)):
    name = self.name + '_tree_' + str(i) + depth_suffix
    e,f,g,h = a.pd_code
    p,q,r,s = b.pd_code
    bridges = {0:[f,h], 1:[q,s]}
```

If bridges have been designated, the list of overpasses to designate as a bridge is all free crossings which share an edge with bridge crossing.

```
for a,b in itertools.product(self.bridge_crossings(),self.free_crossings):
  knot_copy = copy.deepcopy(self)
  if list(set(a.pd_code).intersection(b.pd_code)):
    knot_copy.designate_bridge(b)
```

3.4 Finding crossings for "drag the underpass" moves

Once bridges have been designated for a knot, "drag the underpass" moves may be performed. A naïve approach to performing "drag the underpass" moves is to try to drag each crossing which is not covered by a bridge. However, since our goal is to reduce the number of free crossings and dragging crossing (a, b, c, d) results in fewer free crossings only if the other crossing containing b or d is the intersection of two bridges and not the end of both bridges (refer to section 3.3), we can more efficiently choose crossings to drag.

A free crossings which can be eliminated by performing a sequence of "drag the underpass" moves can be found as follows, which is illustrated in Figure 3.3:

1. Beginning from the stem of a bridge "T", travel toward the cross bar of the "T" and continue traveling in this direction until a free crossing is reached. Let the PD code tuple of this free crossing be denoted by (a, b, c, d).

2. If b or d is the edge of the free crossing reached when traveling from the “T” stem, then crossing (a, b, c, d) can be dragged back along the edges traveled, under the cross bar of the bridge “T”, stopping under the stem of the bridge “T”.

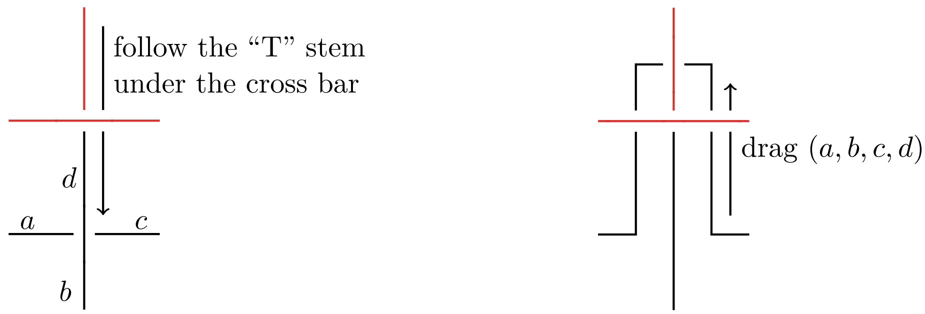


Figure 3.3: Follow a bridge “T” stem to the next free crossing and drag the free crossing back under the “T” stem

Note that a sequence of “drag the underpass” moves may be performed to drag (a, b, c, d) under multiple bridges on the way to a “T” stem, as illustrated in Figure 3.4.

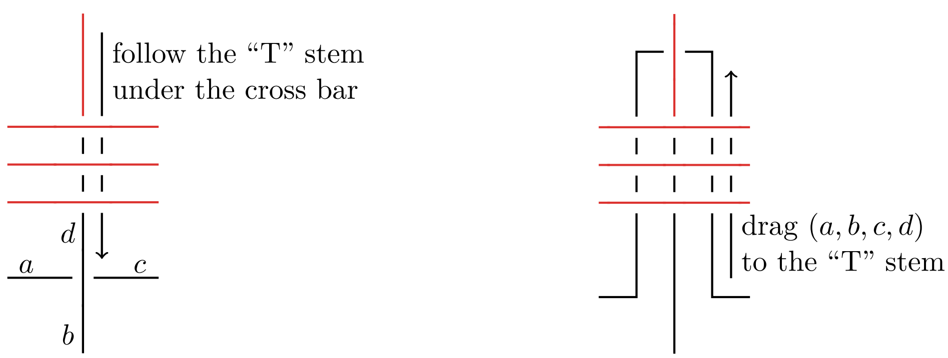


Figure 3.4: Drag a free crossing under multiple bridges to a bridge “T” stem

The method in the program which finds crossings to drag, `Knot.find_crossing_to_drag()`, does not necessarily find the most shortest sequence of “drag the underpass” moves necessarily to eliminate a free crossing. For example, given the knot segment in Figure 3.5 where the colored edges are bridges, `Knot.find_crossing_to_drag()` may follow the blue "T" stem and find that the free crossing can be dragged under the blue "T" stem by performing a sequence of 3 "drag the underpass" moves - one move for each bridge crossing traveled under as the bridge "T" stem is followed in search of a free crossing to drag. We call this number the **drag count**.

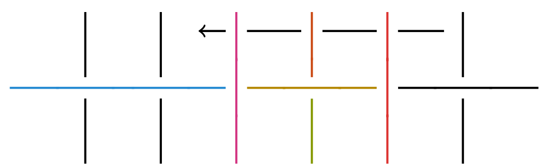


Figure 3.5: A free crossing found with a drag count of 3

However, as illustrated in Figure 3.6, only one "drag the underpass" move is necessary to eliminate the free crossing. Hence the drag count is only an upper bound on the number of "drag the underpass" moves necessary to eliminate a free crossing.

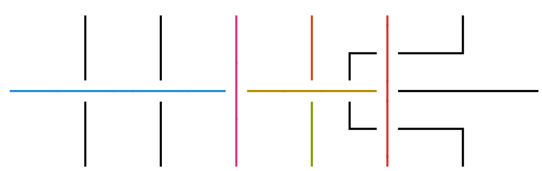


Figure 3.6: A free crossing eliminated with fewer "drag the underpass" moves than the drag count

To avoid unnecessarily adding crossings, we stop dragging the crossing once it is covered by a bridge, even if the number of "drag the underpass" moves performed is less than the drag count.

```
def drag_crossing_under_bridge_resursively(self, crossing_to_drag,
  adjacent_segment, drag_count):
  while (drag_count > 0):
    crossing_to_drag, adjacent_segment = self.drag_crossing_under_bridge(
    crossing_to_drag, adjacent_segment)
    drag_count -= 1
    # Stop if the crossing being dragged has been assigned to a bridge.
```

```
    if crossing_to_drag.bridge:
      break;
```

3.5 Checking for bridges eliminated by Reidemeister moves

Recall from section 3.1 that `Knot.bridges` is a dictionary of key:value pairs where keys are integers and values are lists of the PD code labels of the first and last edges of a bridge. As an example, `Knot.bridges == {0:[3,4], 1:[9,10], 2:[5,7]}` for the labeled diagram in Figure 3.7. Note that the integer used for each key is not important so long as the keys are unique and the order of the PD code labels in each values list does not matter.

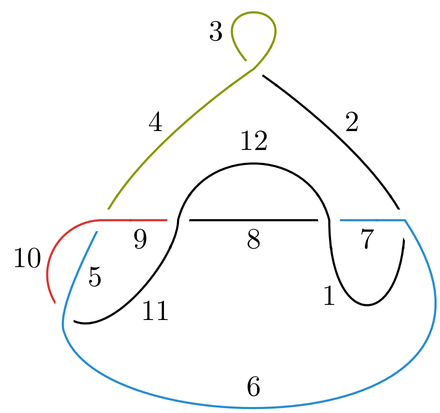
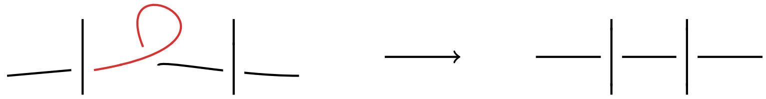
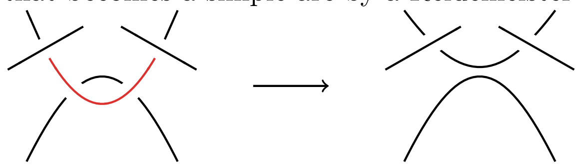


Figure 3.7: A knot diagram with several bridges (colored sections)

When a Reidemeister move of type 1 or 2 or a "drag the underpass" move is performed, the PD code labels stored in `Knot.bridges` are updated just as all of the labels in the PD code are updated. A bridge may become a simple arc (i.e., no longer pass over other strands of the knot) as illustrated in Figure 3.8 or be merged with another bridge as illustrated in Figure 3.9 when a Reidemeister move of type 1 or 2 is performed. We can determine if either of these changes has occured by inspecting the elements in `Knot.bridges` and then must prune the elements of `Knot.bridges` accrodingly.

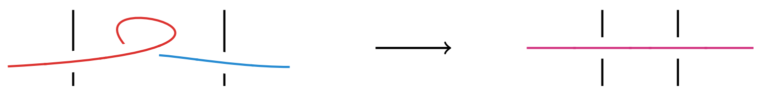


A bridge (red) that becomes a simple arc by a Reidemeister move of type 1

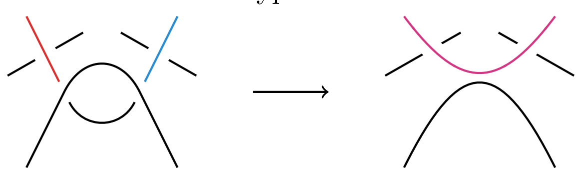


A bridge (red) that becomes a simple arc by a Reidemeister move of type 2

Figure 3.8: Bridges that become simple arcs by a Reidemeister move



Two bridges (red and blue) that become one bridge (purple) by a Reidemeister move of type 1



Two bridges (red and blue) that become one bridge (purple) by a Reidemeister move of type 2

Figure 3.9: Bridges that become one by a Reidemeister move

When a bridge becomes a simple arc, both of the PD code values stored in the corresponding element of `Knot.bridges` are equal. As an example, consider how the value of `Knot.bridges` corresponding to the labeled diagram in Figure 3.10 changes when the bridge covering crossing $(2, 3, 3, 4)$ is eliminated via a Reidemeister move of type 1. Initially, `Knot.bridges == {0:[3,4], 1:[9,10], 2:[5,7]}`. When the loop at crossing $(2, 3, 3, 4)$ is eliminated, `Knot.bridges` becomes `{0:[2,2], 1:[7,8], 2:[3,5]}`. The PD code values in the element corresponding to the bridge which had covered crossing

(2, 3, 3, 4) are equal, indicating that the bridge has become a simple arc. Since the bridge no longer exists, the corresponding element is removed from `Knot.bridges` and the value of `Knot.bridges` becomes `{1:[7,8], 2:[3,5]}`.

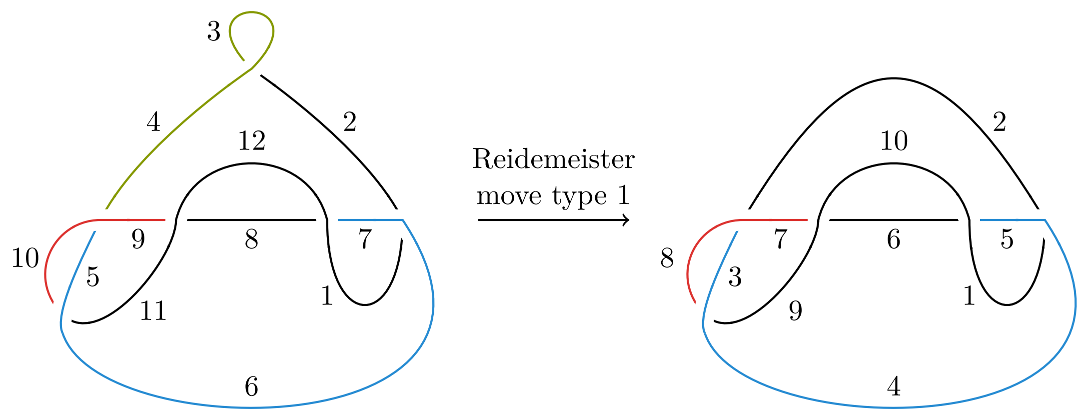


Figure 3.10: A bridge becomes a simple arc when a Reidemeister move of type 1 is performed

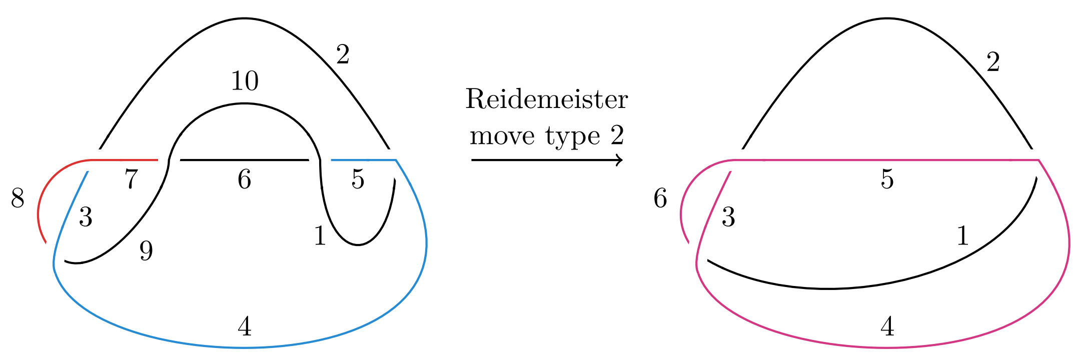


Figure 3.11: Bridges merged when a Reidemeister move of type 2 is performed

When two bridges are merged together, there is a shared value in their corresponding lists of PD code values in `Knot.bridges`. Continuing with the results of the previous example, consider how the value of `Knot.bridges` corresponding to the labeled diagram in Figure 3.11 changes when the red and blue bridges are merged together.

Initially the value of `Knot.bridges` is `{1:[7,8], 2:[3,5]}`. When edges 10 and 6 are eliminated via a Reidemeister move of type 2, `Knot.bridges` becomes `{1:[5,6], 2:[3,5]}`. The PD code value 5 is in each element of `Knot.bridges`, indicating that the bridges share an end and have been merged together. Since the bridges have been merged together, one of the elements containing in `Knot.bridges` containing 5 is removed and the value of `Knot.bridges` becomes `{1:[3,6]}`. Also, the `Crossing` objects for which `Crossing.bridge == 2` are updated so that `Crossing.bridge == 1`.

CHAPTER 4
RESULTS

The results of our program's computation of the upper bound on the bridge index of each prime knot with 3 through 12 crossings is given in Appendix B. These results are equal to the values published on Knotinfo [4] for prime knots with 3 through 11 crossings, which have been verified and accepted by the mathematics community. Our results also verify the values of the bridge indices of prime knots with 12 crossings which were published by Ryan Blair et al [3] while this project was in progress.

CHAPTER 5
FUTURE WORK

1. Automatically run `analyze_output.py` when `bridge_computation.py` is run.

2. Delete the csv files output by `bridge_computation.py` once they have been processed by `analyze_output.py`.

3. A combination of items 1 and 2 above, it would be ideal to run the process of `analyze_output.py` and clean up the csv files output by `bridge_computation.py` as each line in the input CSV file is processed to reduce the storage space necessary to run the program.

4. Return 1 as the number of bridges if `Knot.bridges` is empty, since by definition the bridge index of the unknot is 1 and not 0.

5. Validate the PD codes provided in the input CSV file.

6. How does the output of the program for prime knots with more than 12 crossings compare to the bridge indexes published by Ryan Blair, et al.[3] or others?

7. Can it be proven (or disproven) that the number of bridges output by the program for a particular knot is the bridge index of the knot?

8. Does the program successfully process links which have more than one component?

9. The program currently requires that the tuples in the PD codes input are ordered according to the order the crossings of the knot are traversed. While it is common practice to order the tuples in a PD code in this manner, it is not necessary. Adjust the methods which identify if a Reidemeister move or "drag the underpass" move can be performed to not require that the tuples be in any particular order.

10. We chose to designate an overpass as a bridge only if the overpass formed as “T” with a bridge to minimize the chances of designating a bridge which later becomes a simple arc or is merged with another bridge. For each knot diagram, what is the maximum number of overpasses which can be designated as a bridge without forming a “T”. Is this number meaningful in any way?

BIBLIOGRAPHY

[1] Adams, Colin Conrad. *The Knot Book: An Elementary Introduction to the Mathematical Theory of Knots*. 1994.

[2] Alexander, J. W., and G. B. Briggs. “On Types of Knotted Curves.” *Annals of Mathematics*, Second Series 28, no. 1/4 (1926): 562-86. doi:10.2307/1968399.

[3] Blair, Ryan, Alexandra Kjuchukova, Roman Velazquez, and Paul Villanueva. *Wirtinger systems of generators of knot groups*. (May 2017) arXiv:1705.03108v1.

[4] Cha, J. C., and C. Livingston, *KnotInfo: Table of Knot Invariants*, http://www.indiana.edu/~knotinfo, March 3, 2017.

[5] Earnst, Charles, and D Sumners, “The growth of the number of prime knots.” *Proc. Cambridge Phil. Soc.* (1987) 102:303-315.

[6] Livingston, Charles. *Knot Theory*. Carus. 1993.

[7] Musick, Chad. *Minimal Bridge Projections for 11-Crossing Prime Knots*. (September 2012) arXiv:1208.4233v3.

[8] Reidemeister, Kurt. “Elementare Begründung der Knotentheorie” *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 5, no. 1 (December 1927): 24-32. doi:10.1007/BF02952507.

[9] Rolfsen, Dale. *Knots and Links*. Mathematics Lecture Series ; 7. 1990.

[10] Schubert, Horst. “Über eine numerische Knoteninvariante” *Mathematische Zeitschrift* (1954) 61: 245. https://doi-org.proxy.lib.uiowa.edu/10.1007/BF01181346.

APPENDIX A

ALTERATIONS TO PD CODE TUPLES (A,B,C,D) AND (E,F,G,H) WHEN DRAGGING AN UNDERPASS

Let m denote the maximum value of the elements in the planar diagram code before the drag the underpass move is preformed, and let y denote the element f or h we travel from toward the other assuming that we start traveling the diagram from the edge labeled 1. If $|f-h|=1$, then $y=min\,(f,h)$. Otherwise $min\,(f,h)=1$ and $max\,(f,h)=m$, in which case $y=m$.

Case $d=e$**,** $a<e<y$**,** $y=f$**:**

$(a,b,c,d)\rightarrow(a,f+4,a+1,(f+5)\bmod m)(a+1,e+2,a+2,g+2)(a+2,f+3,a+3,f+2$

$(e,f,g,h)\rightarrow(b+2,f+3,e+2,f+4)$

Case $d=e$**,** $a<e<y$**,** $y=h$**:**

$(a,b,c,d)\rightarrow(a,h+3,a+1,h+2)(a+1,e+2,a+2,g+2)(a+2,h+4,a+3,(h+5)\bmod m$

$(e,f,g,h)\rightarrow(b+2,h+4,e+2,h+3)$

Case $d=e$**,** $a<y<e$**,** $y=f$**:**

$(a,b,c,d)\rightarrow(a,f+4,a+1,(f+5)\bmod m)(a+1,e+4,a+2,g+4)(a+2,f+3,a+3,f+2)$

$(e,f,g,h)\rightarrow(b+4,f+3,e+4,f+4)$

Case $d=e$**,** $a<y<e$**,** $y=h$**:**

$(a,b,c,d)\rightarrow(a,h+3,a+1,h+2)(a+1,e+4,a+2,g+4)(a+2,h+4,a+3,(h+5)\bmod m)$

$(e,f,g,h)\rightarrow(b+4,h+4,e+4,h+3)$

Case $d=e$**,** $e<a<y$**,** $y=f$**:**

$(a,b,c,d)\rightarrow(a,f+4,a+1,(f+5)\bmod m)(a+1,e,a+2,g)(a+2,f+3,a+3,f+2)$

$(e,f,g,h)\rightarrow(b,f+3,e,f+4)$

Case $d = e$**,** $e < a < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+3, a+1, h+2)(a+1, e, a+2, g)(a+2, h+4, a+3, (h+5) \bmod m)$

$(e, f, g, h) \to (b, h+4, e, h+3)$

Case $d = e$**,** $e < y < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+2, a+3, f+3)(a+3, e, a+4, g)(a+4, f+1, (a+5) \bmod m, f)$

$(e, f, g, h) \to (b, f+1, e, f+2)$

Case $d = e$**,** $e < y < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+1, a+3, h)(a+3, e, a+4, g)(a+4, h+2, (a+5) \bmod m, h+3)$

$(e, f, g, h) \to (b, h+2, e, h+1)$

Case $d = e$**,** $y < a < e$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+2, a+3, f+3)(a+3, e+4, a+4, g+4)(a+4, f+1, (a+5) \bmod m, f)$

$(e, f, g, h) \to (b+4, f+1, e+4, f+2)$

Case $d = e$**,** $y < a < e$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+1, a+3, h)(a+3, e+4, a+4, g+4)(a+4, h+2, (a+5) \bmod m, h+3)$

$(e, f, g, h) \to (b+4, h+2, e+4, h+1)$

Case $d = e$**,** $y < e < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+2, a+3, f+3)(a+3, e+2, a+4, g+2)(a+4, f+1, (a+5) \bmod m, f)$

$(e, f, g, h) \to (b+2, f+1, e+2, f+2)$

Case $d = e$**,** $y < e < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+1, a+3, h)(a+3, e+2, a+4, g+2)(a+4, h+2, (a+5) \bmod m, h+3)$

$(e, f, g, h) \to (b+2, h+2, e+2, h+1)$

Case $b = e$**,** $a < e < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+2, a+1, f+3)(a+1, g+2, a+2, e+2)(a+2, (f+5) \bmod m, a+3, f+4)$

$(e, f, g, h) \to (d+2, f+3, e+2, f+4)$

Case $b = e$**,** $a < e < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, (h+5) \bmod m, a+1, h+4)(a+1, g+2, a+2, e+2)(a+2, h+2, a+3, h+3)$

$(e, f, g, h) \to (d+2, h+4, e+2, h+3)$

Case $b = e$**,** $a < y < e$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+2, a+1, f+3)(a+1, g+4, a+2, e+4)(a+2, (f+5) \bmod m, a+3, f+4)$

$(e, f, g, h) \to (d+4, f+3, e+4, f+4)$

Case $b = e$**,** $a < y < e$**,** $y = h$**:**

$(a, b, c, d) \to (a, (h+5) \bmod m, a+1, h+4)(a+1, g+4, a+2, e+4)(a+2, h+2, a+3, h+3)$

$(e, f, g, h) \to (d+4, h+4, e+4, h+3)$

Case $b = e$**,** $e < a < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+2, a+1, f+3)(a+1, g, a+2, e)(a+2, (f+5) \bmod m, a+3, f+4)$

$(e, f, g, h) \to (d, f+3, e, f+4)$

Case $b = e$**,** $e < a < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, (h+5) \bmod m, a+1, h+4)(a+1, g, a+2, e)(a+2, h+2, a+3, h+3)$

$(e, f, g, h) \to (d, h+4, e, h+3)$

Case $b = e$**,** $e < y < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f, a+3, f+1)(a+3, g, a+4, e)(a+4, f+3, (a+5) \bmod m, f+2)$

$(e, f, g, h) \to (d, f+1, e, f+2)$

Case $b = e$**,** $e < y < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+3, a+3, h+2)(a+3, g, a+4, e)(a+4, h, (a+5) \bmod m, h+1)$

$(e, f, g, h) \to (d, h+2, e, h+1)$

Case $b = e$**,** $y < a < e$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f, a+3, f+1)(a+3, g+4, a+4, e+4)(a+4, f+3, (a+5) \bmod m, f+2)$

$(e, f, g, h) \to (d+4, f+1, e+4, f+2)$

Case $b = e$**,** $y < a < e$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+3, a+3, h+2)(a+3, g+4, a+4, e+4)(a+4, h, (a+5) \bmod m, h+1)$

$(e, f, g, h) \to (d+4, h+2, e+4, h+1)$

Case $b = e$**,** $y < e < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f, a+3, f+1)(a+3, g+2, a+4, e+2)(a+4, f+3, (a+5) \bmod m, f+2)$

$(e, f, g, h) \to (d+2, f+1, e+2, f+2)$

Case $b = e$**,** $y < e < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+3, a+3, h+2)(a+3, g+2, a+4, e+2)(a+4, h, (a+5) \bmod m, h+1)$

$(e, f, g, h) \to (d+2, h+2, e+2, h+1)$

Case $d = g$**,** $a < e < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+3, a+1, f+2)(a+1, g+2, a+2, e+2)(a+2, f+4, a+3, (f+5) \bmod m)$

$(e, f, g, h) \to (g+2, f+3, b+2, f+4)$

Case $d = g$**,** $a < e < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+4, a+1, (h+5) \bmod m)(a+1, g+2, a+2, e+2)(a+2, h+3, a+3, h+2)$

$(e, f, g, h) \to (g+2, h+4, b+2, h+3)$

Case $d = g$**,** $a < y < e$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+3, a+1, f+2)(a+1, g+4, a+2, e+4)(a+2, f+4, a+3, (f+5) \bmod m)$

$(e, f, g, h) \to (g+4, f+3, b+4, f+4)$

Case $d = g$**,** $a < y < e$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+4, a+1, (h+5) \bmod m)(a+1, g+4, a+2, e+4)(a+2, h+3, a+3, h+2)$

$(e, f, g, h) \to (g+4, h+4, b+4, h+3)$

Case $d = g$**,** $e < a < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, f+3, a+1, f+2)(a+1, g, a+2, e)(a+2, f+4, a+3, (f+5) \bmod m)$

$(e, f, g, h) \to (g, f+3, b, f+4)$

Case $d = g$**,** $e < a < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, h + 4, a + 1, (h + 5) \bmod m)(a + 1, g, a + 2, e)(a + 2, h + 3, a + 3, h + 2)$

$(e, f, g, h) \to (g, h + 4, b, h + 3)$

Case $d = g$**,** $e < y < a$**,** $y = f$**:**

$(a, b, c, d) \to (a + 2, f + 1, a + 3, f)(a + 3, g, a + 4, e)(a + 4, f + 2, (a + 5) \bmod m, f + 3)$

$(e, f, g, h) \to (g, f + 1, b, f + 2)$

Case $d = g$**,** $e < y < a$**,** $y = h$**:**

$(a, b, c, d) \to (a + 2, h + 2, a + 3, h + 3)(a + 3, g, a + 4, e)(a + 4, h + 1, (a + 5) \bmod m, h)$

$(e, f, g, h) \to (g, h + 2, b, h + 1)$

Case $d = g$**,** $y < a < e$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+1, a+3, f)(a+3, g+4, a+4, e+4)(a+4, f+2, (a+5) \bmod m, f+3)$

$(e, f, g, h) \to (g + 4, f + 1, b + 4, f + 2)$

Case $d = g$**,** $y < a < e$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+2, a+3, h+3)(a+3, g+4, a+4, e+4)(a+4, h+1, (a+5) \bmod m, h)$

$(e, f, g, h) \to (g + 4, h + 2, b + 4, h + 1)$

Case $d = g$**,** $y < e < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+1, a+3, f)(a+3, g+2, a+4, e+2)(a+4, f+2, (a+5) \bmod m, f+3)$

$(e, f, g, h) \to (g + 2, f + 1, b + 2, f + 2)$

Case $d = g$**,** $y < e < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h+2, a+3, h+3)(a+3, g+2, a+4, e+2)(a+4, h+1, (a+5) \bmod m, h)$

$(e, f, g, h) \to (g + 2, h + 2, b + 2, h + 1)$

Case $b = g$**,** $a < e < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, (f+5) \bmod m, a+1, f+4)(a+1, e+2, a+2, g+2)(a+2, f+2, a+3, f+3)$

$(e, f, g, h) \to (g + 2, f + 3, d + 2, f + 4)$

Case $b = g$**,** $a < e < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+2, a+1, h+3)(a+1, e+2, a+2, g+2)(a+2, (h+5) \bmod m, a+3, h+4)$

$(e, f, g, h) \to (g+2, h+4, d+2, h+3)$

Case $b = g$**,** $a < y < e$**,** $y = f$**:**

$(a, b, c, d) \to (a, (f+5) \bmod m, a+1, f+4)(a+1, e+4, a+2, g+4)(a+2, f+2, a+3, f+3)$

$(e, f, g, h) \to (g+4, f+3, d+4, f+4)$

Case $b = g$**,** $a < y < e$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+2, a+1, h+3)(a+1, e+4, a+2, g+4)(a+2, (h+5) \bmod m, a+3, h+4)$

$(e, f, g, h) \to (g+4, h+4, d+4, h+3)$

Case $b = g$**,** $e < a < y$**,** $y = f$**:**

$(a, b, c, d) \to (a, (f+5) \bmod m, a+1, f+4)(a+1, e, a+2, g)(a+2, f+2, a+3, f+3)$

$(e, f, g, h) \to (g, f+3, d, f+4)$

Case $b = g$**,** $e < a < y$**,** $y = h$**:**

$(a, b, c, d) \to (a, h+2, a+1, h+3)(a+1, e, a+2, g)(a+2, (h+5) \bmod m, a+3, h+4)$

$(e, f, g, h) \to (g, h+4, d, h+3)$

Case $b = g$**,** $e < y < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+3, a+3, f+2)(a+3, e, a+4, g)(a+4, f, (a+5) \bmod m, f+1)$

$(e, f, g, h) \to (g, f+1, d, f+2)$

Case $b = g$**,** $e < y < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h, a+3, h+1)(a+3, e, a+4, g)(a+4, h+3, (a+5) \bmod m, h+2)$

$(e, f, g, h) \to (g, h+2, d, h+1)$

Case $b = g$**,** $y < a < e$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+3, a+3, f+2)(a+3, e+4, a+4, g+4)(a+4, f, (a+5) \bmod m, f+1)$

$(e, f, g, h) \to (g+4, f+1, d+4, f+2)$

Case $b = g$**,** $y < a < e$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h, a+3, h+1)(a+3, e+4, a+4, g+4)(a+4, h+3, (a+5) \bmod m, h+2)$

$(e, f, g, h) \to (g+4, h+2, d+4, h+1)$

Case $b = g$**,** $y < e < a$**,** $y = f$**:**

$(a, b, c, d) \to (a+2, f+3, a+3, f+2)(a+3, e+2, a+4, g+2)(a+4, f, (a+5) \bmod m, f+1)$

$(e, f, g, h) \to (g+2, f+1, d+2, f+2)$

Case $b = g$**,** $y < e < a$**,** $y = h$**:**

$(a, b, c, d) \to (a+2, h, a+3, h+1)(a+3, e+2, a+4, g+2)(a+4, h+3, (a+5) \bmod m, h+2)$

$(e, f, g, h) \to (g+2, h+2, d+2, h+1)$

APPENDIX B

COMPUTED UPPER BOUNDS OF BRIDGE INDEXES

Table B.1: Computed bridge index of prime knots with 3 through 9 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
3_{0001}	2	8_{0008}	2	9_{0008}	2	9_{0029}	3
4_{0001}	2	8_{0009}	2	9_{0009}	2	9_{0030}	3
5_{0001}	2	8_{0010}	3	9_{0010}	2	9_{0031}	2
5_{0002}	2	8_{0011}	2	9_{0011}	2	9_{0032}	3
6_{0001}	2	8_{0012}	2	9_{0012}	2	9_{0033}	3
6_{0002}	2	8_{0013}	2	9_{0013}	2	9_{0034}	3
6_{0003}	2	8_{0014}	2	9_{0014}	2	9_{0035}	3
7_{0001}	2	8_{0015}	3	9_{0015}	2	9_{0036}	3
7_{0002}	2	8_{0016}	3	9_{0016}	3	9_{0037}	3
7_{0003}	2	8_{0017}	3	9_{0017}	2	9_{0038}	3
7_{0004}	2	8_{0018}	3	9_{0018}	2	9_{0039}	3
7_{0005}	2	8_{0019}	3	9_{0019}	2	9_{0040}	3
7_{0006}	2	8_{0020}	3	9_{0020}	2	9_{0041}	3
7_{0007}	2	8_{0021}	3	9_{0021}	2	9_{0042}	3
8_{0001}	2	9_{0001}	2	9_{0022}	3	9_{0043}	3
8_{0002}	2	9_{0002}	2	9_{0023}	2	9_{0044}	3
8_{0003}	2	9_{0003}	2	9_{0024}	3	9_{0045}	3
8_{0004}	2	9_{0004}	2	9_{0025}	3	9_{0046}	3
8_{0005}	3	9_{0005}	2	9_{0026}	2	9_{0047}	3
8_{0006}	2	9_{0006}	2	9_{0027}	2	9_{0048}	3
8_{0007}	2	9_{0007}	2	9_{0028}	3	9_{0049}	3

Table B.2: Computed bridge index of prime knots with 10 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
10_{0001}	2	10_{0025}	2	10_{0049}	3	10_{0073}	3
10_{0002}	2	10_{0026}	2	10_{0050}	3	10_{0074}	3
10_{0003}	2	10_{0027}	2	10_{0051}	3	10_{0075}	3
10_{0004}	2	10_{0028}	2	10_{0052}	3	10_{0076}	3
10_{0005}	2	10_{0029}	2	10_{0053}	3	10_{0077}	3
10_{0006}	2	10_{0030}	2	10_{0054}	3	10_{0078}	3
10_{0007}	2	10_{0031}	2	10_{0055}	3	10_{0079}	3
10_{0008}	2	10_{0032}	2	10_{0056}	3	10_{0080}	3
10_{0009}	2	10_{0033}	2	10_{0057}	3	10_{0081}	3
10_{0010}	2	10_{0034}	2	10_{0058}	3	10_{0082}	3
10_{0011}	2	10_{0035}	2	10_{0059}	3	10_{0083}	3
10_{0012}	2	10_{0036}	2	10_{0060}	3	10_{0084}	3
10_{0013}	2	10_{0037}	2	10_{0061}	3	10_{0085}	3
10_{0014}	2	10_{0038}	2	10_{0062}	3	10_{0086}	3
10_{0015}	2	10_{0039}	2	10_{0063}	3	10_{0087}	3
10_{0016}	2	10_{0040}	2	10_{0064}	3	10_{0088}	3
10_{0017}	2	10_{0041}	2	10_{0065}	3	10_{0089}	3
10_{0018}	2	10_{0042}	2	10_{0066}	3	10_{0090}	3
10_{0019}	2	10_{0043}	2	10_{0067}	3	10_{0091}	3
10_{0020}	2	10_{0044}	2	10_{0068}	3	10_{0092}	3
10_{0021}	2	10_{0045}	2	10_{0069}	3	10_{0093}	3
10_{0022}	2	10_{0046}	3	10_{0070}	3	10_{0094}	3
10_{0023}	2	10_{0047}	3	10_{0071}	3	10_{0095}	3
10_{0024}	2	10_{0048}	3	10_{0072}	3	10_{0096}	3

(table continues)

K	$b(K)$
10_{0097}	3
10_{0098}	3
10_{0099}	3
10_{0100}	3
10_{0101}	3
10_{0102}	3
10_{0103}	3
10_{0104}	3
10_{0105}	3
10_{0106}	3
10_{0107}	3
10_{0108}	3
10_{0109}	3
10_{0110}	3
10_{0111}	3
10_{0112}	3
10_{0113}	3
10_{0114}	3

K	$b(K)$
10_{0115}	3
10_{0116}	3
10_{0117}	3
10_{0118}	3
10_{0119}	3
10_{0120}	3
10_{0121}	3
10_{0122}	3
10_{0123}	3
10_{0124}	3
10_{0125}	3
10_{0126}	3
10_{0127}	3
10_{0128}	3
10_{0129}	3
10_{0130}	3
10_{0131}	3
10_{0132}	3

K	$b(K)$
10_{0133}	3
10_{0134}	3
10_{0135}	3
10_{0136}	3
10_{0137}	3
10_{0138}	3
10_{0139}	3
10_{0140}	3
10_{0141}	3
10_{0142}	3
10_{0143}	3
10_{0144}	3
10_{0145}	3
10_{0146}	3
10_{0147}	3
10_{0148}	3
10_{0149}	3
10_{0150}	3

K	$b(K)$
10_{0151}	3
10_{0152}	3
10_{0153}	3
10_{0154}	3
10_{0155}	3
10_{0156}	3
10_{0157}	3
10_{0158}	3
10_{0159}	3
10_{0160}	3
10_{0161}	3
10_{0162}	3
10_{0163}	3
10_{0164}	3
10_{0165}	3

Table B.3: Computed bridge index of alternating prime knots with 11 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11a_{0001}$	3	$11a_{0026}$	3	$11a_{0051}$	3	$11a_{0076}$	3
$11a_{0002}$	3	$11a_{0027}$	3	$11a_{0052}$	3	$11a_{0077}$	2
$11a_{0003}$	3	$11a_{0028}$	3	$11a_{0053}$	3	$11a_{0078}$	3
$11a_{0004}$	3	$11a_{0029}$	3	$11a_{0054}$	3	$11a_{0079}$	3
$11a_{0005}$	3	$11a_{0030}$	3	$11a_{0055}$	3	$11a_{0080}$	3
$11a_{0006}$	3	$11a_{0031}$	3	$11a_{0056}$	3	$11a_{0081}$	3
$11a_{0007}$	3	$11a_{0032}$	3	$11a_{0057}$	4	$11a_{0082}$	3
$11a_{0008}$	3	$11a_{0033}$	3	$11a_{0058}$	3	$11a_{0083}$	3
$11a_{0009}$	3	$11a_{0034}$	3	$11a_{0059}$	2	$11a_{0084}$	2
$11a_{0010}$	3	$11a_{0035}$	3	$11a_{0060}$	3	$11a_{0085}$	2
$11a_{0011}$	3	$11a_{0036}$	3	$11a_{0061}$	3	$11a_{0086}$	3
$11a_{0012}$	3	$11a_{0037}$	3	$11a_{0062}$	3	$11a_{0087}$	3
$11a_{0013}$	2	$11a_{0038}$	3	$11a_{0063}$	3	$11a_{0088}$	3
$11a_{0014}$	3	$11a_{0039}$	3	$11a_{0064}$	3	$11a_{0089}$	2
$11a_{0015}$	3	$11a_{0040}$	3	$11a_{0065}$	2	$11a_{0090}$	2
$11a_{0016}$	3	$11a_{0041}$	3	$11a_{0066}$	3	$11a_{0091}$	2
$11a_{0017}$	3	$11a_{0042}$	3	$11a_{0067}$	3	$11a_{0092}$	3
$11a_{0018}$	3	$11a_{0043}$	4	$11a_{0068}$	3	$11a_{0093}$	2
$11a_{0019}$	3	$11a_{0044}$	4	$11a_{0069}$	3	$11a_{0094}$	3
$11a_{0020}$	3	$11a_{0045}$	3	$11a_{0070}$	3	$11a_{0095}$	2
$11a_{0021}$	3	$11a_{0046}$	3	$11a_{0071}$	3	$11a_{0096}$	2
$11a_{0022}$	3	$11a_{0047}$	4	$11a_{0072}$	3	$11a_{0097}$	3
$11a_{0023}$	3	$11a_{0048}$	3	$11a_{0073}$	3	$11a_{0098}$	2
$11a_{0024}$	3	$11a_{0049}$	3	$11a_{0074}$	3	$11a_{0099}$	3
$11a_{0025}$	3	$11a_{0050}$	3	$11a_{0075}$	2	$11a_{0100}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11a_{0101}$	3	$11a_{0128}$	3	$11a_{0155}$	3	$11a_{0182}$	2
$11a_{0102}$	3	$11a_{0129}$	3	$11a_{0156}$	3	$11a_{0183}$	2
$11a_{0103}$	3	$11a_{0130}$	3	$11a_{0157}$	3	$11a_{0184}$	2
$11a_{0104}$	3	$11a_{0131}$	3	$11a_{0158}$	3	$11a_{0185}$	2
$11a_{0105}$	3	$11a_{0132}$	3	$11a_{0159}$	2	$11a_{0186}$	2
$11a_{0106}$	3	$11a_{0133}$	3	$11a_{0160}$	3	$11a_{0187}$	3
$11a_{0107}$	3	$11a_{0134}$	3	$11a_{0161}$	3	$11a_{0188}$	2
$11a_{0108}$	3	$11a_{0135}$	3	$11a_{0162}$	3	$11a_{0189}$	3
$11a_{0109}$	3	$11a_{0136}$	3	$11a_{0163}$	3	$11a_{0190}$	2
$11a_{0110}$	2	$11a_{0137}$	3	$11a_{0164}$	3	$11a_{0191}$	2
$11a_{0111}$	2	$11a_{0138}$	3	$11a_{0165}$	3	$11a_{0192}$	2
$11a_{0112}$	3	$11a_{0139}$	3	$11a_{0166}$	2	$11a_{0193}$	2
$11a_{0113}$	3	$11a_{0140}$	2	$11a_{0167}$	3	$11a_{0194}$	3
$11a_{0114}$	3	$11a_{0141}$	3	$11a_{0168}$	3	$11a_{0195}$	2
$11a_{0115}$	3	$11a_{0142}$	3	$11a_{0169}$	3	$11a_{0196}$	3
$11a_{0116}$	3	$11a_{0143}$	3	$11a_{0170}$	3	$11a_{0197}$	3
$11a_{0117}$	2	$11a_{0144}$	2	$11a_{0171}$	3	$11a_{0198}$	3
$11a_{0118}$	3	$11a_{0145}$	2	$11a_{0172}$	3	$11a_{0199}$	3
$11a_{0119}$	2	$11a_{0146}$	3	$11a_{0173}$	3	$11a_{0200}$	3
$11a_{0120}$	2	$11a_{0147}$	3	$11a_{0174}$	2	$11a_{0201}$	3
$11a_{0121}$	2	$11a_{0148}$	3	$11a_{0175}$	2	$11a_{0202}$	3
$11a_{0122}$	3	$11a_{0149}$	3	$11a_{0176}$	2	$11a_{0203}$	2
$11a_{0123}$	3	$11a_{0150}$	3	$11a_{0177}$	2	$11a_{0204}$	2
$11a_{0124}$	3	$11a_{0151}$	3	$11a_{0178}$	2	$11a_{0205}$	2
$11a_{0125}$	3	$11a_{0152}$	3	$11a_{0179}$	2	$11a_{0206}$	2
$11a_{0126}$	3	$11a_{0153}$	3	$11a_{0180}$	2	$11a_{0207}$	2
$11a_{0127}$	3	$11a_{0154}$	2	$11a_{0181}$	3	$11a_{0208}$	2

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11a_{0209}$	3	$11a_{0236}$	2	$11a_{0263}$	4	$11a_{0290}$	3
$11a_{0210}$	2	$11a_{0237}$	2	$11a_{0264}$	3	$11a_{0291}$	3
$11a_{0211}$	2	$11a_{0238}$	2	$11a_{0265}$	3	$11a_{0292}$	3
$11a_{0212}$	3	$11a_{0239}$	2	$11a_{0266}$	3	$11a_{0293}$	3
$11a_{0213}$	3	$11a_{0240}$	2	$11a_{0267}$	3	$11a_{0294}$	3
$11a_{0214}$	3	$11a_{0241}$	2	$11a_{0268}$	3	$11a_{0295}$	3
$11a_{0215}$	3	$11a_{0242}$	2	$11a_{0269}$	3	$11a_{0296}$	3
$11a_{0216}$	3	$11a_{0243}$	2	$11a_{0270}$	3	$11a_{0297}$	3
$11a_{0217}$	3	$11a_{0244}$	2	$11a_{0271}$	3	$11a_{0298}$	3
$11a_{0218}$	3	$11a_{0245}$	2	$11a_{0272}$	3	$11a_{0299}$	3
$11a_{0219}$	3	$11a_{0246}$	2	$11a_{0273}$	3	$11a_{0300}$	3
$11a_{0220}$	2	$11a_{0247}$	2	$11a_{0274}$	3	$11a_{0301}$	3
$11a_{0221}$	3	$11a_{0248}$	2	$11a_{0275}$	3	$11a_{0302}$	3
$11a_{0222}$	3	$11a_{0249}$	2	$11a_{0276}$	3	$11a_{0303}$	3
$11a_{0223}$	3	$11a_{0250}$	2	$11a_{0277}$	3	$11a_{0304}$	3
$11a_{0224}$	2	$11a_{0251}$	2	$11a_{0278}$	3	$11a_{0305}$	3
$11a_{0225}$	2	$11a_{0252}$	2	$11a_{0279}$	3	$11a_{0306}$	2
$11a_{0226}$	2	$11a_{0253}$	2	$11a_{0280}$	3	$11a_{0307}$	2
$11a_{0227}$	3	$11a_{0254}$	2	$11a_{0281}$	3	$11a_{0308}$	2
$11a_{0228}$	3	$11a_{0255}$	2	$11a_{0282}$	3	$11a_{0309}$	2
$11a_{0229}$	2	$11a_{0256}$	2	$11a_{0283}$	3	$11a_{0310}$	2
$11a_{0230}$	2	$11a_{0257}$	2	$11a_{0284}$	3	$11a_{0311}$	2
$11a_{0231}$	4	$11a_{0258}$	2	$11a_{0285}$	3	$11a_{0312}$	3
$11a_{0232}$	3	$11a_{0259}$	2	$11a_{0286}$	3	$11a_{0313}$	3
$11a_{0233}$	3	$11a_{0260}$	2	$11a_{0287}$	3	$11a_{0314}$	3
$11a_{0234}$	2	$11a_{0261}$	2	$11a_{0288}$	3	$11a_{0315}$	3
$11a_{0235}$	2	$11a_{0262}$	3	$11a_{0289}$	3	$11a_{0316}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11a_{0317}$	3	$11a_{0330}$	3	$11a_{0343}$	2	$11a_{0356}$	2
$11a_{0318}$	3	$11a_{0331}$	3	$11a_{0344}$	3	$11a_{0357}$	2
$11a_{0319}$	3	$11a_{0332}$	3	$11a_{0345}$	3	$11a_{0358}$	2
$11a_{0320}$	3	$11a_{0333}$	2	$11a_{0346}$	3	$11a_{0359}$	2
$11a_{0321}$	3	$11a_{0334}$	2	$11a_{0347}$	3	$11a_{0360}$	2
$11a_{0322}$	3	$11a_{0335}$	2	$11a_{0348}$	3	$11a_{0361}$	3
$11a_{0323}$	3	$11a_{0336}$	2	$11a_{0349}$	3	$11a_{0362}$	3
$11a_{0324}$	3	$11a_{0337}$	2	$11a_{0350}$	3	$11a_{0363}$	2
$11a_{0325}$	3	$11a_{0338}$	3	$11a_{0351}$	3	$11a_{0364}$	2
$11a_{0326}$	3	$11a_{0339}$	2	$11a_{0352}$	3	$11a_{0365}$	2
$11a_{0327}$	3	$11a_{0340}$	3	$11a_{0353}$	3	$11a_{0366}$	3
$11a_{0328}$	3	$11a_{0341}$	2	$11a_{0354}$	3	$11a_{0367}$	2
$11a_{0329}$	3	$11a_{0342}$	2	$11a_{0355}$	2		

Table B.4: Computed bridge index of non-alternating prime knots with 11 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11n_{0001}$	3	$11n_{0026}$	3	$11n_{0051}$	3	$11n_{0076}$	4
$11n_{0002}$	3	$11n_{0027}$	3	$11n_{0052}$	3	$11n_{0077}$	4
$11n_{0003}$	3	$11n_{0028}$	3	$11n_{0053}$	3	$11n_{0078}$	4
$11n_{0004}$	3	$11n_{0029}$	3	$11n_{0054}$	3	$11n_{0079}$	3
$11n_{0005}$	3	$11n_{0030}$	3	$11n_{0055}$	3	$11n_{0080}$	3
$11n_{0006}$	3	$11n_{0031}$	3	$11n_{0056}$	3	$11n_{0081}$	4
$11n_{0007}$	3	$11n_{0032}$	3	$11n_{0057}$	3	$11n_{0082}$	3
$11n_{0008}$	3	$11n_{0033}$	3	$11n_{0058}$	3	$11n_{0083}$	3
$11n_{0009}$	3	$11n_{0034}$	3	$11n_{0059}$	3	$11n_{0084}$	3
$11n_{0010}$	3	$11n_{0035}$	3	$11n_{0060}$	3	$11n_{0085}$	3
$11n_{0011}$	3	$11n_{0036}$	3	$11n_{0061}$	3	$11n_{0086}$	3
$11n_{0012}$	3	$11n_{0037}$	3	$11n_{0062}$	3	$11n_{0087}$	3
$11n_{0013}$	3	$11n_{0038}$	3	$11n_{0063}$	3	$11n_{0088}$	3
$11n_{0014}$	3	$11n_{0039}$	3	$11n_{0064}$	3	$11n_{0089}$	3
$11n_{0015}$	3	$11n_{0040}$	3	$11n_{0065}$	3	$11n_{0090}$	3
$11n_{0016}$	3	$11n_{0041}$	3	$11n_{0066}$	3	$11n_{0091}$	3
$11n_{0017}$	3	$11n_{0042}$	3	$11n_{0067}$	3	$11n_{0092}$	3
$11n_{0018}$	3	$11n_{0043}$	3	$11n_{0068}$	3	$11n_{0093}$	3
$11n_{0019}$	3	$11n_{0044}$	3	$11n_{0069}$	3	$11n_{0094}$	3
$11n_{0020}$	3	$11n_{0045}$	3	$11n_{0070}$	3	$11n_{0095}$	3
$11n_{0021}$	3	$11n_{0046}$	3	$11n_{0071}$	4	$11n_{0096}$	3
$11n_{0022}$	3	$11n_{0047}$	3	$11n_{0072}$	4	$11n_{0097}$	3
$11n_{0023}$	3	$11n_{0048}$	3	$11n_{0073}$	4	$11n_{0098}$	3
$11n_{0024}$	3	$11n_{0049}$	3	$11n_{0074}$	4	$11n_{0099}$	3
$11n_{0025}$	3	$11n_{0050}$	3	$11n_{0075}$	4	$11n_{0100}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$11n_{0101}$	3	$11n_{0123}$	3	$11n_{0145}$	3	$11n_{0167}$	3
$11n_{0102}$	3	$11n_{0124}$	3	$11n_{0146}$	3	$11n_{0168}$	3
$11n_{0103}$	3	$11n_{0125}$	3	$11n_{0147}$	3	$11n_{0169}$	3
$11n_{0104}$	3	$11n_{0126}$	3	$11n_{0148}$	3	$11n_{0170}$	3
$11n_{0105}$	3	$11n_{0127}$	3	$11n_{0149}$	3	$11n_{0171}$	3
$11n_{0106}$	3	$11n_{0128}$	3	$11n_{0150}$	3	$11n_{0172}$	3
$11n_{0107}$	3	$11n_{0129}$	3	$11n_{0151}$	3	$11n_{0173}$	3
$11n_{0108}$	3	$11n_{0130}$	3	$11n_{0152}$	3	$11n_{0174}$	3
$11n_{0109}$	3	$11n_{0131}$	3	$11n_{0153}$	3	$11n_{0175}$	3
$11n_{0110}$	3	$11n_{0132}$	3	$11n_{0154}$	3	$11n_{0176}$	3
$11n_{0111}$	3	$11n_{0133}$	3	$11n_{0155}$	3	$11n_{0177}$	3
$11n_{0112}$	3	$11n_{0134}$	3	$11n_{0156}$	3	$11n_{0178}$	3
$11n_{0113}$	3	$11n_{0135}$	3	$11n_{0157}$	3	$11n_{0179}$	3
$11n_{0114}$	3	$11n_{0136}$	3	$11n_{0158}$	3	$11n_{0180}$	3
$11n_{0115}$	3	$11n_{0137}$	3	$11n_{0159}$	3	$11n_{0181}$	3
$11n_{0116}$	3	$11n_{0138}$	3	$11n_{0160}$	3	$11n_{0182}$	3
$11n_{0117}$	3	$11n_{0139}$	3	$11n_{0161}$	3	$11n_{0183}$	3
$11n_{0118}$	3	$11n_{0140}$	3	$11n_{0162}$	3	$11n_{0184}$	3
$11n_{0119}$	3	$11n_{0141}$	3	$11n_{0163}$	3	$11n_{0185}$	3
$11n_{0120}$	3	$11n_{0142}$	3	$11n_{0164}$	3		
$11n_{0121}$	3	$11n_{0143}$	3	$11n_{0165}$	3		
$11n_{0122}$	3	$11n_{0144}$	3	$11n_{0166}$	3		

Table B.5: Computed bridge index of alternating prime knots with 12 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0001}$	3	$12a_{0026}$	3	$12a_{0051}$	3	$12a_{0076}$	3
$12a_{0002}$	3	$12a_{0027}$	3	$12a_{0052}$	3	$12a_{0077}$	3
$12a_{0003}$	3	$12a_{0028}$	3	$12a_{0053}$	3	$12a_{0078}$	3
$12a_{0004}$	3	$12a_{0029}$	4	$12a_{0054}$	3	$12a_{0079}$	3
$12a_{0005}$	3	$12a_{0030}$	4	$12a_{0055}$	3	$12a_{0080}$	3
$12a_{0006}$	3	$12a_{0031}$	3	$12a_{0056}$	3	$12a_{0081}$	3
$12a_{0007}$	3	$12a_{0032}$	3	$12a_{0057}$	3	$12a_{0082}$	3
$12a_{0008}$	3	$12a_{0033}$	4	$12a_{0058}$	3	$12a_{0083}$	3
$12a_{0009}$	3	$12a_{0034}$	3	$12a_{0059}$	3	$12a_{0084}$	3
$12a_{0010}$	3	$12a_{0035}$	3	$12a_{0060}$	3	$12a_{0085}$	3
$12a_{0011}$	3	$12a_{0036}$	4	$12a_{0061}$	3	$12a_{0086}$	3
$12a_{0012}$	3	$12a_{0037}$	3	$12a_{0062}$	3	$12a_{0087}$	3
$12a_{0013}$	3	$12a_{0038}$	2	$12a_{0063}$	3	$12a_{0088}$	3
$12a_{0014}$	3	$12a_{0039}$	3	$12a_{0064}$	3	$12a_{0089}$	3
$12a_{0015}$	3	$12a_{0040}$	3	$12a_{0065}$	3	$12a_{0090}$	3
$12a_{0016}$	3	$12a_{0041}$	3	$12a_{0066}$	3	$12a_{0091}$	3
$12a_{0017}$	3	$12a_{0042}$	3	$12a_{0067}$	3	$12a_{0092}$	3
$12a_{0018}$	3	$12a_{0043}$	3	$12a_{0068}$	3	$12a_{0093}$	3
$12a_{0019}$	3	$12a_{0044}$	3	$12a_{0069}$	3	$12a_{0094}$	3
$12a_{0020}$	3	$12a_{0045}$	3	$12a_{0070}$	3	$12a_{0095}$	3
$12a_{0021}$	3	$12a_{0046}$	3	$12a_{0071}$	3	$12a_{0096}$	3
$12a_{0022}$	3	$12a_{0047}$	3	$12a_{0072}$	3	$12a_{0097}$	3
$12a_{0023}$	3	$12a_{0048}$	3	$12a_{0073}$	3	$12a_{0098}$	3
$12a_{0024}$	3	$12a_{0049}$	3	$12a_{0074}$	3	$12a_{0099}$	3
$12a_{0025}$	3	$12a_{0050}$	3	$12a_{0075}$	3	$12a_{0100}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0101}$	3	$12a_{0128}$	3	$12a_{0155}$	3	$12a_{0182}$	4
$12a_{0102}$	3	$12a_{0129}$	3	$12a_{0156}$	3	$12a_{0183}$	4
$12a_{0103}$	3	$12a_{0130}$	3	$12a_{0157}$	3	$12a_{0184}$	4
$12a_{0104}$	3	$12a_{0131}$	3	$12a_{0158}$	3	$12a_{0185}$	4
$12a_{0105}$	3	$12a_{0132}$	3	$12a_{0159}$	3	$12a_{0186}$	4
$12a_{0106}$	3	$12a_{0133}$	3	$12a_{0160}$	3	$12a_{0187}$	4
$12a_{0107}$	3	$12a_{0134}$	3	$12a_{0161}$	3	$12a_{0188}$	4
$12a_{0108}$	3	$12a_{0135}$	3	$12a_{0162}$	3	$12a_{0189}$	4
$12a_{0109}$	3	$12a_{0136}$	3	$12a_{0163}$	3	$12a_{0190}$	4
$12a_{0110}$	3	$12a_{0137}$	3	$12a_{0164}$	3	$12a_{0191}$	4
$12a_{0111}$	3	$12a_{0138}$	3	$12a_{0165}$	3	$12a_{0192}$	4
$12a_{0112}$	3	$12a_{0139}$	3	$12a_{0166}$	3	$12a_{0193}$	4
$12a_{0113}$	3	$12a_{0140}$	3	$12a_{0167}$	3	$12a_{0194}$	4
$12a_{0114}$	3	$12a_{0141}$	3	$12a_{0168}$	3	$12a_{0195}$	4
$12a_{0115}$	3	$12a_{0142}$	3	$12a_{0169}$	3	$12a_{0196}$	4
$12a_{0116}$	3	$12a_{0143}$	3	$12a_{0170}$	3	$12a_{0197}$	4
$12a_{0117}$	3	$12a_{0144}$	3	$12a_{0171}$	3	$12a_{0198}$	4
$12a_{0118}$	3	$12a_{0145}$	3	$12a_{0172}$	3	$12a_{0199}$	4
$12a_{0119}$	3	$12a_{0146}$	3	$12a_{0173}$	3	$12a_{0200}$	4
$12a_{0120}$	3	$12a_{0147}$	3	$12a_{0174}$	3	$12a_{0201}$	4
$12a_{0121}$	3	$12a_{0148}$	3	$12a_{0175}$	3	$12a_{0202}$	4
$12a_{0122}$	3	$12a_{0149}$	3	$12a_{0176}$	3	$12a_{0203}$	4
$12a_{0123}$	3	$12a_{0150}$	3	$12a_{0177}$	3	$12a_{0204}$	4
$12a_{0124}$	3	$12a_{0151}$	3	$12a_{0178}$	3	$12a_{0205}$	4
$12a_{0125}$	3	$12a_{0152}$	3	$12a_{0179}$	3	$12a_{0206}$	4
$12a_{0126}$	3	$12a_{0153}$	3	$12a_{0180}$	3	$12a_{0207}$	4
$12a_{0127}$	3	$12a_{0154}$	3	$12a_{0181}$	3	$12a_{0208}$	4

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0209}$	3	$12a_{0236}$	3	$12a_{0263}$	3	$12a_{0290}$	3
$12a_{0210}$	3	$12a_{0237}$	3	$12a_{0264}$	3	$12a_{0291}$	3
$12a_{0211}$	3	$12a_{0238}$	3	$12a_{0265}$	3	$12a_{0292}$	3
$12a_{0212}$	3	$12a_{0239}$	2	$12a_{0266}$	3	$12a_{0293}$	3
$12a_{0213}$	3	$12a_{0240}$	3	$12a_{0267}$	3	$12a_{0294}$	3
$12a_{0214}$	3	$12a_{0241}$	2	$12a_{0268}$	3	$12a_{0295}$	3
$12a_{0215}$	3	$12a_{0242}$	3	$12a_{0269}$	3	$12a_{0296}$	3
$12a_{0216}$	3	$12a_{0243}$	2	$12a_{0270}$	3	$12a_{0297}$	3
$12a_{0217}$	3	$12a_{0244}$	3	$12a_{0271}$	3	$12a_{0298}$	3
$12a_{0218}$	3	$12a_{0245}$	3	$12a_{0272}$	3	$12a_{0299}$	3
$12a_{0219}$	3	$12a_{0246}$	3	$12a_{0273}$	3	$12a_{0300}$	3
$12a_{0220}$	3	$12a_{0247}$	2	$12a_{0274}$	3	$12a_{0301}$	3
$12a_{0221}$	2	$12a_{0248}$	3	$12a_{0275}$	3	$12a_{0302}$	3
$12a_{0222}$	3	$12a_{0249}$	3	$12a_{0276}$	3	$12a_{0303}$	3
$12a_{0223}$	3	$12a_{0250}$	3	$12a_{0277}$	3	$12a_{0304}$	3
$12a_{0224}$	3	$12a_{0251}$	2	$12a_{0278}$	3	$12a_{0305}$	3
$12a_{0225}$	3	$12a_{0252}$	3	$12a_{0279}$	3	$12a_{0306}$	3
$12a_{0226}$	2	$12a_{0253}$	3	$12a_{0280}$	3	$12a_{0307}$	3
$12a_{0227}$	3	$12a_{0254}$	3	$12a_{0281}$	3	$12a_{0308}$	3
$12a_{0228}$	3	$12a_{0255}$	3	$12a_{0282}$	3	$12a_{0309}$	3
$12a_{0229}$	3	$12a_{0256}$	3	$12a_{0283}$	3	$12a_{0310}$	3
$12a_{0230}$	3	$12a_{0257}$	2	$12a_{0284}$	3	$12a_{0311}$	3
$12a_{0231}$	3	$12a_{0258}$	3	$12a_{0285}$	3	$12a_{0312}$	3
$12a_{0232}$	3	$12a_{0259}$	2	$12a_{0286}$	3	$12a_{0313}$	3
$12a_{0233}$	3	$12a_{0260}$	3	$12a_{0287}$	3	$12a_{0314}$	3
$12a_{0234}$	3	$12a_{0261}$	2	$12a_{0288}$	3	$12a_{0315}$	3
$12a_{0235}$	3	$12a_{0262}$	3	$12a_{0289}$	3	$12a_{0316}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0317}$	3	$12a_{0344}$	3	$12a_{0371}$	3	$12a_{0398}$	3
$12a_{0318}$	3	$12a_{0345}$	3	$12a_{0372}$	3	$12a_{0399}$	3
$12a_{0319}$	3	$12a_{0346}$	3	$12a_{0373}$	3	$12a_{0400}$	3
$12a_{0320}$	3	$12a_{0347}$	3	$12a_{0374}$	3	$12a_{0401}$	3
$12a_{0321}$	3	$12a_{0348}$	3	$12a_{0375}$	3	$12a_{0402}$	3
$12a_{0322}$	3	$12a_{0349}$	3	$12a_{0376}$	3	$12a_{0403}$	3
$12a_{0323}$	3	$12a_{0350}$	3	$12a_{0377}$	3	$12a_{0404}$	3
$12a_{0324}$	3	$12a_{0351}$	3	$12a_{0378}$	3	$12a_{0405}$	3
$12a_{0325}$	3	$12a_{0352}$	3	$12a_{0379}$	3	$12a_{0406}$	3
$12a_{0326}$	3	$12a_{0353}$	3	$12a_{0380}$	3	$12a_{0407}$	3
$12a_{0327}$	3	$12a_{0354}$	3	$12a_{0381}$	3	$12a_{0408}$	3
$12a_{0328}$	3	$12a_{0355}$	3	$12a_{0382}$	3	$12a_{0409}$	3
$12a_{0329}$	3	$12a_{0356}$	3	$12a_{0383}$	3	$12a_{0410}$	3
$12a_{0330}$	2	$12a_{0357}$	3	$12a_{0384}$	3	$12a_{0411}$	3
$12a_{0331}$	3	$12a_{0358}$	3	$12a_{0385}$	3	$12a_{0412}$	3
$12a_{0332}$	3	$12a_{0359}$	3	$12a_{0386}$	3	$12a_{0413}$	3
$12a_{0333}$	3	$12a_{0360}$	3	$12a_{0387}$	3	$12a_{0414}$	3
$12a_{0334}$	3	$12a_{0361}$	3	$12a_{0388}$	3	$12a_{0415}$	3
$12a_{0335}$	3	$12a_{0362}$	3	$12a_{0389}$	3	$12a_{0416}$	3
$12a_{0336}$	3	$12a_{0363}$	3	$12a_{0390}$	3	$12a_{0417}$	3
$12a_{0337}$	3	$12a_{0364}$	3	$12a_{0391}$	3	$12a_{0418}$	3
$12a_{0338}$	3	$12a_{0365}$	3	$12a_{0392}$	3	$12a_{0419}$	3
$12a_{0339}$	3	$12a_{0366}$	3	$12a_{0393}$	3	$12a_{0420}$	3
$12a_{0340}$	3	$12a_{0367}$	3	$12a_{0394}$	3	$12a_{0421}$	3
$12a_{0341}$	3	$12a_{0368}$	3	$12a_{0395}$	3	$12a_{0422}$	3
$12a_{0342}$	3	$12a_{0369}$	3	$12a_{0396}$	3	$12a_{0423}$	3
$12a_{0343}$	3	$12a_{0370}$	3	$12a_{0397}$	3	$12a_{0424}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0425}$	2	$12a_{0452}$	3	$12a_{0479}$	3	$12a_{0506}$	2
$12a_{0426}$	3	$12a_{0453}$	3	$12a_{0480}$	3	$12a_{0507}$	3
$12a_{0427}$	3	$12a_{0454}$	3	$12a_{0481}$	3	$12a_{0508}$	2
$12a_{0428}$	3	$12a_{0455}$	3	$12a_{0482}$	2	$12a_{0509}$	3
$12a_{0429}$	3	$12a_{0456}$	3	$12a_{0483}$	3	$12a_{0510}$	2
$12a_{0430}$	3	$12a_{0457}$	3	$12a_{0484}$	3	$12a_{0511}$	2
$12a_{0431}$	3	$12a_{0458}$	3	$12a_{0485}$	3	$12a_{0512}$	2
$12a_{0432}$	3	$12a_{0459}$	3	$12a_{0486}$	3	$12a_{0513}$	3
$12a_{0433}$	3	$12a_{0460}$	3	$12a_{0487}$	3	$12a_{0514}$	2
$12a_{0434}$	3	$12a_{0461}$	3	$12a_{0488}$	3	$12a_{0515}$	3
$12a_{0435}$	3	$12a_{0462}$	3	$12a_{0489}$	3	$12a_{0516}$	3
$12a_{0436}$	3	$12a_{0463}$	3	$12a_{0490}$	3	$12a_{0517}$	2
$12a_{0437}$	2	$12a_{0464}$	3	$12a_{0491}$	3	$12a_{0518}$	2
$12a_{0438}$	3	$12a_{0465}$	3	$12a_{0492}$	3	$12a_{0519}$	2
$12a_{0439}$	3	$12a_{0466}$	3	$12a_{0493}$	3	$12a_{0520}$	2
$12a_{0440}$	3	$12a_{0467}$	3	$12a_{0494}$	3	$12a_{0521}$	2
$12a_{0441}$	3	$12a_{0468}$	3	$12a_{0495}$	3	$12a_{0522}$	2
$12a_{0442}$	3	$12a_{0469}$	3	$12a_{0496}$	3	$12a_{0523}$	3
$12a_{0443}$	3	$12a_{0470}$	3	$12a_{0497}$	2	$12a_{0524}$	3
$12a_{0444}$	3	$12a_{0471}$	3	$12a_{0498}$	2	$12a_{0525}$	3
$12a_{0445}$	3	$12a_{0472}$	3	$12a_{0499}$	2	$12a_{0526}$	3
$12a_{0446}$	3	$12a_{0473}$	3	$12a_{0500}$	2	$12a_{0527}$	3
$12a_{0447}$	2	$12a_{0474}$	3	$12a_{0501}$	2	$12a_{0528}$	2
$12a_{0448}$	3	$12a_{0475}$	3	$12a_{0502}$	2	$12a_{0529}$	3
$12a_{0449}$	3	$12a_{0476}$	3	$12a_{0503}$	3	$12a_{0530}$	3
$12a_{0450}$	3	$12a_{0477}$	3	$12a_{0504}$	3	$12a_{0531}$	3
$12a_{0451}$	3	$12a_{0478}$	3	$12a_{0505}$	3	$12a_{0532}$	2

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0533}$	2	$12a_{0560}$	3	$12a_{0587}$	3	$12a_{0614}$	3
$12a_{0534}$	2	$12a_{0561}$	3	$12a_{0588}$	3	$12a_{0615}$	3
$12a_{0535}$	2	$12a_{0562}$	3	$12a_{0589}$	3	$12a_{0616}$	3
$12a_{0536}$	2	$12a_{0563}$	3	$12a_{0590}$	3	$12a_{0617}$	3
$12a_{0537}$	2	$12a_{0564}$	3	$12a_{0591}$	3	$12a_{0618}$	3
$12a_{0538}$	2	$12a_{0565}$	3	$12a_{0592}$	3	$12a_{0619}$	3
$12a_{0539}$	2	$12a_{0566}$	3	$12a_{0593}$	3	$12a_{0620}$	3
$12a_{0540}$	2	$12a_{0567}$	3	$12a_{0594}$	3	$12a_{0621}$	3
$12a_{0541}$	2	$12a_{0568}$	3	$12a_{0595}$	2	$12a_{0622}$	3
$12a_{0542}$	3	$12a_{0569}$	3	$12a_{0596}$	2	$12a_{0623}$	3
$12a_{0543}$	3	$12a_{0570}$	3	$12a_{0597}$	2	$12a_{0624}$	3
$12a_{0544}$	3	$12a_{0571}$	3	$12a_{0598}$	3	$12a_{0625}$	3
$12a_{0545}$	2	$12a_{0572}$	3	$12a_{0599}$	3	$12a_{0626}$	3
$12a_{0546}$	3	$12a_{0573}$	3	$12a_{0600}$	2	$12a_{0627}$	3
$12a_{0547}$	3	$12a_{0574}$	3	$12a_{0601}$	2	$12a_{0628}$	3
$12a_{0548}$	3	$12a_{0575}$	3	$12a_{0602}$	3	$12a_{0629}$	3
$12a_{0549}$	2	$12a_{0576}$	3	$12a_{0603}$	3	$12a_{0630}$	3
$12a_{0550}$	2	$12a_{0577}$	3	$12a_{0604}$	3	$12a_{0631}$	3
$12a_{0551}$	2	$12a_{0578}$	3	$12a_{0605}$	3	$12a_{0632}$	3
$12a_{0552}$	2	$12a_{0579}$	2	$12a_{0606}$	3	$12a_{0633}$	3
$12a_{0553}$	3	$12a_{0580}$	2	$12a_{0607}$	3	$12a_{0634}$	3
$12a_{0554}$	4	$12a_{0581}$	2	$12a_{0608}$	3	$12a_{0635}$	3
$12a_{0555}$	3	$12a_{0582}$	2	$12a_{0609}$	3	$12a_{0636}$	3
$12a_{0556}$	3	$12a_{0583}$	2	$12a_{0610}$	3	$12a_{0637}$	3
$12a_{0557}$	3	$12a_{0584}$	2	$12a_{0611}$	3	$12a_{0638}$	3
$12a_{0558}$	3	$12a_{0585}$	2	$12a_{0612}$	3	$12a_{0639}$	3
$12a_{0559}$	3	$12a_{0586}$	3	$12a_{0613}$	3	$12a_{0640}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0641}$	3	$12a_{0668}$	3	$12a_{0695}$	3	$12a_{0722}$	2
$12a_{0642}$	3	$12a_{0669}$	3	$12a_{0696}$	3	$12a_{0723}$	2
$12a_{0643}$	2	$12a_{0670}$	3	$12a_{0697}$	3	$12a_{0724}$	2
$12a_{0644}$	2	$12a_{0671}$	3	$12a_{0698}$	3	$12a_{0725}$	3
$12a_{0645}$	3	$12a_{0672}$	3	$12a_{0699}$	3	$12a_{0726}$	2
$12a_{0646}$	3	$12a_{0673}$	3	$12a_{0700}$	3	$12a_{0727}$	2
$12a_{0647}$	3	$12a_{0674}$	3	$12a_{0701}$	3	$12a_{0728}$	2
$12a_{0648}$	3	$12a_{0675}$	3	$12a_{0702}$	3	$12a_{0729}$	2
$12a_{0649}$	2	$12a_{0676}$	3	$12a_{0703}$	3	$12a_{0730}$	3
$12a_{0650}$	2	$12a_{0677}$	3	$12a_{0704}$	3	$12a_{0731}$	2
$12a_{0651}$	2	$12a_{0678}$	3	$12a_{0705}$	3	$12a_{0732}$	2
$12a_{0652}$	2	$12a_{0679}$	3	$12a_{0706}$	3	$12a_{0733}$	2
$12a_{0653}$	3	$12a_{0680}$	3	$12a_{0707}$	3	$12a_{0734}$	3
$12a_{0654}$	3	$12a_{0681}$	3	$12a_{0708}$	3	$12a_{0735}$	3
$12a_{0655}$	3	$12a_{0682}$	2	$12a_{0709}$	3	$12a_{0736}$	2
$12a_{0656}$	3	$12a_{0683}$	3	$12a_{0710}$	3	$12a_{0737}$	3
$12a_{0657}$	3	$12a_{0684}$	2	$12a_{0711}$	3	$12a_{0738}$	2
$12a_{0658}$	3	$12a_{0685}$	3	$12a_{0712}$	3	$12a_{0739}$	3
$12a_{0659}$	3	$12a_{0686}$	3	$12a_{0713}$	2	$12a_{0740}$	2
$12a_{0660}$	3	$12a_{0687}$	3	$12a_{0714}$	2	$12a_{0741}$	3
$12a_{0661}$	3	$12a_{0688}$	3	$12a_{0715}$	2	$12a_{0742}$	3
$12a_{0662}$	3	$12a_{0689}$	3	$12a_{0716}$	2	$12a_{0743}$	2
$12a_{0663}$	3	$12a_{0690}$	2	$12a_{0717}$	2	$12a_{0744}$	2
$12a_{0664}$	3	$12a_{0691}$	2	$12a_{0718}$	2	$12a_{0745}$	2
$12a_{0665}$	3	$12a_{0692}$	4	$12a_{0719}$	3	$12a_{0746}$	3
$12a_{0666}$	3	$12a_{0693}$	4	$12a_{0720}$	2	$12a_{0747}$	3
$12a_{0667}$	3	$12a_{0694}$	4	$12a_{0721}$	2	$12a_{0748}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0749}$	3	$12a_{0776}$	3	$12a_{0803}$	2	$12a_{0830}$	3
$12a_{0750}$	4	$12a_{0777}$	3	$12a_{0804}$	3	$12a_{0831}$	3
$12a_{0751}$	3	$12a_{0778}$	3	$12a_{0805}$	3	$12a_{0832}$	3
$12a_{0752}$	3	$12a_{0779}$	3	$12a_{0806}$	3	$12a_{0833}$	3
$12a_{0753}$	3	$12a_{0780}$	3	$12a_{0807}$	3	$12a_{0834}$	3
$12a_{0754}$	3	$12a_{0781}$	3	$12a_{0808}$	3	$12a_{0835}$	3
$12a_{0755}$	3	$12a_{0782}$	3	$12a_{0809}$	3	$12a_{0836}$	3
$12a_{0756}$	3	$12a_{0783}$	3	$12a_{0810}$	3	$12a_{0837}$	3
$12a_{0757}$	3	$12a_{0784}$	3	$12a_{0811}$	3	$12a_{0838}$	3
$12a_{0758}$	2	$12a_{0785}$	3	$12a_{0812}$	3	$12a_{0839}$	3
$12a_{0759}$	2	$12a_{0786}$	3	$12a_{0813}$	3	$12a_{0840}$	3
$12a_{0760}$	2	$12a_{0787}$	3	$12a_{0814}$	3	$12a_{0841}$	3
$12a_{0761}$	2	$12a_{0788}$	3	$12a_{0815}$	3	$12a_{0842}$	3
$12a_{0762}$	2	$12a_{0789}$	3	$12a_{0816}$	3	$12a_{0843}$	3
$12a_{0763}$	2	$12a_{0790}$	3	$12a_{0817}$	3	$12a_{0844}$	3
$12a_{0764}$	2	$12a_{0791}$	2	$12a_{0818}$	3	$12a_{0845}$	3
$12a_{0765}$	3	$12a_{0792}$	2	$12a_{0819}$	3	$12a_{0846}$	3
$12a_{0766}$	3	$12a_{0793}$	3	$12a_{0820}$	3	$12a_{0847}$	3
$12a_{0767}$	3	$12a_{0794}$	3	$12a_{0821}$	3	$12a_{0848}$	3
$12a_{0768}$	3	$12a_{0795}$	3	$12a_{0822}$	3	$12a_{0849}$	3
$12a_{0769}$	3	$12a_{0796}$	2	$12a_{0823}$	3	$12a_{0850}$	3
$12a_{0770}$	3	$12a_{0797}$	2	$12a_{0824}$	3	$12a_{0851}$	3
$12a_{0771}$	3	$12a_{0798}$	3	$12a_{0825}$	3	$12a_{0852}$	3
$12a_{0772}$	3	$12a_{0799}$	3	$12a_{0826}$	3	$12a_{0853}$	3
$12a_{0773}$	2	$12a_{0800}$	3	$12a_{0827}$	3	$12a_{0854}$	3
$12a_{0774}$	2	$12a_{0801}$	4	$12a_{0828}$	3	$12a_{0855}$	3
$12a_{0775}$	2	$12a_{0802}$	2	$12a_{0829}$	3	$12a_{0856}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0857}$	3	$12a_{0884}$	3	$12a_{0911}$	3	$12a_{0938}$	3
$12a_{0858}$	3	$12a_{0885}$	3	$12a_{0912}$	3	$12a_{0939}$	3
$12a_{0859}$	3	$12a_{0886}$	3	$12a_{0913}$	3	$12a_{0940}$	3
$12a_{0860}$	3	$12a_{0887}$	3	$12a_{0914}$	3	$12a_{0941}$	3
$12a_{0861}$	3	$12a_{0888}$	3	$12a_{0915}$	3	$12a_{0942}$	3
$12a_{0862}$	3	$12a_{0889}$	3	$12a_{0916}$	3	$12a_{0943}$	3
$12a_{0863}$	3	$12a_{0890}$	3	$12a_{0917}$	3	$12a_{0944}$	3
$12a_{0864}$	3	$12a_{0891}$	3	$12a_{0918}$	3	$12a_{0945}$	3
$12a_{0865}$	3	$12a_{0892}$	3	$12a_{0919}$	3	$12a_{0946}$	3
$12a_{0866}$	3	$12a_{0893}$	3	$12a_{0920}$	3	$12a_{0947}$	3
$12a_{0867}$	3	$12a_{0894}$	3	$12a_{0921}$	3	$12a_{0948}$	3
$12a_{0868}$	3	$12a_{0895}$	3	$12a_{0922}$	3	$12a_{0949}$	3
$12a_{0869}$	3	$12a_{0896}$	3	$12a_{0923}$	3	$12a_{0950}$	3
$12a_{0870}$	3	$12a_{0897}$	3	$12a_{0924}$	3	$12a_{0951}$	3
$12a_{0871}$	3	$12a_{0898}$	3	$12a_{0925}$	3	$12a_{0952}$	3
$12a_{0872}$	3	$12a_{0899}$	3	$12a_{0926}$	3	$12a_{0953}$	3
$12a_{0873}$	3	$12a_{0900}$	3	$12a_{0927}$	3	$12a_{0954}$	3
$12a_{0874}$	3	$12a_{0901}$	3	$12a_{0928}$	3	$12a_{0955}$	3
$12a_{0875}$	3	$12a_{0902}$	3	$12a_{0929}$	3	$12a_{0956}$	3
$12a_{0876}$	3	$12a_{0903}$	3	$12a_{0930}$	3	$12a_{0957}$	3
$12a_{0877}$	3	$12a_{0904}$	3	$12a_{0931}$	3	$12a_{0958}$	3
$12a_{0878}$	3	$12a_{0905}$	3	$12a_{0932}$	3	$12a_{0959}$	3
$12a_{0879}$	3	$12a_{0906}$	3	$12a_{0933}$	3	$12a_{0960}$	3
$12a_{0880}$	3	$12a_{0907}$	3	$12a_{0934}$	3	$12a_{0961}$	3
$12a_{0881}$	3	$12a_{0908}$	3	$12a_{0935}$	3	$12a_{0962}$	3
$12a_{0882}$	3	$12a_{0909}$	3	$12a_{0936}$	3	$12a_{0963}$	3
$12a_{0883}$	3	$12a_{0910}$	3	$12a_{0937}$	3	$12a_{0964}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{0965}$	3	$12a_{0992}$	3	$12a_{1019}$	3	$12a_{1046}$	3
$12a_{0966}$	3	$12a_{0993}$	3	$12a_{1020}$	3	$12a_{1047}$	3
$12a_{0967}$	3	$12a_{0994}$	3	$12a_{1021}$	3	$12a_{1048}$	3
$12a_{0968}$	3	$12a_{0995}$	3	$12a_{1022}$	3	$12a_{1049}$	3
$12a_{0969}$	3	$12a_{0996}$	3	$12a_{1023}$	2	$12a_{1050}$	3
$12a_{0970}$	3	$12a_{0997}$	3	$12a_{1024}$	2	$12a_{1051}$	3
$12a_{0971}$	3	$12a_{0998}$	3	$12a_{1025}$	3	$12a_{1052}$	3
$12a_{0972}$	3	$12a_{0999}$	3	$12a_{1026}$	3	$12a_{1053}$	3
$12a_{0973}$	3	$12a_{1000}$	3	$12a_{1027}$	3	$12a_{1054}$	3
$12a_{0974}$	3	$12a_{1001}$	3	$12a_{1028}$	3	$12a_{1055}$	3
$12a_{0975}$	3	$12a_{1002}$	3	$12a_{1029}$	2	$12a_{1056}$	3
$12a_{0976}$	3	$12a_{1003}$	3	$12a_{1030}$	2	$12a_{1057}$	3
$12a_{0977}$	3	$12a_{1004}$	3	$12a_{1031}$	3	$12a_{1058}$	3
$12a_{0978}$	3	$12a_{1005}$	3	$12a_{1032}$	3	$12a_{1059}$	3
$12a_{0979}$	3	$12a_{1006}$	3	$12a_{1033}$	2	$12a_{1060}$	3
$12a_{0980}$	3	$12a_{1007}$	3	$12a_{1034}$	2	$12a_{1061}$	3
$12a_{0981}$	3	$12a_{1008}$	3	$12a_{1035}$	3	$12a_{1062}$	3
$12a_{0982}$	3	$12a_{1009}$	3	$12a_{1036}$	3	$12a_{1063}$	3
$12a_{0983}$	3	$12a_{1010}$	3	$12a_{1037}$	3	$12a_{1064}$	3
$12a_{0984}$	3	$12a_{1011}$	3	$12a_{1038}$	3	$12a_{1065}$	3
$12a_{0985}$	3	$12a_{1012}$	3	$12a_{1039}$	2	$12a_{1066}$	3
$12a_{0986}$	3	$12a_{1013}$	3	$12a_{1040}$	2	$12a_{1067}$	3
$12a_{0987}$	3	$12a_{1014}$	3	$12a_{1041}$	3	$12a_{1068}$	3
$12a_{0988}$	3	$12a_{1015}$	3	$12a_{1042}$	3	$12a_{1069}$	3
$12a_{0989}$	3	$12a_{1016}$	3	$12a_{1043}$	3	$12a_{1070}$	3
$12a_{0990}$	3	$12a_{1017}$	3	$12a_{1044}$	3	$12a_{1071}$	3
$12a_{0991}$	3	$12a_{1018}$	3	$12a_{1045}$	3	$12a_{1072}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{1073}$	3	$12a_{1100}$	3	$12a_{1127}$	2	$12a_{1154}$	3
$12a_{1074}$	3	$12a_{1101}$	3	$12a_{1128}$	2	$12a_{1155}$	3
$12a_{1075}$	3	$12a_{1102}$	3	$12a_{1129}$	2	$12a_{1156}$	3
$12a_{1076}$	3	$12a_{1103}$	3	$12a_{1130}$	2	$12a_{1157}$	2
$12a_{1077}$	3	$12a_{1104}$	3	$12a_{1131}$	2	$12a_{1158}$	2
$12a_{1078}$	3	$12a_{1105}$	3	$12a_{1132}$	2	$12a_{1159}$	2
$12a_{1079}$	3	$12a_{1106}$	3	$12a_{1133}$	2	$12a_{1160}$	3
$12a_{1080}$	3	$12a_{1107}$	3	$12a_{1134}$	2	$12a_{1161}$	2
$12a_{1081}$	3	$12a_{1108}$	3	$12a_{1135}$	2	$12a_{1162}$	2
$12a_{1082}$	3	$12a_{1109}$	3	$12a_{1136}$	2	$12a_{1163}$	2
$12a_{1083}$	3	$12a_{1110}$	3	$12a_{1137}$	3	$12a_{1164}$	3
$12a_{1084}$	3	$12a_{1111}$	3	$12a_{1138}$	2	$12a_{1165}$	2
$12a_{1085}$	3	$12a_{1112}$	3	$12a_{1139}$	2	$12a_{1166}$	2
$12a_{1086}$	3	$12a_{1113}$	3	$12a_{1140}$	2	$12a_{1167}$	3
$12a_{1087}$	3	$12a_{1114}$	3	$12a_{1141}$	3	$12a_{1168}$	3
$12a_{1088}$	3	$12a_{1115}$	3	$12a_{1142}$	3	$12a_{1169}$	3
$12a_{1089}$	3	$12a_{1116}$	3	$12a_{1143}$	3	$12a_{1170}$	3
$12a_{1090}$	3	$12a_{1117}$	3	$12a_{1144}$	3	$12a_{1171}$	3
$12a_{1091}$	3	$12a_{1118}$	3	$12a_{1145}$	2	$12a_{1172}$	3
$12a_{1092}$	3	$12a_{1119}$	3	$12a_{1146}$	2	$12a_{1173}$	3
$12a_{1093}$	3	$12a_{1120}$	3	$12a_{1147}$	3	$12a_{1174}$	3
$12a_{1094}$	3	$12a_{1121}$	3	$12a_{1148}$	2	$12a_{1175}$	3
$12a_{1095}$	3	$12a_{1122}$	3	$12a_{1149}$	2	$12a_{1176}$	3
$12a_{1096}$	3	$12a_{1123}$	3	$12a_{1150}$	3	$12a_{1177}$	3
$12a_{1097}$	3	$12a_{1124}$	3	$12a_{1151}$	3	$12a_{1178}$	3
$12a_{1098}$	3	$12a_{1125}$	2	$12a_{1152}$	3	$12a_{1179}$	3
$12a_{1099}$	3	$12a_{1126}$	2	$12a_{1153}$	3	$12a_{1180}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12a_{1181}$	3	$12a_{1208}$	3	$12a_{1235}$	3	$12a_{1262}$	3
$12a_{1182}$	3	$12a_{1209}$	3	$12a_{1236}$	3	$12a_{1263}$	3
$12a_{1183}$	3	$12a_{1210}$	3	$12a_{1237}$	3	$12a_{1264}$	3
$12a_{1184}$	3	$12a_{1211}$	3	$12a_{1238}$	3	$12a_{1265}$	3
$12a_{1185}$	3	$12a_{1212}$	3	$12a_{1239}$	3	$12a_{1266}$	3
$12a_{1186}$	3	$12a_{1213}$	3	$12a_{1240}$	3	$12a_{1267}$	3
$12a_{1187}$	3	$12a_{1214}$	3	$12a_{1241}$	3	$12a_{1268}$	3
$12a_{1188}$	3	$12a_{1215}$	3	$12a_{1242}$	3	$12a_{1269}$	3
$12a_{1189}$	3	$12a_{1216}$	3	$12a_{1243}$	3	$12a_{1270}$	3
$12a_{1190}$	3	$12a_{1217}$	3	$12a_{1244}$	3	$12a_{1271}$	3
$12a_{1191}$	3	$12a_{1218}$	3	$12a_{1245}$	3	$12a_{1272}$	3
$12a_{1192}$	3	$12a_{1219}$	3	$12a_{1246}$	3	$12a_{1273}$	2
$12a_{1193}$	3	$12a_{1220}$	3	$12a_{1247}$	3	$12a_{1274}$	2
$12a_{1194}$	3	$12a_{1221}$	3	$12a_{1248}$	3	$12a_{1275}$	2
$12a_{1195}$	3	$12a_{1222}$	3	$12a_{1249}$	3	$12a_{1276}$	2
$12a_{1196}$	3	$12a_{1223}$	3	$12a_{1250}$	3	$12a_{1277}$	2
$12a_{1197}$	3	$12a_{1224}$	3	$12a_{1251}$	3	$12a_{1278}$	2
$12a_{1198}$	3	$12a_{1225}$	3	$12a_{1252}$	3	$12a_{1279}$	2
$12a_{1199}$	3	$12a_{1226}$	3	$12a_{1253}$	3	$12a_{1280}$	3
$12a_{1200}$	3	$12a_{1227}$	3	$12a_{1254}$	3	$12a_{1281}$	2
$12a_{1201}$	3	$12a_{1228}$	3	$12a_{1255}$	3	$12a_{1282}$	2
$12a_{1202}$	3	$12a_{1229}$	3	$12a_{1256}$	3	$12a_{1283}$	3
$12a_{1203}$	3	$12a_{1230}$	3	$12a_{1257}$	3	$12a_{1284}$	3
$12a_{1204}$	3	$12a_{1231}$	3	$12a_{1258}$	3	$12a_{1285}$	3
$12a_{1205}$	3	$12a_{1232}$	3	$12a_{1259}$	3	$12a_{1286}$	3
$12a_{1206}$	3	$12a_{1233}$	3	$12a_{1260}$	3	$12a_{1287}$	2
$12a_{1207}$	3	$12a_{1234}$	3	$12a_{1261}$	3	$12a_{1288}$	3

Table B.6: Computed bridge index of non-alternating prime knots with 12 crossings

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0001}$	3	$12n_{0026}$	3	$12n_{0051}$	3	$12n_{0076}$	3
$12n_{0002}$	3	$12n_{0027}$	3	$12n_{0052}$	3	$12n_{0077}$	3
$12n_{0003}$	3	$12n_{0028}$	3	$12n_{0053}$	3	$12n_{0078}$	3
$12n_{0004}$	3	$12n_{0029}$	3	$12n_{0054}$	3	$12n_{0079}$	3
$12n_{0005}$	3	$12n_{0030}$	3	$12n_{0055}$	4	$12n_{0080}$	3
$12n_{0006}$	3	$12n_{0031}$	3	$12n_{0056}$	4	$12n_{0081}$	3
$12n_{0007}$	3	$12n_{0032}$	3	$12n_{0057}$	4	$12n_{0082}$	3
$12n_{0008}$	3	$12n_{0033}$	3	$12n_{0058}$	4	$12n_{0083}$	3
$12n_{0009}$	3	$12n_{0034}$	3	$12n_{0059}$	4	$12n_{0084}$	3
$12n_{0010}$	3	$12n_{0035}$	3	$12n_{0060}$	4	$12n_{0085}$	3
$12n_{0011}$	3	$12n_{0036}$	3	$12n_{0061}$	4	$12n_{0086}$	3
$12n_{0012}$	3	$12n_{0037}$	3	$12n_{0062}$	4	$12n_{0087}$	3
$12n_{0013}$	3	$12n_{0038}$	3	$12n_{0063}$	4	$12n_{0088}$	3
$12n_{0014}$	3	$12n_{0039}$	3	$12n_{0064}$	4	$12n_{0089}$	3
$12n_{0015}$	3	$12n_{0040}$	3	$12n_{0065}$	3	$12n_{0090}$	3
$12n_{0016}$	3	$12n_{0041}$	3	$12n_{0066}$	4	$12n_{0091}$	3
$12n_{0017}$	3	$12n_{0042}$	3	$12n_{0067}$	4	$12n_{0092}$	3
$12n_{0018}$	3	$12n_{0043}$	3	$12n_{0068}$	3	$12n_{0093}$	3
$12n_{0019}$	3	$12n_{0044}$	3	$12n_{0069}$	3	$12n_{0094}$	3
$12n_{0020}$	3	$12n_{0045}$	3	$12n_{0070}$	3	$12n_{0095}$	3
$12n_{0021}$	3	$12n_{0046}$	3	$12n_{0071}$	3	$12n_{0096}$	3
$12n_{0022}$	3	$12n_{0047}$	3	$12n_{0072}$	3	$12n_{0097}$	3
$12n_{0023}$	3	$12n_{0048}$	3	$12n_{0073}$	3	$12n_{0098}$	3
$12n_{0024}$	3	$12n_{0049}$	3	$12n_{0074}$	3	$12n_{0099}$	3
$12n_{0025}$	3	$12n_{0050}$	3	$12n_{0075}$	3	$12n_{0100}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0101}$	3	$12n_{0128}$	3	$12n_{0155}$	3	$12n_{0182}$	3
$12n_{0102}$	3	$12n_{0129}$	3	$12n_{0156}$	3	$12n_{0183}$	3
$12n_{0103}$	3	$12n_{0130}$	3	$12n_{0157}$	3	$12n_{0184}$	3
$12n_{0104}$	3	$12n_{0131}$	3	$12n_{0158}$	3	$12n_{0185}$	3
$12n_{0105}$	3	$12n_{0132}$	3	$12n_{0159}$	3	$12n_{0186}$	3
$12n_{0106}$	3	$12n_{0133}$	3	$12n_{0160}$	3	$12n_{0187}$	3
$12n_{0107}$	3	$12n_{0134}$	3	$12n_{0161}$	3	$12n_{0188}$	3
$12n_{0108}$	3	$12n_{0135}$	3	$12n_{0162}$	3	$12n_{0189}$	3
$12n_{0109}$	3	$12n_{0136}$	3	$12n_{0163}$	3	$12n_{0190}$	3
$12n_{0110}$	3	$12n_{0137}$	3	$12n_{0164}$	3	$12n_{0191}$	3
$12n_{0111}$	3	$12n_{0138}$	3	$12n_{0165}$	3	$12n_{0192}$	3
$12n_{0112}$	3	$12n_{0139}$	3	$12n_{0166}$	3	$12n_{0193}$	3
$12n_{0113}$	3	$12n_{0140}$	3	$12n_{0167}$	3	$12n_{0194}$	3
$12n_{0114}$	3	$12n_{0141}$	3	$12n_{0168}$	3	$12n_{0195}$	3
$12n_{0115}$	3	$12n_{0142}$	3	$12n_{0169}$	3	$12n_{0196}$	3
$12n_{0116}$	3	$12n_{0143}$	3	$12n_{0170}$	3	$12n_{0197}$	3
$12n_{0117}$	3	$12n_{0144}$	3	$12n_{0171}$	3	$12n_{0198}$	3
$12n_{0118}$	3	$12n_{0145}$	3	$12n_{0172}$	3	$12n_{0199}$	3
$12n_{0119}$	3	$12n_{0146}$	3	$12n_{0173}$	3	$12n_{0200}$	3
$12n_{0120}$	3	$12n_{0147}$	3	$12n_{0174}$	3	$12n_{0201}$	3
$12n_{0121}$	3	$12n_{0148}$	3	$12n_{0175}$	3	$12n_{0202}$	3
$12n_{0122}$	3	$12n_{0149}$	3	$12n_{0176}$	3	$12n_{0203}$	3
$12n_{0123}$	3	$12n_{0150}$	3	$12n_{0177}$	3	$12n_{0204}$	3
$12n_{0124}$	3	$12n_{0151}$	3	$12n_{0178}$	3	$12n_{0205}$	3
$12n_{0125}$	3	$12n_{0152}$	3	$12n_{0179}$	3	$12n_{0206}$	3
$12n_{0126}$	3	$12n_{0153}$	3	$12n_{0180}$	3	$12n_{0207}$	3
$12n_{0127}$	3	$12n_{0154}$	3	$12n_{0181}$	3	$12n_{0208}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0209}$	3	$12n_{0236}$	3	$12n_{0263}$	3	$12n_{0290}$	3
$12n_{0210}$	3	$12n_{0237}$	3	$12n_{0264}$	3	$12n_{0291}$	3
$12n_{0211}$	3	$12n_{0238}$	3	$12n_{0265}$	3	$12n_{0292}$	3
$12n_{0212}$	3	$12n_{0239}$	3	$12n_{0266}$	3	$12n_{0293}$	3
$12n_{0213}$	3	$12n_{0240}$	3	$12n_{0267}$	3	$12n_{0294}$	3
$12n_{0214}$	3	$12n_{0241}$	3	$12n_{0268}$	3	$12n_{0295}$	3
$12n_{0215}$	3	$12n_{0242}$	3	$12n_{0269}$	3	$12n_{0296}$	3
$12n_{0216}$	3	$12n_{0243}$	3	$12n_{0270}$	3	$12n_{0297}$	3
$12n_{0217}$	3	$12n_{0244}$	3	$12n_{0271}$	3	$12n_{0298}$	3
$12n_{0218}$	3	$12n_{0245}$	3	$12n_{0272}$	3	$12n_{0299}$	3
$12n_{0219}$	4	$12n_{0246}$	3	$12n_{0273}$	3	$12n_{0300}$	3
$12n_{0220}$	4	$12n_{0247}$	3	$12n_{0274}$	3	$12n_{0301}$	3
$12n_{0221}$	4	$12n_{0248}$	3	$12n_{0275}$	3	$12n_{0302}$	3
$12n_{0222}$	4	$12n_{0249}$	3	$12n_{0276}$	3	$12n_{0303}$	3
$12n_{0223}$	4	$12n_{0250}$	3	$12n_{0277}$	3	$12n_{0304}$	3
$12n_{0224}$	4	$12n_{0251}$	3	$12n_{0278}$	3	$12n_{0305}$	3
$12n_{0225}$	4	$12n_{0252}$	3	$12n_{0279}$	3	$12n_{0306}$	3
$12n_{0226}$	3	$12n_{0253}$	3	$12n_{0280}$	3	$12n_{0307}$	3
$12n_{0227}$	3	$12n_{0254}$	3	$12n_{0281}$	3	$12n_{0308}$	3
$12n_{0228}$	3	$12n_{0255}$	3	$12n_{0282}$	3	$12n_{0309}$	3
$12n_{0229}$	4	$12n_{0256}$	3	$12n_{0283}$	3	$12n_{0310}$	3
$12n_{0230}$	3	$12n_{0257}$	3	$12n_{0284}$	3	$12n_{0311}$	3
$12n_{0231}$	3	$12n_{0258}$	3	$12n_{0285}$	3	$12n_{0312}$	3
$12n_{0232}$	3	$12n_{0259}$	3	$12n_{0286}$	3	$12n_{0313}$	3
$12n_{0233}$	3	$12n_{0260}$	3	$12n_{0287}$	3	$12n_{0314}$	3
$12n_{0234}$	3	$12n_{0261}$	4	$12n_{0288}$	3	$12n_{0315}$	3
$12n_{0235}$	3	$12n_{0262}$	3	$12n_{0289}$	3	$12n_{0316}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0317}$	3	$12n_{0344}$	3	$12n_{0371}$	3	$12n_{0398}$	3
$12n_{0318}$	3	$12n_{0345}$	3	$12n_{0372}$	3	$12n_{0399}$	3
$12n_{0319}$	3	$12n_{0346}$	3	$12n_{0373}$	3	$12n_{0400}$	3
$12n_{0320}$	3	$12n_{0347}$	3	$12n_{0374}$	3	$12n_{0401}$	3
$12n_{0321}$	3	$12n_{0348}$	3	$12n_{0375}$	3	$12n_{0402}$	3
$12n_{0322}$	3	$12n_{0349}$	3	$12n_{0376}$	3	$12n_{0403}$	3
$12n_{0323}$	3	$12n_{0350}$	3	$12n_{0377}$	3	$12n_{0404}$	3
$12n_{0324}$	3	$12n_{0351}$	3	$12n_{0378}$	3	$12n_{0405}$	3
$12n_{0325}$	3	$12n_{0352}$	3	$12n_{0379}$	3	$12n_{0406}$	3
$12n_{0326}$	3	$12n_{0353}$	3	$12n_{0380}$	3	$12n_{0407}$	3
$12n_{0327}$	3	$12n_{0354}$	3	$12n_{0381}$	3	$12n_{0408}$	3
$12n_{0328}$	3	$12n_{0355}$	3	$12n_{0382}$	3	$12n_{0409}$	3
$12n_{0329}$	3	$12n_{0356}$	3	$12n_{0383}$	3	$12n_{0410}$	3
$12n_{0330}$	3	$12n_{0357}$	3	$12n_{0384}$	3	$12n_{0411}$	3
$12n_{0331}$	3	$12n_{0358}$	3	$12n_{0385}$	3	$12n_{0412}$	3
$12n_{0332}$	3	$12n_{0359}$	3	$12n_{0386}$	3	$12n_{0413}$	3
$12n_{0333}$	3	$12n_{0360}$	3	$12n_{0387}$	3	$12n_{0414}$	3
$12n_{0334}$	3	$12n_{0361}$	3	$12n_{0388}$	3	$12n_{0415}$	3
$12n_{0335}$	3	$12n_{0362}$	3	$12n_{0389}$	3	$12n_{0416}$	3
$12n_{0336}$	3	$12n_{0363}$	3	$12n_{0390}$	3	$12n_{0417}$	3
$12n_{0337}$	3	$12n_{0364}$	3	$12n_{0391}$	3	$12n_{0418}$	3
$12n_{0338}$	3	$12n_{0365}$	3	$12n_{0392}$	3	$12n_{0419}$	3
$12n_{0339}$	3	$12n_{0366}$	3	$12n_{0393}$	3	$12n_{0420}$	3
$12n_{0340}$	3	$12n_{0367}$	3	$12n_{0394}$	3	$12n_{0421}$	3
$12n_{0341}$	3	$12n_{0368}$	3	$12n_{0395}$	3	$12n_{0422}$	3
$12n_{0342}$	3	$12n_{0369}$	3	$12n_{0396}$	3	$12n_{0423}$	3
$12n_{0343}$	3	$12n_{0370}$	3	$12n_{0397}$	3	$12n_{0424}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0425}$	3	$12n_{0452}$	3	$12n_{0479}$	3	$12n_{0506}$	3
$12n_{0426}$	3	$12n_{0453}$	3	$12n_{0480}$	3	$12n_{0507}$	3
$12n_{0427}$	3	$12n_{0454}$	3	$12n_{0481}$	3	$12n_{0508}$	3
$12n_{0428}$	3	$12n_{0455}$	3	$12n_{0482}$	3	$12n_{0509}$	3
$12n_{0429}$	3	$12n_{0456}$	3	$12n_{0483}$	3	$12n_{0510}$	3
$12n_{0430}$	3	$12n_{0457}$	3	$12n_{0484}$	3	$12n_{0511}$	3
$12n_{0431}$	3	$12n_{0458}$	3	$12n_{0485}$	3	$12n_{0512}$	3
$12n_{0432}$	3	$12n_{0459}$	3	$12n_{0486}$	3	$12n_{0513}$	3
$12n_{0433}$	3	$12n_{0460}$	3	$12n_{0487}$	3	$12n_{0514}$	3
$12n_{0434}$	3	$12n_{0461}$	3	$12n_{0488}$	3	$12n_{0515}$	3
$12n_{0435}$	3	$12n_{0462}$	3	$12n_{0489}$	3	$12n_{0516}$	3
$12n_{0436}$	3	$12n_{0463}$	3	$12n_{0490}$	3	$12n_{0517}$	3
$12n_{0437}$	3	$12n_{0464}$	3	$12n_{0491}$	3	$12n_{0518}$	3
$12n_{0438}$	3	$12n_{0465}$	3	$12n_{0492}$	3	$12n_{0519}$	3
$12n_{0439}$	3	$12n_{0466}$	3	$12n_{0493}$	3	$12n_{0520}$	3
$12n_{0440}$	3	$12n_{0467}$	3	$12n_{0494}$	3	$12n_{0521}$	3
$12n_{0441}$	3	$12n_{0468}$	3	$12n_{0495}$	3	$12n_{0522}$	3
$12n_{0442}$	3	$12n_{0469}$	3	$12n_{0496}$	3	$12n_{0523}$	3
$12n_{0443}$	3	$12n_{0470}$	3	$12n_{0497}$	3	$12n_{0524}$	3
$12n_{0444}$	3	$12n_{0471}$	3	$12n_{0498}$	3	$12n_{0525}$	3
$12n_{0445}$	3	$12n_{0472}$	3	$12n_{0499}$	3	$12n_{0526}$	3
$12n_{0446}$	3	$12n_{0473}$	3	$12n_{0500}$	3	$12n_{0527}$	3
$12n_{0447}$	3	$12n_{0474}$	3	$12n_{0501}$	3	$12n_{0528}$	3
$12n_{0448}$	3	$12n_{0475}$	3	$12n_{0502}$	3	$12n_{0529}$	3
$12n_{0449}$	3	$12n_{0476}$	3	$12n_{0503}$	3	$12n_{0530}$	3
$12n_{0450}$	3	$12n_{0477}$	3	$12n_{0504}$	3	$12n_{0531}$	3
$12n_{0451}$	3	$12n_{0478}$	3	$12n_{0505}$	3	$12n_{0532}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0533}$	3	$12n_{0560}$	3	$12n_{0587}$	3	$12n_{0614}$	3
$12n_{0534}$	3	$12n_{0561}$	3	$12n_{0588}$	3	$12n_{0615}$	3
$12n_{0535}$	3	$12n_{0562}$	3	$12n_{0589}$	3	$12n_{0616}$	3
$12n_{0536}$	3	$12n_{0563}$	3	$12n_{0590}$	3	$12n_{0617}$	3
$12n_{0537}$	3	$12n_{0564}$	3	$12n_{0591}$	3	$12n_{0618}$	3
$12n_{0538}$	3	$12n_{0565}$	3	$12n_{0592}$	3	$12n_{0619}$	3
$12n_{0539}$	3	$12n_{0566}$	3	$12n_{0593}$	3	$12n_{0620}$	3
$12n_{0540}$	3	$12n_{0567}$	3	$12n_{0594}$	3	$12n_{0621}$	3
$12n_{0541}$	3	$12n_{0568}$	3	$12n_{0595}$	3	$12n_{0622}$	3
$12n_{0542}$	3	$12n_{0569}$	3	$12n_{0596}$	3	$12n_{0623}$	3
$12n_{0543}$	3	$12n_{0570}$	3	$12n_{0597}$	3	$12n_{0624}$	3
$12n_{0544}$	3	$12n_{0571}$	3	$12n_{0598}$	3	$12n_{0625}$	3
$12n_{0545}$	3	$12n_{0572}$	3	$12n_{0599}$	3	$12n_{0626}$	3
$12n_{0546}$	3	$12n_{0573}$	3	$12n_{0600}$	3	$12n_{0627}$	3
$12n_{0547}$	3	$12n_{0574}$	3	$12n_{0601}$	3	$12n_{0628}$	3
$12n_{0548}$	3	$12n_{0575}$	3	$12n_{0602}$	3	$12n_{0629}$	3
$12n_{0549}$	3	$12n_{0576}$	3	$12n_{0603}$	3	$12n_{0630}$	3
$12n_{0550}$	3	$12n_{0577}$	3	$12n_{0604}$	3	$12n_{0631}$	3
$12n_{0551}$	3	$12n_{0578}$	3	$12n_{0605}$	3	$12n_{0632}$	3
$12n_{0552}$	3	$12n_{0579}$	3	$12n_{0606}$	3	$12n_{0633}$	3
$12n_{0553}$	4	$12n_{0580}$	3	$12n_{0607}$	3	$12n_{0634}$	3
$12n_{0554}$	4	$12n_{0581}$	3	$12n_{0608}$	3	$12n_{0635}$	3
$12n_{0555}$	4	$12n_{0582}$	3	$12n_{0609}$	3	$12n_{0636}$	3
$12n_{0556}$	4	$12n_{0583}$	3	$12n_{0610}$	3	$12n_{0637}$	3
$12n_{0557}$	3	$12n_{0584}$	3	$12n_{0611}$	3	$12n_{0638}$	3
$12n_{0558}$	3	$12n_{0585}$	3	$12n_{0612}$	3	$12n_{0639}$	3
$12n_{0559}$	3	$12n_{0586}$	3	$12n_{0613}$	3	$12n_{0640}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0641}$	3	$12n_{0668}$	3	$12n_{0695}$	3	$12n_{0722}$	3
$12n_{0642}$	4	$12n_{0669}$	3	$12n_{0696}$	3	$12n_{0723}$	3
$12n_{0643}$	3	$12n_{0670}$	3	$12n_{0697}$	3	$12n_{0724}$	3
$12n_{0644}$	3	$12n_{0671}$	3	$12n_{0698}$	3	$12n_{0725}$	3
$12n_{0645}$	3	$12n_{0672}$	3	$12n_{0699}$	3	$12n_{0726}$	3
$12n_{0646}$	3	$12n_{0673}$	3	$12n_{0700}$	3	$12n_{0727}$	3
$12n_{0647}$	3	$12n_{0674}$	3	$12n_{0701}$	3	$12n_{0728}$	3
$12n_{0648}$	3	$12n_{0675}$	3	$12n_{0702}$	3	$12n_{0729}$	3
$12n_{0649}$	3	$12n_{0676}$	3	$12n_{0703}$	3	$12n_{0730}$	3
$12n_{0650}$	3	$12n_{0677}$	3	$12n_{0704}$	3	$12n_{0731}$	3
$12n_{0651}$	3	$12n_{0678}$	3	$12n_{0705}$	3	$12n_{0732}$	3
$12n_{0652}$	3	$12n_{0679}$	3	$12n_{0706}$	3	$12n_{0733}$	3
$12n_{0653}$	3	$12n_{0680}$	3	$12n_{0707}$	3	$12n_{0734}$	3
$12n_{0654}$	3	$12n_{0681}$	3	$12n_{0708}$	3	$12n_{0735}$	3
$12n_{0655}$	3	$12n_{0682}$	3	$12n_{0709}$	3	$12n_{0736}$	3
$12n_{0656}$	3	$12n_{0683}$	3	$12n_{0710}$	3	$12n_{0737}$	3
$12n_{0657}$	3	$12n_{0684}$	3	$12n_{0711}$	3	$12n_{0738}$	3
$12n_{0658}$	3	$12n_{0685}$	3	$12n_{0712}$	3	$12n_{0739}$	3
$12n_{0659}$	3	$12n_{0686}$	3	$12n_{0713}$	3	$12n_{0740}$	3
$12n_{0660}$	3	$12n_{0687}$	3	$12n_{0714}$	3	$12n_{0741}$	3
$12n_{0661}$	3	$12n_{0688}$	3	$12n_{0715}$	3	$12n_{0742}$	3
$12n_{0662}$	3	$12n_{0689}$	3	$12n_{0716}$	3	$12n_{0743}$	3
$12n_{0663}$	3	$12n_{0690}$	3	$12n_{0717}$	3	$12n_{0744}$	3
$12n_{0664}$	3	$12n_{0691}$	3	$12n_{0718}$	3	$12n_{0745}$	3
$12n_{0665}$	3	$12n_{0692}$	3	$12n_{0719}$	3	$12n_{0746}$	3
$12n_{0666}$	3	$12n_{0693}$	3	$12n_{0720}$	3	$12n_{0747}$	3
$12n_{0667}$	3	$12n_{0694}$	3	$12n_{0721}$	3	$12n_{0748}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0749}$	3	$12n_{0776}$	3	$12n_{0803}$	3	$12n_{0830}$	3
$12n_{0750}$	3	$12n_{0777}$	3	$12n_{0804}$	3	$12n_{0831}$	3
$12n_{0751}$	3	$12n_{0778}$	3	$12n_{0805}$	3	$12n_{0832}$	3
$12n_{0752}$	3	$12n_{0779}$	3	$12n_{0806}$	3	$12n_{0833}$	3
$12n_{0753}$	3	$12n_{0780}$	3	$12n_{0807}$	3	$12n_{0834}$	3
$12n_{0754}$	3	$12n_{0781}$	3	$12n_{0808}$	3	$12n_{0835}$	3
$12n_{0755}$	3	$12n_{0782}$	3	$12n_{0809}$	3	$12n_{0836}$	3
$12n_{0756}$	3	$12n_{0783}$	3	$12n_{0810}$	3	$12n_{0837}$	3
$12n_{0757}$	3	$12n_{0784}$	3	$12n_{0811}$	3	$12n_{0838}$	3
$12n_{0758}$	3	$12n_{0785}$	3	$12n_{0812}$	3	$12n_{0839}$	3
$12n_{0759}$	3	$12n_{0786}$	3	$12n_{0813}$	3	$12n_{0840}$	3
$12n_{0760}$	3	$12n_{0787}$	3	$12n_{0814}$	3	$12n_{0841}$	3
$12n_{0761}$	3	$12n_{0788}$	3	$12n_{0815}$	3	$12n_{0842}$	3
$12n_{0762}$	3	$12n_{0789}$	3	$12n_{0816}$	3	$12n_{0843}$	3
$12n_{0763}$	3	$12n_{0790}$	3	$12n_{0817}$	3	$12n_{0844}$	3
$12n_{0764}$	3	$12n_{0791}$	3	$12n_{0818}$	3	$12n_{0845}$	3
$12n_{0765}$	3	$12n_{0792}$	3	$12n_{0819}$	3	$12n_{0846}$	3
$12n_{0766}$	3	$12n_{0793}$	3	$12n_{0820}$	3	$12n_{0847}$	3
$12n_{0767}$	3	$12n_{0794}$	3	$12n_{0821}$	3	$12n_{0848}$	3
$12n_{0768}$	3	$12n_{0795}$	3	$12n_{0822}$	3	$12n_{0849}$	3
$12n_{0769}$	3	$12n_{0796}$	3	$12n_{0823}$	3	$12n_{0850}$	3
$12n_{0770}$	3	$12n_{0797}$	3	$12n_{0824}$	3	$12n_{0851}$	3
$12n_{0771}$	3	$12n_{0798}$	3	$12n_{0825}$	3	$12n_{0852}$	3
$12n_{0772}$	3	$12n_{0799}$	3	$12n_{0826}$	3	$12n_{0853}$	3
$12n_{0773}$	3	$12n_{0800}$	3	$12n_{0827}$	3	$12n_{0854}$	3
$12n_{0774}$	3	$12n_{0801}$	3	$12n_{0828}$	3	$12n_{0855}$	3
$12n_{0775}$	3	$12n_{0802}$	3	$12n_{0829}$	3	$12n_{0856}$	3

(table continues)

K	$b(K)$	K	$b(K)$	K	$b(K)$	K	$b(K)$
$12n_{0857}$	3	$12n_{0865}$	3	$12n_{0873}$	3	$12n_{0881}$	3
$12n_{0858}$	3	$12n_{0866}$	3	$12n_{0874}$	3	$12n_{0882}$	3
$12n_{0859}$	3	$12n_{0867}$	3	$12n_{0875}$	3	$12n_{0883}$	3
$12n_{0860}$	3	$12n_{0868}$	3	$12n_{0876}$	3	$12n_{0884}$	3
$12n_{0861}$	3	$12n_{0869}$	3	$12n_{0877}$	3	$12n_{0885}$	3
$12n_{0862}$	3	$12n_{0870}$	3	$12n_{0878}$	3	$12n_{0886}$	3
$12n_{0863}$	3	$12n_{0871}$	3	$12n_{0879}$	3	$12n_{0887}$	3
$12n_{0864}$	3	$12n_{0872}$	3	$12n_{0880}$	3	$12n_{0888}$	3

APPENDIX C
CODE USED TO COMPUTE THE BRIDGE INDEXES

Following is the code we used to compute the upper bound on the bridge indexes. The formatting of the code has been adjusting slightly to improve readability in print format.

C.1 bridge_computation.py

```
#!/usr/bin/env python2.7

import ast
import csv
import json
import logging
import sys, getopt, os
from reduce_bridges import *

logging.basicConfig(filename='bridge_computation.log', filemode='w',
    format='%(asctime)s: %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p',
    level=logging.WARNING)

def bridge_computation(argv):
    inputfile = ''
    outputdir = 'output'
    try:
        opts, args = getopt.getopt(argv,"hi:o:",["inputfile=", "outputdir",])
    except getopt.GetoptError:
        print 'bridge_computation.py -i <inputfile> -o <outputdir>'
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print 'bridge_computation.py -i <inputfile> -o <outputdir>'
            sys.exit()
        elif opt in ("-i", "--inputfile"):
            inputfile = arg
```

```
        elif opt in ("-o", "--outputdir"):
            outputdir = arg

    # Create a directory for outputs.
    if not os.path.exists(outputdir):
        os.makedirs(outputdir)

    if os.path.isdir(inputfile):
        # Traverse the directory to process all csv files.
        for root, dirs, files in os.walk(inputfile):
            for file in files:
                if file.endswith(".csv"):
                    try:
                        calculate_bridge_index(
                            os.path.join(root, file), outputdir)
                    except:
                        logging.error('Failed to fully process ' + str(file))
                        print 'Failed to fully process ' + str(file)
    elif os.path.isfile(inputfile):
        calculate_bridge_index(inputfile, outputdir)
    else:
        input_message = ' '.join([
            'The specified input is not a file or a directory.',
            'Please try a different input.'])
        print input_message
        logging.warning(input_message)

def calculate_bridge_index(inputfile, outputdir):
    # Read in a CSV.
    with open(inputfile) as csvfile:
        fieldnames = ['name', 'pd_notation']
        knotreader = csv.DictReader(csvfile)
        for row in knotreader:
            # Create a file to store the output of all trees of this knot.
            outfile_name = outputdir + '/' + row['name'] + '_output.csv'
            with open(outfile_name, "w") as outfile:
                outputwriter = csv.writer(outfile, delimiter=',')
                outputwriter.writerow(['name','computed_bridge_index'])
            try:
                # Create a knot object.
```

```
                knot = create_knot_from_pd_code(
                    ast.literal_eval(row['pd_notation']), row['name'])
                logging.info('Processing knot ' + str(knot.name))
                logging.debug('The initial PD code of the knot is ' + str(knot))
                # Simplify the knot now to avoid choosing bridges which will be
                # discarded during simplification.
                knot.simplify_rm1_rm2_recursively()
                if knot.free_crossings != []:
                    base_knot_name = row['name']
                    directory = 'knot_trees/' + base_knot_name
                    knot.list_bridge_ts(directory, 0)
                    for subdir, dirs, files in os.walk(directory):
                        more_to_process = True
                        depth_to_process = 0
                        while more_to_process == True:
                            more_to_process = process_tree_with_depth(subdir,
                                depth_to_process, outfile_name)
                            depth_to_process += 1
                        break
                else:
                    write_output(knot, outfile_name)
            except:
                print 'Failed to fully process the knot. Moving on to the next knot'
                warning_message = ' '.join([
                    'Failed to fully process', str(knot.name) + '.',
                    'Moving on to the next knot.'])
                logging.warning(warning_message)
                continue

def process_tree_with_depth(directory, depth, outfile_name):
    more_to_process = False
    for subdir, dirs, files in os.walk(directory):
        for file in files:
            # If file name ends in "_" + depth + ".csv", open the file.
            if (depth == int(file.rsplit('_', 1)[-1].rsplit('.', 1)[0])):
                file_path = os.path.join(subdir, file)
                with open(file_path) as treecsvfile:
                    treereader = csv.DictReader(treecsvfile)
                    for tree in treereader:
                        knot = create_knot_from_pd_code(
```

```
                        ast.literal_eval(tree['pd_notation']),
                        tree['name'],
                        ast.literal_eval(tree['bridges']))
                    while knot.free_crossings != []:
                        try:
                            # Drag underpasses & simplify
                            # until no moves are possible.
                            args = knot.find_crossing_to_drag()
                            knot.drag_crossing_under_bridge_resursively(*args)
                            knot.simplify_rm1_rm2_recursively()
                        except:
                            break
                    if knot.free_crossings == []:
                        write_output(knot, outfile_name)
                    else:
                        knot.list_bridge_ts(subdir, depth + 1)
                        more_to_process = True
    return more_to_process

def write_output(knot, outfile_name):
    computed_bridge_index = len(knot.bridges)
    info_message = ' '.join([
        'Finished processing', str(knot.name) + '.',
        'The final bridge number is', str(computed_bridge_index)])
    logging.info(info_message)
    logging.debug('The final PD code of ' + str(knot.name) + ' is ' + str(knot))
    # Add the results to our output file.
    try:
        with open(outfile_name, "a") as outfile:
            outputwriter = csv.writer(outfile, delimiter=',')
            outputwriter.writerow([knot.name, computed_bridge_index])
    except IOError:
        error_message = ' '.join([
            'Cannot write output file.'
            'Be sure the directory "outputs" exists and is writeable.'])
        sys.exit(error_message)

if __name__ == "__main__":
    bridge_computation(sys.argv[1:])
```

C.2 reduce_bridges.py

```
#!/usr/bin/env python2.7

import itertools
from itertools import repeat
import logging
import numpy
import sys, os, csv, copy

class Crossing:
    def __init__(self, pd_code, bridge = None):
        self.pd_code = pd_code
        self.bridge = bridge

    def __eq__(self, other):
        return self.pd_code == other.pd_code and self.bridge == other.bridge

    def __hash__(self):
        return hash(tuple(self.pd_code))

    def __str__(self):
        return str([self.pd_code, self.bridge])

    def alter_elements_greater_than(self, value, addend, maximum = None):
        """
        Change the value of all elements in a Crossing which are greater
        than the provided value.

        Arguments:
        value -- (int) The number to compare each element of the crossing with.
        addend -- (int) The number to add to crossing elements greater than value.
        maximum -- (int) The maximum allowed value of elements in the crossing.
        """
        self.pd_code = [alter_if_greater(x, value, addend, maximum) for x in
            self.pd_code]
        return self

    def alter_for_drag(self, ordered_segments):
```

```
        self.pd_code = [
            alter_element_for_drag(x, ordered_segments[0],
                ordered_segments[1]) for x in self.pd_code]
        return self

    def has_duplicate_value(self):
        """
        Determine if there are duplicate values in the PD notation of a crossing.
        """
        sets = reduce(
            lambda (u, d), o : (u.union([o]), d.union(u.intersection([o]))),
            self.pd_code,
            (set(), set()))
        if sets[1]:
            return list(sets[1])[0]
        else:
            return False

    def overpass_traveled_from(self):
        """
        Find the value of the overcross segment of a crossing we travel from
        toward the other.
        """
        e,f,g,h = self.pd_code
        if abs(f - h) == 1:
            return min(f, h)
        else:
            return max(f, h)

class Knot:
    def __init__(self, crossings, name = None, bridges = None):
        self.name = name
        self.crossings = crossings # crossings is a list of Crossing objects
        self.free_crossings = crossings[:]
        self.bridges = {}
        if bridges:
            for bridge in bridges.itervalues():
                bridge_end = bridge[0]
                for free_crossing in self.free_crossings:
                    count = free_crossing.pd_code.count(bridge_end)
```

```
                if count == 1:
                    i = free_crossing.pd_code.index(bridge_end)
                    if ((i == 1) or (i == 3)):
                        self.designate_bridge(free_crossing)
                        break
                elif count == 2:
                    self.designate_bridge(free_crossing)
                    break

def __eq__(self, other):
    return self.crossings == other.crossings

def __str__(self):
    return str([crossing.pd_code for crossing in self.crossings])

def alter_bridge_segments_greater_than(self, value, addend, maximum = None):
    """
    Change the value of the bridge end segments if they are greater
    than the provided value.

    Arguments:
    value -- (int) The number to compare each segment with.
    addend -- (int) The number to add to the segments greater than value.
    maximum -- (int) The maximum allowed value of segments in the bridge.
    """
    for bridge_index, bridge in self.bridges.iteritems():
        for x in bridge:
            x_index = bridge.index(x)
            self.bridges[bridge_index][x_index] = alter_if_greater(x, value,
                                                                    addend,
                                                                    maximum)
    return self

def bridge_crossings(self):
    return diff(self.crossings, self.free_crossings)

def delete_bridge(self, bridge_key):
    for crossing in self.bridge_crossings():
        if (crossing.bridge == bridge_key):
            crossing.bridge = None
```

```
                self.free_crossings.append(crossing)
        del(self.bridges[bridge_key])
        logging.debug('The bridge with key ' + str(bridge_key)
            + ' has been deleted.')
        return self

    def delete_crossings(self, indices):
        """
        Delete crossings from a knot.
        This removes objects from both knot.crossings and knot.free_crossings.

        Arguments:
        indices -- (list) the indices of the crossings to delete
        """
        # Delete crossings from last to first to avoid changing
        # the index of crossings not yet processed.
        indices.sort(reverse = True)
        for index in indices:
            del self.crossings[index]
        self.free_crossings = list(
            set(self.crossings).intersection(self.free_crossings))
        return self

    def designate_additional_bridge(self):
        """
        Choose a crossing to designate as a bridge based on existing bridges.
        """
        bridge_crossings = self.bridge_crossings()
        bridge_ends = [x for bridge_ends in self.bridges.itervalues()
            for x in bridge_ends]
        all_bridge_segments = [crossing.pd_code[i] for crossing in bridge_crossings
            for i in [0, 2]]
        bridge_interior_segments = diff(all_bridge_segments, bridge_ends)

        for free_crossing in self.free_crossings:
            interior_match = list(
                set([free_crossing.pd_code[1], free_crossing.pd_code[3]])
                & set(bridge_interior_segments))
            end_match = list(set(bridge_ends) & set(free_crossing.pd_code))
            if (interior_match or end_match):
```

```
                self.designate_bridge(free_crossing)
                return self

        logging.critical('We were unable to designate an additional bridge.')
        sys.exit('We were unable to designate an additional bridge for '
            + self.name + '.')

    def designate_bridge(self, crossing):
        """
        Identify a crossing as a bridge and extend until it deadends.

        Arguments:
        crossing -- (obj) a crossing
        """
        # Determine the key for this bridge.
        bridge_keys = self.bridges.keys()
        if (bridge_keys):
            key = max(bridge_keys) + 1
        else:
            key = 0
        # Designate the bridge and update the crossing's info.
        self.bridges[key] = [crossing.pd_code[1], crossing.pd_code[3]]
        self.free_crossings.remove(crossing)
        crossing.bridge = key
        logging.debug('Crossing ' + str(crossing.pd_code)
            + ' has been designated as a bridge with key ' + str(key))
        self.extend_bridge(crossing.bridge)

    def drag_crossing_under_bridge(self, crossing_to_drag, adjacent_segment):
        def find_bridge_to_go_under(adjacent_segment):
            """
            Return the bridge crossing under which to drag a free crossing.

            Arguments:
            adjacent_segment -- (int) The PD code value of the segment to drag
                a crossing along.
            """
            for crossing in self.bridge_crossings():
                if adjacent_segment in crossing.pd_code:
                    return crossing
```

```
bridge_crossing = find_bridge_to_go_under(adjacent_segment)
a, b, c, d = crossing_to_drag.pd_code
e, f, g, h = bridge_crossing.pd_code
new_max_pd_val = self.max_pd_code_value()+4
bid = bridge_crossing.bridge
y = bridge_crossing.overpass_traveled_from()
adjacent_segment_index = crossing_to_drag.pd_code.index(adjacent_segment)

logging.debug('We will drag ' + str(crossing_to_drag.pd_code)
    + ' under ' + str(bridge_crossing.pd_code))

# Alter the PD codes of all crossings not invloved in the drag.
a_y_sorted = sorted([a, y])
for crossing in diff(self.crossings, [crossing_to_drag, bridge_crossing]):
    crossing.alter_for_drag(a_y_sorted)

# Replace the crossing being dragged, (a,b,c,d).
if d == e:
    i = sorted([a, y, b]).index(b)
    if a < y:
        m, n, r, s, t, u, v, w = a, a+1, a+2, a+3, a+1, a+2,
            alter_if_greater(b+1+2*i, new_max_pd_val, 0, new_max_pd_val),
            alter_if_greater(b+2+2*i, new_max_pd_val, 0, new_max_pd_val)
        if y == f:
            logging.debug('Dragging case d=e, a<y, y==f')
            y_vals_one = alter_y_values(y, [4,5], new_max_pd_val)
            y_vals_two = alter_y_values(y, [3,2], new_max_pd_val)
        elif y == h:
            logging.debug('Dragging case d=e, a<y, y==h')
            y_vals_one = alter_y_values(y, [3,2], new_max_pd_val)
            y_vals_two = alter_y_values(y, [4,5], new_max_pd_val)
    if a > y:
        m, n, r, s, t, u, v, w = a+2, a+3, a+4,
            alter_if_greater(a+5, new_max_pd_val, 0, new_max_pd_val),
            a+3, a+4,
            alter_if_greater(b+1+2*i, new_max_pd_val, 0, new_max_pd_val),
            alter_if_greater(b+2+2*i, new_max_pd_val, 0, new_max_pd_val)
        if y == f:
            logging.debug('Dragging case d=e, a>y, y==f')
```

```
            y_vals_one = alter_y_values(y, [2,3], new_max_pd_val)
            y_vals_two = alter_y_values(y, [1,0], new_max_pd_val)
        elif y == h:
            logging.debug('Dragging case d=e, a>y, y==h')
            y_vals_one = alter_y_values(y, [1,0], new_max_pd_val)
            y_vals_two = alter_y_values(y, [2,3], new_max_pd_val)
elif b == e:
    i = sorted([a,y,d]).index(d)
    if a < y:
        m, n, r, s, t, u, v, w = a, a+1, a+2, a+3, a+1, a+2,
            alter_if_greater(d+2+2*i, new_max_pd_val, 0, new_max_pd_val),
            alter_if_greater(d+1+2*i, new_max_pd_val, 0, new_max_pd_val)
        if y == f:
            logging.debug('Dragging case b=e, a<y, y==f')
            y_vals_one = alter_y_values(y, [2,3], new_max_pd_val)
            y_vals_two = alter_y_values(y, [5,4], new_max_pd_val)
        elif y == h:
            logging.debug('Dragging case b=e, a<y, y==h')
            y_vals_one = alter_y_values(y, [5,4], new_max_pd_val)
            y_vals_two = alter_y_values(y, [2,3], new_max_pd_val)
    if a > y:
        m, n, r, s, t, u, v, w = a+2, a+3, a+4,
        alter_if_greater(a+5, new_max_pd_val, 0, new_max_pd_val),
        a+3, a+4,
        alter_if_greater(d+2+2*i, new_max_pd_val, 0, new_max_pd_val),
        alter_if_greater(d+1+2*i, new_max_pd_val, 0, new_max_pd_val)
        if y == f:
            logging.debug('Dragging case b=e, a>y, y==f')
            y_vals_one = alter_y_values(y, [0,1], new_max_pd_val)
            y_vals_two = alter_y_values(y, [3,2], new_max_pd_val)
        elif y == h:
            logging.debug('Dragging case b=e, a>y, y==h')
            y_vals_one = alter_y_values(y, [3,2], new_max_pd_val)
            y_vals_two = alter_y_values(y, [0,1], new_max_pd_val)
elif d == g:
    i = sorted([a,y,e]).index(e)
    if a < y:
        m, n, r, s, t, u, v, w = a, a+1, a+2, a+3, a+1, a+2,
            alter_if_greater(e+1+2*i, new_max_pd_val, 0, new_max_pd_val),
            e+2*i
```

```
            if y == f:
                logging.debug('Dragging case d=g, a<y, y==f')
                y_vals_one = alter_y_values(y, [3,2], new_max_pd_val)
                y_vals_two = alter_y_values(y, [4,5], new_max_pd_val)
            elif y == h:
                logging.debug('Dragging case d=g, a<y, y==h')
                y_vals_one = alter_y_values(y, [4,5], new_max_pd_val)
                y_vals_two = alter_y_values(y, [3,2], new_max_pd_val)
        if a > y:
            m, n, r, s, t, u, v, w = a+2, a+3, a+4,
                alter_if_greater(a+5, new_max_pd_val, 0, new_max_pd_val),
                a+3, a+4,
                alter_if_greater(e+1+2*i, new_max_pd_val, 0, new_max_pd_val),
                e+2*i
            if y == f:
                logging.debug('Dragging case d=g, a>y, y==f')
                y_vals_one = alter_y_values(y, [1,0], new_max_pd_val)
                y_vals_two = alter_y_values(y, [2,3], new_max_pd_val)
            elif y == h:
                logging.debug('Dragging case d=g, a>y, y==h')
                y_vals_one = alter_y_values(y, [2,3], new_max_pd_val)
                y_vals_two = alter_y_values(y, [1,0], new_max_pd_val)
    elif b == g:
        i = sorted([a,y,e]).index(e)
        if a < y:
            m, n, r, s, t, u, v, w = a, a+1, a+2, a+3, a+1, a+2, e+2*i,
            alter_if_greater(e+1+2*i, new_max_pd_val, 0, new_max_pd_val)
            if y == f:
                logging.debug('Dragging case b=g, a<y, y==f')
                y_vals_one = alter_y_values(y, [5,4], new_max_pd_val)
                y_vals_two = alter_y_values(y, [2,3], new_max_pd_val)
            elif y == h:
                logging.debug('Dragging case b=g, a<y, y==h')
                y_vals_one = alter_y_values(y, [2,3], new_max_pd_val)
                y_vals_two = alter_y_values(y, [5,4], new_max_pd_val)
        if a > y:
            m, n, r, s, t, u, v, w = a+2, a+3, a+4,
                alter_if_greater(a+5, new_max_pd_val, 0, new_max_pd_val),
                a+3, a+4, e+2*i,
                alter_if_greater(e+1+2*i, new_max_pd_val, 0, new_max_pd_val)
```

```
        if y == f:
            logging.debug('Dragging case b=g, a>y, y==f')
            y_vals_one = alter_y_values(y, [3,2], new_max_pd_val)
            y_vals_two = alter_y_values(y, [0,1], new_max_pd_val)
        elif y == h:
            logging.debug('Dragging case b=g, a>y, y==h')
            y_vals_one = alter_y_values(y, [0,1], new_max_pd_val)
            y_vals_two = alter_y_values(y, [3,2], new_max_pd_val)

crossing_one = Crossing([m, y_vals_one[0], n, y_vals_one[1]], bid)
crossing_two = Crossing([r, y_vals_two[0], s, y_vals_two[1]], bid)
crossing_to_drag.pd_code = [t, v, u, w]
index = self.crossings.index(crossing_to_drag)
self.crossings[index:index+1] = crossing_one, crossing_to_drag, crossing_two
logging.debug('(a,b,c,d) becomes ' + str(crossing_one.pd_code)
    + str(crossing_to_drag.pd_code) + str(crossing_two.pd_code))

# Alter the PD code of the bridge crossing, (e,f,g,h).
if b == e:
    m = alter_if_greater(d+2*i, new_max_pd_val, 0, new_max_pd_val)
    n = alter_if_greater(d+1+2*i, new_max_pd_val, 0, new_max_pd_val)
if d == e:
    m = alter_if_greater(b+2*i, new_max_pd_val, 0, new_max_pd_val)
    n = alter_if_greater(b+1+2*i, new_max_pd_val, 0, new_max_pd_val)
elif (d == g) or (b == g):
    m = alter_if_greater(e+1+2*i, new_max_pd_val, 0, new_max_pd_val)
    n = alter_if_greater(e+2+2*i, new_max_pd_val, 0, new_max_pd_val)
addends = get_y_addends(a, h, y)
bridge_crossing.pd_code = [m, y+addends[0], n, y+addends[1]]
logging.debug('(e,f,g,h) becomes ' + str(bridge_crossing.pd_code))

logging.debug('PD code of the knot after dragging is ' + str(self))

# Alter PD code values of bridge ends.
for i, bridge in self.bridges.iteritems():
    self.bridges[i] = map(alter_element_for_drag, bridge,
        repeat(a_y_sorted[0],2), repeat(a_y_sorted[1],2))

# Check if the crossing we dragged is now covered by a bridge.
for i, bridge in self.bridges.iteritems():
```

```
        for end in bridge:
            if (end == crossing_to_drag.pd_code[1])
                    or (end == crossing_to_drag.pd_code[3]):
                self.extend_bridge(i)
                logging.debug('Bridge end ' + str(end)
                    + ' has been extended to cover the crossing we dragged')
    logging.debug('After dragging and altering, the bridges are '
        + str(self.bridges))

    # Get the value of the next segment to drag along in case we continue
    # with this crossing.
    next_segment = crossing_to_drag.pd_code[adjacent_segment_index]
    logging.debug('If we drag this crossing again, we should drag it along '
        + str(next_segment))

    return crossing_to_drag, next_segment

def drag_crossing_under_bridge_resursively(self, crossing_to_drag,
                                        adjacent_segment, drag_count):
    """
    Drag a crossing under multiple, consecutive bridges.

    Arguments:
    crossing_to_drag -- (obj) A Crossing to drag
    adjacent_segment -- (int) The PD code value of the adjacent segment to
        drag along
    drag_count -- (int) The number of bridges to drag the crossing underneath
    """
    while (drag_count > 0):
        crossing_to_drag, adjacent_segment = self.drag_crossing_under_bridge(
            crossing_to_drag, adjacent_segment)
        drag_count -= 1
        # Stop if the crossing being dragged has been assigned to a bridge.
        if crossing_to_drag.bridge:
            break;

def extend_bridge(self, bridge_index):
    """
    Extend both ends of a bridge until it deadends.
```

```
        Arguments:
        bridge_index -- (int) the index of the bridge to extend
        """
        bridge = self.bridges[bridge_index]
        logging.debug('We will try to extend the bridge ' + str(bridge))
        for x in bridge:
            index = bridge.index(x)
            x_is_deadend = False
            while (x_is_deadend == False):
                result = filter(lambda free_crossing: x in free_crossing.pd_code,
                    self.free_crossings)
                if result:
                    crossing = result.pop()
                    if x == crossing.pd_code[1]:
                        logging.debug('Bridge end ' + str(x)
                            + ' can be extended to ' + str(crossing.pd_code[3]))
                        bridge[index] = crossing.pd_code[3]
                        x = crossing.pd_code[3]
                        self.free_crossings.remove(crossing)
                        crossing.bridge = bridge_index
                    elif x == crossing.pd_code[3]:
                        logging.debug('Bridge end ' + str(x)
                            + ' can be extended to ' + str(crossing.pd_code[1]))
                        bridge[index] = crossing.pd_code[1]
                        x = crossing.pd_code[1]
                        self.free_crossings.remove(crossing)
                        crossing.bridge = bridge_index
                    else:
                        logging.debug('Bridge end ' + str(x)
                            + ' is a dead-end and cannot be extended')
                        x_is_deadend = True
                else:
                    break;

    def find_crossing_to_drag(self):
        max_pd_code_value = self.max_pd_code_value()
        for bridge in self.bridges.itervalues():
            for end in bridge:
                crossings_containing_end = []
                for crossing in self.bridge_crossings():
```

```
        if end in crossing.pd_code:
            crossings_containing_end.append(crossing)
    if len(crossings_containing_end) == 2:
        # end is a T stem.
        logging.debug(str(end) + ' is a T stem')

        # Get the value of the segment adjacent to end.
        for end_crossing in crossings_containing_end:
            i = end_crossing.pd_code.index(end)
            if (i%2 == 0):
                adjacent_segment = end_crossing.pd_code[(i+2)%4]
                logging.debug('The segment adjacent to the T stem is '
                    + str(adjacent_segment))
                # Determine the addend needed to calculate the next
                # adjacent segment based on the direction we travel
                # along the T stem.
                if i == 0:
                    next_segment_addend = 1
                else:
                    next_segment_addend = -1
                logging.debug('The next_segment_addend is '
                    + str(next_segment_addend))
                break;

        drag_count = 0
        continue_search = True
        while continue_search:
            reached_deadend = False

            # Does adjacent_segment belong to a free crossing (deadend)?
            for free_crossing in self.free_crossings:
                if adjacent_segment in free_crossing.pd_code:
                    reached_deadend = True
                    break;

            if reached_deadend:
                if (free_crossing.pd_code.index(adjacent_segment)%2
                     == 1):
                    # The crossing is oriented such that we can drag it.
                    drag_count += 1
```

```
                        logging.debug('Crossing '
                            + str(free_crossing.pd_code)
                            + ' can be dragged along '
                            + str(adjacent_segment))
                        return (free_crossing, adjacent_segment, drag_count)
                    else:
                        continue_search = False
                        logging.debug(
                            'We have completed our search of this stem')
                        break
                else:
                    drag_count += 1
                    # Consider the next crossing along the T stem.
                    adjacent_segment = next_adjacent_segment(
                        adjacent_segment, next_segment_addend,
                        max_pd_code_value)
                    logging.debug('We need to consider the next crossing'
                        + ' along the T stem containing '
                        + str(adjacent_segment))

    # If we check all of the bridge Ts and cannot find a crossing to drag,
    # return False to signify we need to identify a new bridge.
    logging.debug('There are no crossings to drag. '
        + ' We need to identify another bridge.')
    return False

def has_rm1(self):
    """
    Inspect a knot for crossings that can be eliminated
    by Reidemeister moves of type 1.
    """
    twisted_crossings = []
    for index, crossing in enumerate(self.crossings):
        if crossing.has_duplicate_value():
            twisted_crossings.append(index)
            logging.debug('The knot can be simplified by RM1 at crossing '
                + str(crossing.pd_code))
            return twisted_crossings
    return False
```

```
def has_rm2(self):
    """
    Inspect a knot for crossings that can be eliminated
    by Reidemeister moves of type 2.

    Return the crossings which form an arc and
    the PD code value of the segments which will be eliminated when the
    knot is simplified.
    """
    def compare_pd_codes_for_rm2(indices_to_compare, current_crossing,
            next_crossing):
        output = False
        for comparision in indices_to_compare:
            current_comparision = [current_crossing.pd_code[comparision[0][0]],
                current_crossing.pd_code[comparision[0][1]]]
            next_comparison = [next_crossing.pd_code[comparision[1][0]],
                next_crossing.pd_code[comparision[1][1]]]
            if current_comparision == next_comparison:
                # True if a RM2 move is possible.
                pd_code_segments_to_eliminate = []
                for segment_to_eliminate in current_comparision:
                    if segment_to_eliminate == 1:
                        pd_code_segments_to_eliminate.append(
                            [segment_to_eliminate, -1])
                    else:
                        pd_code_segments_to_eliminate.append(
                            [segment_to_eliminate, -2])
                output = ([index, next_index], pd_code_segments_to_eliminate)
                break
        return output

    num_crossings = len(self.crossings)
    has_rm2 = False
    for index, current_crossing in enumerate(self.crossings):
        if has_rm2 == False:
            next_index = (index+1)%num_crossings
            next_crossing = self.crossings[next_index]
            difference = max(
                current_crossing.pd_code[0],
                next_crossing.pd_code[0]) - min(current_crossing.pd_code[0],
```

```
                                            next_crossing.pd_code[0])
            if (difference == 1):
                indices_to_compare = [[[2,3],[0,3]],[[1,2],[1,0]]]
                has_rm2 = compare_pd_codes_for_rm2(indices_to_compare,
                    current_crossing, next_crossing)
            elif (difference == num_crossings-1):
                indices_to_compare = [[[0,3],[2,3]],[[0,1],[2,1]]]
                has_rm2 = compare_pd_codes_for_rm2(indices_to_compare,
                    current_crossing, next_crossing)
        else:
            break
    return has_rm2

def list_bridge_ts(self, directory, depth):
    """
    Generate a list of bridge choices that form a "T".

    Arguments:
    directory -- (str) The base path to store all the output files.
    depth -- (int) The depth of the tree
    """
    if self.bridges == {}:
        i = 1
        depth_suffix = '_' + str(depth)
        for a, b in itertools.combinations(self.free_crossings, 2):
            if list(set(a.pd_code).intersection(b.pd_code)):
                name = self.name + '_tree_' + str(i) + depth_suffix
                e,f,g,h = a.pd_code
                p,q,r,s = b.pd_code
                bridges = {0:[f,h],1:[q,s]}
                logging.debug('We found ' + name + ' at '
                    + str(a.pd_code) + ', ' + str(b.pd_code))
                # Create the directory for this tree.
                tree_directory = directory + '/tree_' + str(i)
                if not os.path.exists(tree_directory):
                    os.makedirs(tree_directory)
                # Create the file to store the tree root.
                tree_file = tree_directory + '/tree_' + str(i)
                    + depth_suffix + '.csv'
                outfile = open(tree_file, "w")
```

```
                outputwriter = csv.writer(outfile, delimiter=',')
                outputwriter.writerow(['name','pd_notation','bridges'])
                outputwriter.writerow([name,str(self),bridges])
                i += 1
        outfile.close()
    else:
        # Check if a file for this knot & depth exists. If not, create the file.
        tree_prefix = directory.rsplit('/', 1)[1]
        depth_suffix = '_' + str(depth)
        file_name = tree_prefix + depth_suffix + '.csv'
        file_path = directory + '/' + file_name
        if not os.path.isfile(file_path):
            # Create the file we need with headers.
            with open(file_path, "w") as outfile:
                outputwriter = csv.writer(outfile, delimiter=',')
                outputwriter.writerow(['name','pd_notation','bridges'])
        # Find and store bridge Ts.
        i = 1
        for a, b in itertools.product(self.bridge_crossings(),
                self.free_crossings):
            knot_copy = copy.deepcopy(self)
            if list(set(a.pd_code).intersection(b.pd_code)):
                knot_copy.designate_bridge(b)
                knot_name_parts = self.name.rsplit('_', 1)
                knot_copy_name = knot_name_parts[0] + '_' + str(i)
                with open(file_path, "a") as outfile:
                    outputwriter = csv.writer(outfile, delimiter=',')
                    outputwriter.writerow([knot_copy_name,
                        str(knot_copy),str(knot_copy.bridges)])
                i += 1

def max_pd_code_value(self):
    """
    Return the maximum value possible in the PD code.
    """
    return len(self.crossings)*2

def merge_bridges(self, bridge_a, bridge_b):
    """
    Merge bridges that become one through Reidemeister moves type 1 or 2.
```

```
        Arguments:
        bridge_a -- The key of a bridge invloved in the merge.
        bridge_b -- The key of the other bridge invovled in the merge.
        """
        self.delete_bridge(bridge_a)
        self.extend_bridge(bridge_b)
        return self

    def num_crossings(self):
        """
        Return the number of crossings in the knot.
        """
        return len(self.crossings)

    def simplify_bridges(self, key):
        """
        Delete or merge bridges eliminated as part of Reidemeister moves.

        Arguments:
        key -- The index of the bridge to eliminate or None.
        """
        if key != None:
            bridge_to_check = self.bridges[key]
            if (bridge_to_check[0] == bridge_to_check[1]):
                # Remove the bridge if it has become a simple arc.
                self.delete_bridge(key)
            else:
                other_bridges = {other_key: value for other_key, value in
                    self.bridges.items() if other_key != key}
                for (end, (other_key, other_ends)) in
                        itertools.product(bridge_to_check, other_bridges.items()):
                    if end in other_ends:
                        # Merge bridges that have been joined.
                        self.merge_bridges(key, other_key)
                        break
        return self

    def simplify_rm1(self, twisted_crossings):
        """
```

```
Simplify one level of a knot by Reidemeister moves of type 1.

Arguments:
twisted_crossings -- (list) the indices of crossings to eliminate
"""
def alter_bridge_end_for_rm1(x, duplicate_value, max_value):
    if x > duplicate_value:
        x -= 2
        if x > max_value:
            x = x%max_value
    elif x == duplicate_value:
        if duplicate_value == 1:
            x = max_value
        elif duplicate_value == max_value+2:
            x = 1
        else:
            x -= 1
    return x

crossings = self.crossings
for index in sorted(twisted_crossings, reverse = True):
    duplicate_value = self.crossings[index].has_duplicate_value()
    key = self.crossings[index].bridge
    original_max_value = len(self.crossings)*2
    self.delete_crossings([index])
    new_max_value = original_max_value-2

    if duplicate_value == original_max_value:
        extend_if_bridge_end = [1, duplicate_value + 1]
        # Adjust crossings.
        for crossing in self.crossings:
            crossing.alter_elements_greater_than(new_max_value,
                -new_max_value, new_max_value)
    else:
        extend_if_bridge_end = [duplicate_value - 1, duplicate_value + 1]
        # Adjust crossings.
        for crossing in self.crossings:
            crossing.alter_elements_greater_than(duplicate_value, -2,
                new_max_value)
    for i, bridge in self.bridges.iteritems():
```

```
            # Adjust bridges.
            self.bridges[i] = map(alter_bridge_end_for_rm1, bridge,
                repeat(duplicate_value, 2), repeat(new_max_value, 2))
            # Try to extend bridges.
            extend_bridge = any(x in bridge for x in extend_if_bridge_end)
            if extend_bridge:
                self.extend_bridge(i)
        self.simplify_bridges(key)
        logging.info('After simplifying the knot for RM1 at segment '
            + str(duplicate_value) + ', the PD code is ' + str(self)
            + ' and the bridges are ' + str(self.bridges))
    return self

def simplify_rm1_recursively(self):
    """
    Simplify a knot by Reidemeister moves of type 1 until
    no more moves are possible.
    """
    while True:
        moves_possible = self.has_rm1()
        if moves_possible:
            self.simplify_rm1(moves_possible)
        if not moves_possible:
            break
    return self

def simplify_rm2(self, crossing_indices, segments_to_eliminate):
    """
    Simplify a knot by one Reidemeister move of type 2.

    Arguments:
    crossing_indices -- (list) the indices of crossings to remove
    segments_to_eliminate -- (list) each element is a list of the PD code of a
            segment that is simplified and the addend to apply to all
            greater PD code values.
    """
    key = self.crossings[crossing_indices[0]].bridge
    self.delete_crossings(crossing_indices)
    maximum = len(self.crossings) * 2
    extend_if_bridge_end = []
```

```
segments_to_eliminate.sort(reverse = True)

logging.info('The segments ' + str(segments_to_eliminate[0][0])
    + ' and ' + str(segments_to_eliminate[1][0])
    + ' can be elimiated by RM2 moves.')

for segment in segments_to_eliminate:
    value, addend = segment

    # Alter values of each crossing.
    for crossing in self.crossings:
        crossing.alter_elements_greater_than(value, addend)

    # Adjust bridges.
    for key, bridge in self.bridges.iteritems():
        self.bridges[key] = map(alter_if_greater, bridge,
            repeat(value, 2), repeat(addend, 2))

    # Alter values of remaining segments to eliminate.
    segments_to_eliminate = alter_segment_elements_greater_than(
        segments_to_eliminate, value, addend)

    # Remove segments as we finish with them.
    del(segments_to_eliminate[-1])

# Mod final crossings based on maximum value allowed.
for crossing in self.crossings:
    crossing.alter_elements_greater_than(maximum, 0, maximum)
# Mod final bridge ends based on maximum value allowed.
self.alter_bridge_segments_greater_than(maximum, 0, maximum)

extend_if_bridge_end = [value - 1, value + 1]
for bridge_index, bridge in self.bridges.iteritems():
    extend_bridge = any(x in bridge for x in extend_if_bridge_end)
    if extend_bridge:
        self.extend_bridge(bridge_index)
self.simplify_bridges(key)
logging.info('After simplifying by RM2, the PD code is ' + str(self)
    + ' and the bridges are ' + str(self.bridges))
```

```
        return self

    def simplify_rm2_recursively(self):
        """Simplify a knot by Reidemeister moves of type 2 until
        no more moves are possible.
        """
        while True:
            moves_possible = self.has_rm2()
            if moves_possible:
                self.simplify_rm2(moves_possible[0], moves_possible[1])
            if not moves_possible:
                break;
        return self

    def simplify_rm1_rm2_recursively(self):
        """
        Simplify a knot by Reidemeister moves of types 1 & 2 until
        no more moves are possible.
        """
        while True:
            if self.has_rm1():
                self.simplify_rm1_recursively()
            if self.has_rm2():
                self.simplify_rm2_recursively()
            if not self.has_rm1() and not self.has_rm2():
                logging.info('No more moves of type RM1 or RM2 are possible.')
                break;
        return self

def alter_element_for_drag(x, first, second):
    """
    A helper function for the drag the underpass move to adjust PD code values
    of crossings not directly invloved in the move.

    Arguments:
    x -- (int) The value to alter.
    first -- (int) The PD code value of the first segment we travel into.
    second -- (int) The PD code value of the second segment we travel into.
    """
    if x <= first:
```

```
        return x
    if first < x <= second:
        return x+2
    if x > second:
        return x+4

def alter_if_greater(x, value, addend, maximum = None):
    """
    Arguments:
    x -- (int) The number to alter.
    value -- (int) The number to compare each element of the crossing with.
    addend -- (int) The number to add to crossing elements greater than value.
    maximum -- (int) The maximum allowed value of elements in the crossing.
    """
    if x > value:
        x += addend
        if x == 0:
            x = maximum
        if maximum and (x > maximum):
            x = x%maximum
    return x

def alter_segment_elements_greater_than(segments, value, addend):
    """
    Arguments:
    segments -- (list) A list of lists of integers to alter.
    value -- (int) The number to compare each element of the crossing with.
    addend -- (int) The number to add to crossing elements greater than value.
    """
    altered_segments = []
    for pair in segments:
        altered_segments.append([alter_if_greater(x, value, addend) for x in pair])
    return altered_segments

def alter_y_values(y, addends, maximum):
    """
    A helper function for the drag the underpass move.

    Arguments:
    y -- (int) The PD code value of f or h, whichever we travel from
```

```
        toward the other.
    addends -- (list) Integer values to add to y.
    maximum -- (int) The maximum PD code value in the knot after dragging
        a crossing.
    """
    y_vals = [alter_if_greater(y+addend, maximum, 0, maximum) for addend in addends]
    return y_vals

def create_knot_from_pd_code(pd_code, name = None, bridges = None):
    """
    Create a Knot object using a provided PD code.

    Arguments:
    pd_code -- (list) the PD notation of a knot expressed as a list of lists
    name -- (str) a string to identify the knot
    bridges -- (list) Each element is a list of PD code values for the
        ends of each bridge
    """
    return Knot([Crossing(crossing) for crossing in pd_code], name, bridges)

def diff(first, second):
    """
    Compute the difference of two lists.

    Arguments:
    first -- (list) The list to prune
    second -- (list) The elements to remove from "first" (if they exist)
    """
    second = set(second)
    return [item for item in first if item not in second]

def get_y_addends(a, h, y):
    """
    Get the addends for y to alter the bridge tuple for a drag.

    Arguments:
    a -- (int) PD code of the 1st element in the tuple being dragged.
    h -- (int) PD code of the 4th element in the bridge tuple.
    y -- (int) PD code of the 2nd or 4th element in the bridge tuple that is
        traveled from toward the other.
```

```
    """
    if a < y:
        addends = [3,4]
    elif a > y:
        addends = [1,2]
    addends.sort(reverse = bool(y == h))
    return addends

def next_adjacent_segment(current_segment, next_segment_addend, max_pd_code_value):
    """
    Given a direction of travel, return the PD code segment of the section
    adjacent to current_segment.

    Arguments:
    current_segment -- (int) The PD code value of the current segment
    next_segment_addend -- (int) 1 or -1, depending on the direction of travel
    max_pd_code_value -- (int) The maximum PD code value for the knot diagram.
    """
    next_segment = alter_if_greater(current_segment, 0, next_segment_addend,
        max_pd_code_value)
    if next_segment == 0:
        next_segment = max_pd_code_value
    return next_segment
```

C.3 analyze_output.py

```
#!/usr/bin/env python2.7

import csv, getopt, numpy, os, sys

def write_analysis_output(argv):
    """
    Return the minimum bridge index returned by bridge_computation.py.
    """
    input_source = ''
    output_dir = 'analyzed_output'
    numeral_places = 4
    help_message = ' '.join(['bridge_computation.py -i <input_source>',
        '-o <output_dir> -p <numeral_places>'])
    try:
        opts, args = getopt.getopt(argv, "hi:o:p:",
            ["input_source=", "output_dir", "numeral_places",])
    except getopt.GetoptError:
        error_message = ' '.join(['There was an error getting the arguments.',
            help_message])
        print error_message
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print help_message
            sys.exit()
        elif opt in ("-i", "--input_source"):
            input_source = arg
        elif opt in ("-o", "--output_dir"):
            output_dir = arg
        elif opt in ("-p", "--numeral_places"):
            numeral_places = int(arg)
    # Create a file for output.
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    outfile_path = output_dir + '/minimum_computed_bridge_indices.csv'
```

```
    with open(outfile_path, "w") as outfile:
        outputwriter = csv.writer(outfile, delimiter=',')
        outputwriter.writerow(['knot','minimum_computed_bridge_index'])

    if os.path.isdir(input_source):
        # Traverse the directory to process all csv files.
        for root, dirs, files in os.walk(input_source):
            for file in files:
                if file.endswith(".csv"):
                    find_minimum_computed_bridge_index(os.path.join(root, file),
                        outfile_path, numeral_places)
    elif os.path.isfile(input_source):
        find_minimum_computed_bridge_index(input_source, outfile_path,
            numeral_places)
    else:
        input_message = ' '.join([
            'The specified input is not a file or a directory.',
            'Please try a different input.'])
        print input_message
        logging.warning(input_message)

def find_minimum_computed_bridge_index(csv_file, outfile_path, numeral_places):
    computed_bridge_indexes = numpy.loadtxt(fname=csv_file, skiprows=1,
        usecols=(1,), delimiter=',', dtype=int)
    min_computed_bridge_index = min(computed_bridge_indexes)
    # Format the name for better sorting.
    root, ext = os.path.splitext(os.path.basename(csv_file))
    knot_name = root[:-7]
    num_crossings, number = knot_name.split('_')
    number = number.zfill(numeral_places)
    knot_name = num_crossings + '_' + number

    with open(outfile_path, "a") as outfile:
        outputwriter = csv.writer(outfile, delimiter=',')
        outputwriter.writerow([knot_name,min_computed_bridge_index])

if __name__ == "__main__":
    write_analysis_output(sys.argv[1:])
```