8-2019

The T_3,T_4-conjecture for links

Katie Tucker
katherine.tucker@huskers.unl.edu

THE $T_3, \overline{T_4}$-CONJECTURE FOR LINKS

by

Katie Tucker

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Mathematics

Under the Supervision of Professors Mark Brittenham and Susan Hermiller

Lincoln, Nebraska

August, 2019

THE $T_3, \overline{T_4}$-CONJECTURE FOR LINKS

Katie Tucker, Ph.D.

University of Nebraska, 2019

Adviser: Mark Brittenham and Susan Hermiller

An oriented n-component link is a smooth embedding of n oriented copies of S^1 into S^3. A diagram of an oriented link is a projection of a link onto $\mathbb{R}^2$ such that there are no triple intersections, with notation at double intersections to indicate under and over strands and arrows on strands to indicate orientation. A local move on an oriented link is a regional change of a diagram where one tangle is replaced with another in a way that preserves orientation. We investigate the local moves t_3 and $\overline{t_4}$, which are conjectured to be an unlinking set (i.e., turns the link into an unlink) on oriented links (Kirby Problem List # 1.59(4)). Using combinatorial and computational methods, we show all oriented links with braid index at most 5, except for possibly the link formed from the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$, are $t_3, \overline{t_4}$ unlinkable. We extend these methods to oriented links with braid index 6 and crossing number at most 12.

Table of Contents

Chapter 1

Introduction

A local move on a knot or link diagram is a move which replaces one section of the diagram, often without regard to isotopy in the original diagram. An unknotting move is a local move that can transform any diagram of a knot into an unknotted S^1 when used in conjunction with isotopy. Analogously, an unlinking move is a local move that can transform any link diagram into some number of copies of disjoint, unknotted S^1. These moves can vary in complexity from crossing changes, which swap the over- and under-strands of a crossing in the diagram, to moves on larger regions, perhaps involving multiple crossings or with restrictions on how the crossings need be arranged in order to utilize the local move.

To this end, when considering a local move, the first question to answer is whether or not the move is an unknotting (or unlinking) move. For some local moves, such as crossing changes and crossing smoothings, it is known that they are unknotting moves; for others, such as the 3-move (similar to Figure 1.1, but without orientation restrictions), counterexamples have been constructed. That is, there are knots which cannot be transformed to an unlink under the given local move.

Two families of local moves are the t_k and $\overline{t_k}$ moves on oriented links [Prz88b]. A t_k move replaces two parallel strands with the same orientation and no twists with two parallel strands with k positive half-twists between them or with k negative half-twists between them, and vice versa. The t_3 move is shown in Figure 1.1. A $\overline{t_k}$ move is analogous but

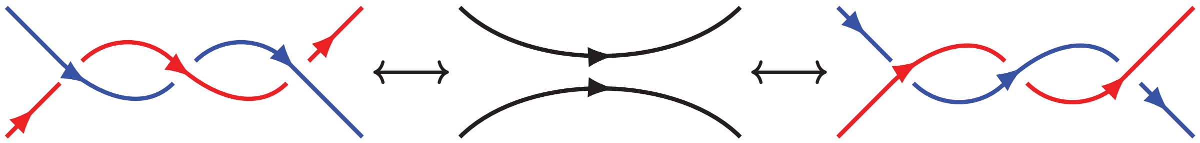


Figure 1.1: t_3 move

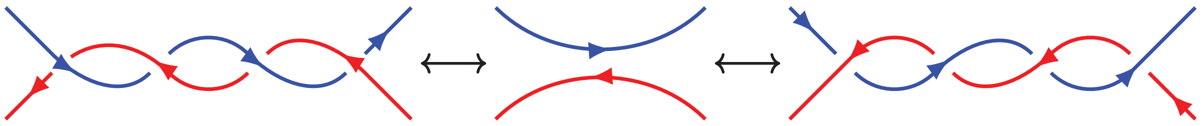


Figure 1.2: $\overline{t_4}$ move

works only on two strands with opposite orientation; see Figure 1.2 for the $\overline{t_4}$ move. These families are special cases of the n-move on unoriented links, which replaces two untwisted strands with two strands with n half-twists between them.

Several combinations of these moves have been studied as possible *unlinking sets*, or sets of local moves which can transform any link into an unlink. The 3-move (the combination t_3 and $\overline{t_3}$) was conjectured to be an unlinking move but this conjecture was disproven in 2002 by Dabkowski and Przytycki [DP02]. On the other hand, Nakanishi's conjecture [Nak90] regarding whether the 4-move is an unlinking move is still an open question. No n-move, for $n \geq 5$, is an unlinking move [Prz88b].

Of particular note for this thesis is the combination $t_3, \overline{t_4}$, shown in Figures 1.1 and 1.2. The $t_3, \overline{t_4}$ moves are conjectured to be an unlinking set (see problem #1.59(4) on the Kirby problem list [Kir]), and this conjecture has been proved for some families of oriented knots and links.

Theorem 1.1. *The $t_3, \overline{t_4}$ moves are unlinking operations for:*

(a) the closures of oriented 3-braids [Prz88b, Example 3.11],

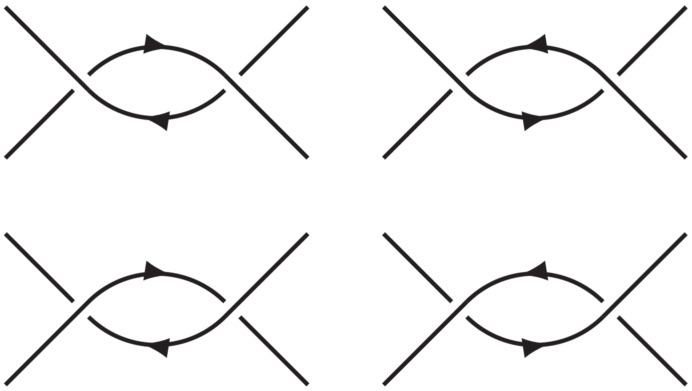


Figure 1.3: Matched diagram tangles

(b) oriented links with matched diagrams, oriented 2-bridge links, and oriented algebraic links [Prz90, Theorem 3, Corollary 6, and Corollary 8], and

(c) oriented links of up to 11 crossings [Prz93, Corollary 1.9].

Chen uses the methods of the proof of Theorem 1.1(a) to prove that the closures of all 5-strand braids are 3-move equivalent to an unlink in [Che00] and asserts that his methods apply to the combination of $t_3, \overline{t_4}$ on the closures of oriented 5-braids. However, there are issues with how his results are proven and extended. This is discussed further in Remark 3.3.

In Theorem 1.1(b), a *matched diagram* refers to a diagram of an oriented knot (or link) where the crossings can be paired up into anti-parallel bigons: regions with two crossings of the same sign where the two strands involved are oriented in opposite directions; see Figure 1.3. However, it is known that not all knots have a matched diagram, as the pretzel knot P(3, 3, -3) is a counterexample [DS11, Corollary in Section 5].

It has also been shown that the $t_3, \overline{t_4}$ moves preserve the first homology of the 3-fold cyclic cover of the exterior of a link with coefficients from $\mathbb{Z}_2$ [Prz88b, Theorem 3.2], [Prz88a, Lemma 2]. That is, if L and L' are $t_3, \overline{t_4}$ equivalent links, then $H_1(M_L^{(3)}, \mathbb{Z}_2) = H_1(M_{L'}^{(3)}, \mathbb{Z}_2)$, where $M_L^{(3)}$ and $M_{L'}^{(3)}$ are the 3-fold cyclic covers of the exteriors of L and L', respectively.

Since $\dim H_1(M_{U_n}^{(3)}, \mathbb{Z}_2) = 2(n-1)$ where U_n is the n-component unlink, this result can be used to predict to which unlink a given link should be $t_3, \overline{t_4}$ equivalent.

In this thesis, we prove the $t_3, \overline{t_4}$-conjecture for all but one $t_3, \overline{t_4}$-equivalence class of oriented links which are the closure of 5-strand braids, and we extend the methods to the closures of 6-strand braids to begin proving a similar result.

Theorem 4.1. *The closures of all oriented 5-braids, except possibly for those $t_3, \overline{t_4}$ equivalent to the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$, are $t_3, \overline{t_4}$ equivalent to an unlink.*

We note that if that the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$ is $t_3, \overline{t_4}$ equivalent to an unlink then using homology calculations it is equivalent to the 5-component unlink U_5.

Theorem 5.1. *All oriented links which result from the closure of a 6-strand braid with at most 12 crossings are $t_3, \overline{t_4}$ unlinkable.*

In Chapter 2 we cover background material on braids, the Todd-Coxeter algorithm for computing cosets, and the homology of the 3-fold cyclic cover of the exterior of a knot. In Chapter 3 we review the proof of Theorem 1.1(a) and extend these methods to prove the $t_3, \overline{t_4}$ conjecture for the closures of oriented 4-braids. In Chapter 4 we prove Theorem 4.1, and in Section 4.5 we examine the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$ in detail. Section 5.1 extends the methods used in the proof in Chapter 4 to the closures of oriented 6-strand braids and proves Theorem 5.1. Section 5.2 investigates extensions of the methods used in Chapter 4 to other local moves. Appendix A.4 contains the Python code used in proving Theorem 4.1.

Chapter 2

Background

This thesis assumes a basic knowledge of classical knot theory; for a good reference, see [Rol03]. Some of the basic facts we will use are contained in this section.

We view knots as smooth embeddings of S^1 in S^3, and two knots are equivalent if there is an isotopy from one to the other. Similarly, an n-component link is a smooth embedding of a disjoint union of n copies of S^1 in S^3. For visualization purposes, we often deal with a knot (or link) diagram, which is a projection of the knot (or link) onto $\mathbb{R}^2$ satisfying some restrictions. For a projection to be a diagram, the resultant image must have no sharp corners, and any self-intersections must involve only two strands; that is, there are no triple (or more) intersection points. To preserve crossing information, when two strands 'intersect' in the projection, the understrand is broken near the crossing. Some knot and link diagram examples are shown in Figure 2.1. Two knot (or link) diagrams are equivalent (i.e., represent the same knot or link) if there is a finite sequence of the three Reidemeister moves, shown in Figure 2.2, taking one diagram to the other.

A local move on a knot or link diagram involves changing a region of the diagram by replacing one tangle with another. Common examples are the crossing change (see Figure 2.3 (a)) and crossing smoothing (see Figure 2.3 (b)). The t_3 and $\overline{t_4}$ moves (see Figures 1.1 and 1.2), which are the focus of this thesis, are also examples of local moves.

This thesis focuses on *oriented* knots and links, which are knots and links where each

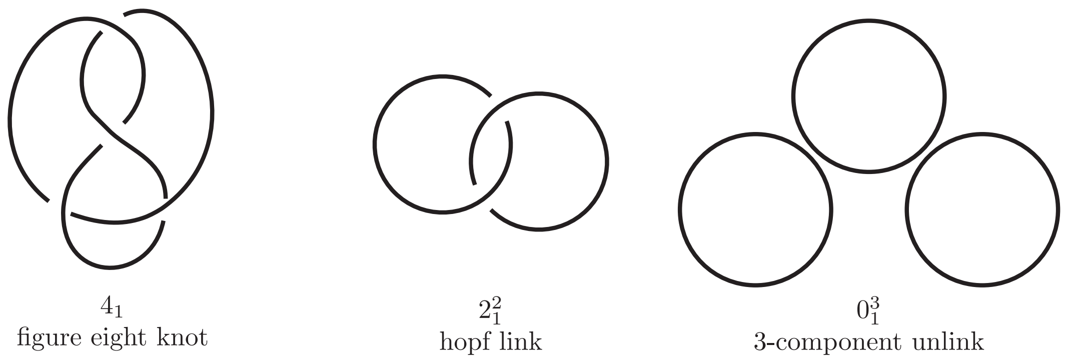


Figure 2.1: Examples of a knot, link, and unlink

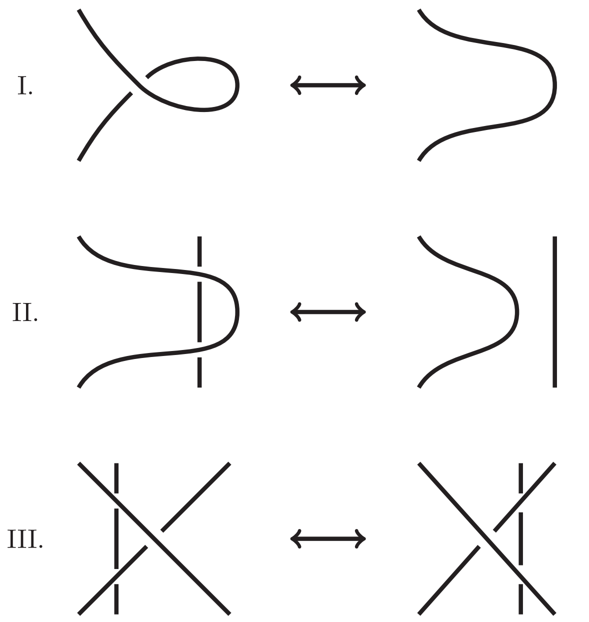
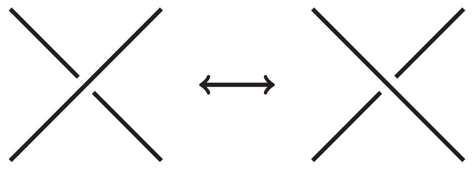
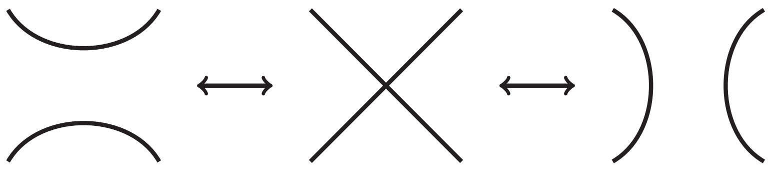


Figure 2.2: Reidemeister Moves I, II, and III



(a) crossing change



(b) crossing smoothing

Figure 2.3: Crossing change and crossing smoothing

embedded S^1 has a fixed orientation. For knots, the choice of orientation does not matter, as changing the orientation of the only component changes the global orientation. For links, however, changing the orientation on one component may change the oriented link type.

2.1 Braids

Knots and links are closely related to braids, which can be viewed as a collection of crossed strands whose origins and endpoints are fixed. To be a braid rather than a tangle, we further place the restriction that the strands all move from their origins to their endpoints monotonically. For example, the first two images in Figure 2.4 are braids with origins at the top and endpoints at the bottom, while the third is not. We can *close* a braid by connecting the first origin to the first endpoint via an arc, the second origin to the second endpoint, etc., without introducing new crossings, to get a link. By a theorem of Alexander [Ale23], the reverse also holds: any oriented link can be viewed as the closure

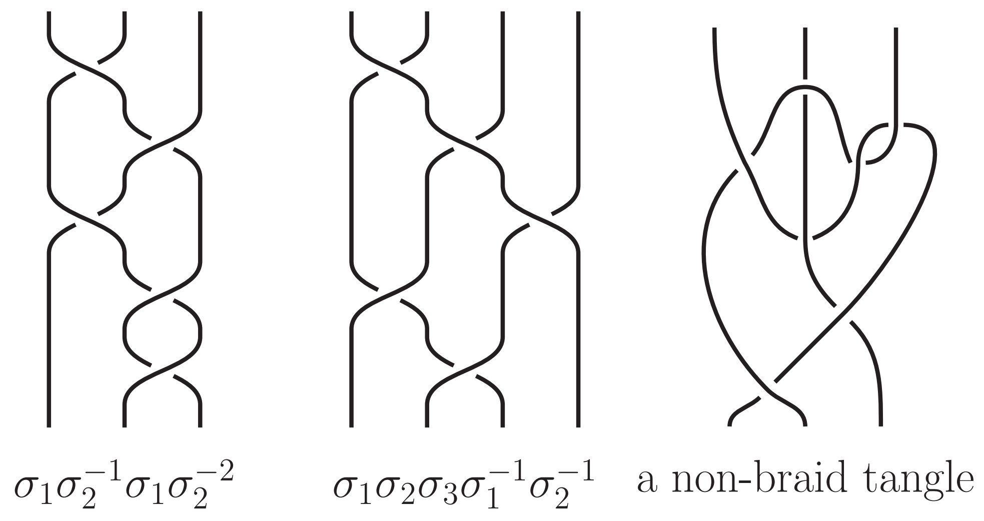

$\sigma_1\sigma_2^{-1}\sigma_1\sigma_2^{-2}$
$\sigma_1\sigma_2\sigma_3\sigma_1^{-1}\sigma_2^{-1}$
a non-braid tangle

Figure 2.4: Two braids, with associated braid words, and a tangle

of an oriented braid on some number of strands. Due to the structure of a braid, braids also have an inherent orientation where all strands are oriented from the origins to the endpoints.

Two braids are *equivalent* if they are isotopic through braids. The collection B_n of equivalences classes of braids on n strands can formally be viewed as a group, with the group operation being composition: given two braids on n strands, we can compose them by gluing the origins of the second braid to the endpoints of the first braid in such a way that the ith origin of the second braid attaches to the ith endpoint of the first braid.

This group is generated by the set of σ_i for $i = 1, \ldots, n-1$, where σ_i is the braid with the strand in the i position crossing over the strand in the $i+1$ position, and all other strands remaining straight and uncrossed. This group also has a commutativity relation: when crossings are far enough apart (i.e., do not involve the same strand(s)), the order of the crossings does not matter. The Reidemeister moves are also realized within the braid group. The first Reidemeister move gives *stabilization* and *destabilization* and changes the number of strands (see Figure 2.5). Thus the first Reidemeister move changes

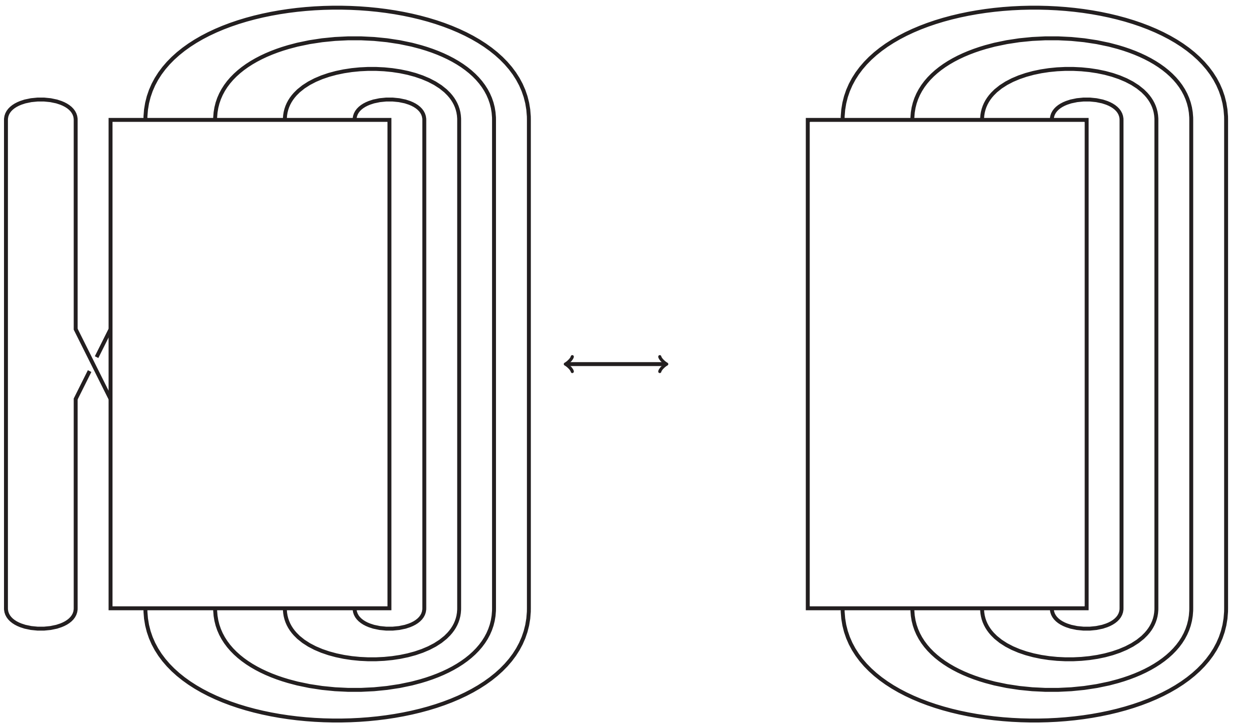


Figure 2.5: Destabilization (right arrow) and stabilization (left arrow) of braids

an element of B_n to an element of B_{n+1} (in the case of stabilization) or B_{n-1} (in the case of destabilization). The second Reidemeister move is realized as $\sigma_i\sigma_i^{-1} = 1_{B_n} = \sigma_i^{-1}\sigma_i$ within B_n, and the third Reidemeister move gives the relation shown in Figure 2.6. With this and the previous, one can construct the following presentation for the n-strand braid group:

Definition 2.1. The braid group on n strands is the group presented by

$$B_n = \langle \sigma_1, \ldots, \sigma_{n-1} \mid [\sigma_i, \sigma_j] = 1 \text{ if } |i-j| \geq 2, \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} \text{ for } 1 \leq i < n-1 \rangle$$

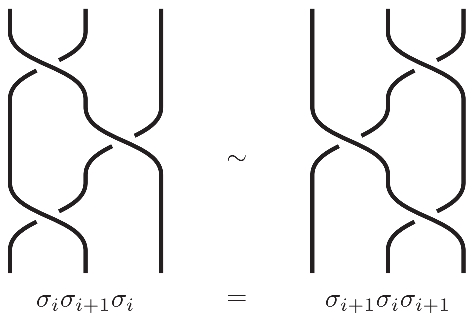

$\sim$
$\sigma_i\sigma_{i+1}\sigma_i$
$=$
$\sigma_{i+1}\sigma_i\sigma_{i+1}$

Figure 2.6: RIII move in braids

2.2 Group theory results

One of the principal group-theoretic tools we will use is the Todd-Coxeter algorithm [CT36]. This algorithm is one method for enumerating the right cosets of the quotient of a finitely presented group and at the same time discovering where the cosets are mapped under the action of right multiplication by the group generators. Upon input of a finite generating set and finite set of relators, the Todd-Coxeter algorithm constructs a coset table where each row corresponds to a coset of the quotient group and each column corresponds to a generator (or its inverse). Thus, if the generators of the group are $\{a_1, \ldots, a_k\}$, the coset table will have $2k$ columns. Each entry in the jth row tells which coset the jth coset, C_j, is mapped to under the right multiplication by the generator corresponding to that column. In particular, the first entry of the jth row gives which coset C_j is mapped to under right multiplication by a_1, the second entry is where C_j is mapped under right multiplication by a_1^{-1}, and so on. By convention, the first coset is represented by the identity element of the group. A representative for C_j can thus be constructed by finding a path from $C_1 = 1_G$ to C_j in the generating set $\{a_1, \ldots, a_k\}^{\pm 1}$. Note the first occurrence of any value s occurs within the first $s-1$ rows; that is, one of the first $s-1$ cosets maps to the sth coset under right multiplication by one of the generators, so when constructing a shortest representative, we continually move up the table and the process terminates in a finite number of steps.

To find a shortest representative for the j-th coset C_j, with $2 \leq j \leq n$, where n is the number of cosets (and thus the number of rows in the coset table), we find the first row in which the value j appears. Call this row r_1; then $1 \leq r_1 < j$ by the construction in the Todd-Coxeter algorithm. Note that for $j = 1$ we have $C_1 = 1_G$ by convention so we only need consider $j \geq 2$. This gives that the r_1-st coset C_{r_1} is mapped to C_j under right multiplication by one of the generators. That is, there is some $\alpha_1 \in \{a_1, \ldots, a_k\}^{\pm 1}$ such that $C_{r_1}\alpha_1 = C_j$. If r_1 is 1 (i.e., j appears in the first row of the coset table), we are done, and a shortest representative for the coset C_j is α_1. Otherwise, repeat: suppose that m such steps have occured, so that the r_m-th coset C_{r_m} is mapped to C_j under right multiplication by $\alpha_m \cdots \alpha_1$ with each $\alpha_i \in \{a_1, \ldots, a_k\}^{\pm 1}$. That is, $C_{r_m}\alpha_m \cdots \alpha_1 = C_j$. Then find the first row r_{m+1}, where $1 \leq r_{m+1} < r_m$, in which the value r_m appears. Then the r_{m+1}-st coset $C_{r_{m+1}}$ goes to coset C_{r_m} by right multiplication by one of the generators, indicated by which column contains the value r_m within row r_{m+1}; call this generator α_{m+1}. Thus the coset $C_{r_{m+1}}$ is mapped to coset C_{r_m} under right multiplication by α_{m+1} and to coset C_j under right multiplication by $\alpha_{m+1}\alpha_m \cdots \alpha_1$. This process continues working up the coset table until we have that the first coset goes to the jth coset by some sequence of generators $\alpha_\ell \cdots \alpha_1$ and therefore the jth coset has shortest representative $\alpha_\ell \cdots \alpha_1$. Because this method always chooses the first occurrence of a coset, the representative produced is of shortest length.

2.3 Cyclic branched covers

Another tool we utilize is the first homology of the 3-fold cyclic cover of the exterior of a link L in S^3 with coefficients from $\mathbb{Z}_2$. This is preserved by t_3 [Prz88a, Theorem 1] and $\overline{t_4}$ moves [Prz88b, Theorem 3.2].

To begin we find the 3-fold cyclic cover of the exterior of K, $S^3\backslash K$. We follow the construction laid out in [Rol03, Chapter 5(C)]. Let Σ be a Seifert surface for the knot K in S^3 and let $N \cong \text{int}(\Sigma) \times (-1,1)$ be an open bicollar of $\text{int}(\Sigma) = \Sigma\backslash K$. Let $N^+ \cong \text{int}(\Sigma) \times (0,1)$ and $N^- \cong \text{int}(\Sigma) \times (-1,0)$. Let $Y = S^3\backslash\Sigma$ be the complement of Σ and $X = S^3\backslash K$ be the complement of K. We thus have two triples (N, N^+, N^-) and (Y, N^+, N^-), as N^+ and N^- can be viewed as subspaces of both N and Y. As we focus on the 3-fold cyclic cover, we form three copies of each, denoted (N_i, N_i^+, N_i^-) and (Y_i, N_i^+, N_i^-) for $i = 0, 1, 2$. Let $\widetilde{N} = \cup_{i=0}^2 N_i$ and $\widetilde{Y} = \cup_{i=0}^2 Y_i$ be the disjoint unions. Finally, identify $N_i^+ \subset Y_i$ with $N_i^+ \subset N_i$ and $N_i^- \subset Y_i$ with $N_{i+1}^- \subset N_{i+1}$ via the identity maps, with $N_2^- \subset Y_2$ identified with $N_0^- \subset N_0$. This resulting space is $M_K^{(3)}$, the 3-fold cyclic cover of the exterior of K.

To compute the homology of $M_K^{(3)}$ with integer coefficients we consider bases B of $H_1(\text{int}(\Sigma))$ and β of $H_1(Y)$. We push each basis element $b \in B$ off of Σ and into N^+ and N^- to get b^+ and b^-, respectively, and mimic this in each of the three copies of Y_i to get a system of equations relating $\{b_i^+, b_i^- | i = 0, 1, 2; b \in B\}$ to $A = \{\alpha_i | i = 0, 1, 2; \alpha \in \beta\}$. The first homology $H_1(M_K^{(3)}, \mathbb{Z})$ is generated by the set A, along with an additional generator loop γ which runs once around $M_K^{(3)}$ and contributes a free abelian component to the group. The group has for relations the consequences of the system of equations.

Finally, to compute the homology of $M_K^{(3)}$ with coefficients from $\mathbb{Z}_2$ we use the exact sequence

$$0 \to H_1(M_K^{(3)}, \mathbb{Z})/\{2[z] : [z] \in H_1(M_K^{(3)}, \mathbb{Z})\} \to H_1(M_K^{(3)}, \mathbb{Z}/2) \to \mathbb{Z}_2 \to 0.$$

Chapter 3

Proof of the $t_3, \overline{t_4}$ conjecture for the closures of 3- and 4-strand braids

In this chapter we prove that the closures of oriented 4-strand braids are $t_3, \overline{t_4}$-unlinkable, following Przytycki's method for the closures of 3-strand braids, which is summarized in Section 3.1. We extend this concept to the closures of 4-strand braids in Section 3.2 and 5-strand braids in Chapter 4.

3.1 3-strand braids

Przytycki's proof begins with the oriented 3-strand braid group $B_3 = \langle \sigma_1, \sigma_2 | \sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2 \rangle$ (see Definition 2.1). This is an infinite group which makes handling each individual element unfeasible. Due to the orientation in braids inherited by following each strand from its origin to its endpoint, the t_3 move can be replicated on oriented 3-strand braids by the following four relations:

$$\sigma_1^3 = 1 \qquad \sigma_1^{-3} = 1 \qquad \sigma_2^3 = 1 \qquad \sigma_2^{-3} = 1$$

Note that including these as relators in the group presentation also yields that all conjugates of $\sigma_i^{\pm 3}$ are also identified with 1 in the quotient. Because conjugation in the

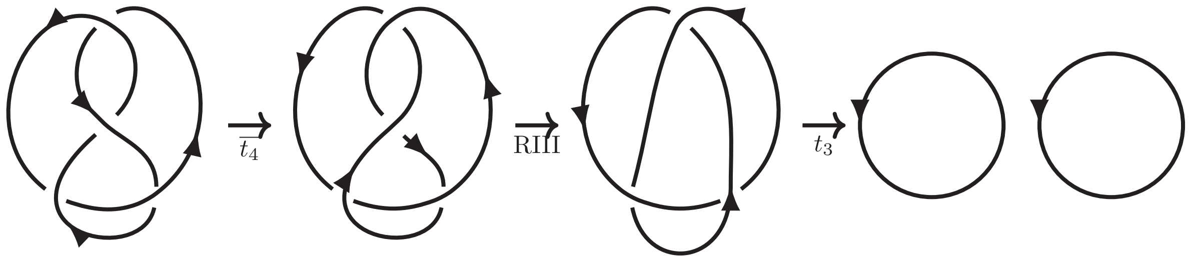


Figure 3.1: Using $t_3, \overline{t_4}$ to unlink the figure-eight knot

braid group is realized as ambient isotopy on knot and link diagrams – the closure of $\alpha\beta\alpha^{-1}$ is isotopic to the closure of $\alpha^{-1}\alpha\beta \simeq \beta$ – these relations do in fact replicate the t_3 move on 3-strand braids. Thus, to prove the $t_3, \overline{t_4}$ conjecture holds for links formed by closing 3-strand braids, it suffices to consider the quotient group Q_3 of B_3 where we include these relations. This quotient has presentation $Q_3 = \langle \sigma_1, \sigma_2 | \sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2, \sigma_1^3 = 1, \sigma_2^3 = 1\rangle = B_3/\langle\sigma_1^3\rangle^N$. Note this final equivalence holds as σ_i is conjugate to σ_j in B_n for each $i, j \in \{1, \ldots, n-1\}$: we have that $\sigma_1 = (\sigma_2\sigma_1)\sigma_2(\sigma_1^{-1}\sigma_2^{-1})$, and in general, $\sigma_i = (\sigma_{i+1}\sigma_i)\sigma_{i+1}(\sigma_i^{-1}\sigma_{i+1}^{-1})$ for $1 \leq i \leq n-2$. As conjugacy is an equivalence relation, we thus have the generators of the braid group on n strands are all conjugate to each other. Thus, looking at B_3 in particular, we have that $\langle\sigma_1^3, \sigma_2^3\rangle^N = \langle\sigma_1^3\rangle^N = \langle\sigma_2^3\rangle^N$.

The quotient group Q_3 has order 24 by a result of Coxeter [Cox57, page 99], and the elements of Q_3 are t_3 equivalence classes of 3-strand braids. That is, there is a bijection between Q_3 and the t_3 equivalence classes of 3-strand braids. Thus if any one element in a t_3 equivalence class is $t_3, \overline{t_4}$ equivalent to an unlink, then so is every element of that t_3 equialence class. Hence only one element from each t_3 equivalence class need be considered. In [Prz88b], Przytycki considers a smallest braid word representing each element of Q_3 and concludes that these elements have braid closures which are isotopic to an unlink of at most 3 components or the figure-eight knot, which is $t_3, \overline{t_4}$ equivalent to the 2-component unlink as shown in Figure 3.1. Theorem 3.1 thus follows.

Theorem 3.1 (Przytycki, 1988)**.** *Every oriented link which can be represented by a braid on 3 strands is $t_3, \overline{t_4}$ equivalent to an unlink.*

3.2 4-strand braids

In this section and the following chapter we extend this method to the 4- and 5-strand braid groups, using a result of Coxeter [Cox57, pages 102, 105] that also gives that the resultant quotients Q_4 and Q_5 are finite, of order 648 and 155,520 respectively. However, $B_n/\langle\sigma_1^3\rangle^N$ is infinite for $n \geq 6$. Using the computer program Groups, Algorithms, Programming (GAP) [GAP17], the Todd-Coxeter method is used to build a coset table for each quotient of the respective braid groups. This table is then used to build a set of shortest representatives for elements of Q_4 and Q_5 as products of the σ_i generators, and we then are left to show each representative is $t_3, \overline{t_4}$ equivalent to an unlink.

In this section, we work with the closures of 4-strand braids and the quotient Q_4 with presentation

$$\begin{aligned} Q_4 &= \langle \sigma_1, \sigma_2, \sigma_3 | \sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2, \sigma_2\sigma_3\sigma_2 = \sigma_3\sigma_2\sigma_3, \sigma_1\sigma_3 = \sigma_3\sigma_1, \sigma_1^3 = 1, \sigma_2^3 = 1, \sigma_3^3 = 1\rangle \\ &= B_4/\langle\sigma_1^3\rangle^N. \end{aligned}$$

The GAP code used to compute a coset table for the cosets of $B_4/\langle\sigma_1^3\rangle^N$ is given below:

```
gap> f := FreeGroup(3);;
gap> b4 := f / [f.1*f.3*f.1^-1*f.3^-1, f.1*f.2*f.1*f.2^-1*f.1^-1*f.2^-1,
                f.2*f.3*f.2*f.3^-1*f.2^-1*f.3^-1];;
gap> n4t3 := NormalClosure(b4, Subgroup(b4, [b4.1^3]));;
gap> table := CosetTable(b4, n4t3);;
gap> PrintArray(TransposedMat(table));
```

$$\begin{array}{rllllllll}
 & & \sigma_1 & \sigma_1^{-1} & \sigma_2 & \sigma_2^{-1} & \sigma_3 & \sigma_3^{-1} & \\
1 & [\ [& 2, & 3, & 4, & 5, & 6, & 7 &], \\
2 & \ \ \ [& 3, & 1, & 8, & 9, & 10, & 11 &], \\
3 & \ \ \ [& 1, & 2, & 12, & 13, & 14, & 15 &], \\
4 & \ \ \ [& 16, & 17, & 5, & 1, & 18, & 19 &], \\
5 & \ \ \ [& 20, & 21, & 1, & 4, & 22, & 23 &], \\
6 & \ \ \ [& 10, & 14, & 24, & 25, & 7, & 1 &], \\
7 & \ \ \ [& 11, & 15, & 26, & 27, & 1, & 6 &], \\
8 & \ \ \ [& 28, & 29, & 9, & 2, & 30, & 31 &], \\
9 & \ \ \ [& 32, & 33, & 2, & 8, & 34, & 35 &], \\
10 & \ \ \ [& 14, & 6, & 36, & 37, & 11, & 2 &], \ldots
\end{array}$$

Figure 3.2: First 10 rows of the $B_4/\langle\sigma_1^3\rangle^N$ coset table from the Todd-Coxeter algorithm

The first 10 rows of this output array are shown in Table 3.2, with line numbers included for clarity. The printed array contains the information needed to construct a representative for each coset by using the methods outlined in Section 2.2.

For example, to compute a shortest representative for coset 36, C_{36}, using Figure 3.2, we can see that coset 10, C_{10}, is mapped to C_{36} under right multiplication by σ_2; coset 2, C_2, is mapped to C_{10} under right multiplication by σ_3; and coset 1, $C_1 = 1_{Q_4}$ goes to C_2 under right multiplication by σ_1. Thus a shortest representative for C_{36} is $\sigma_1\sigma_3\sigma_2$.

To construct a list with a shortest representative for each coset, we wrote a program in Python that iteratively built up a list with the shortest representative for each coset. The program takes the representative for the current row from the existing list of representatives, appends the appropriate element for each column, and then saves the representative corresponding to that column entry in the next open position of the representative list if the coset in that column entry does not already have a representative stored in the list of representatives. The code used for 4-strand braids is in Appendix A.1, with generator values stored by their subscript (with a negative for inverses) for later computational processing. For example, the representative $\sigma_1\sigma_3\sigma_2$ for coset 36 in B_4 found in the previous paragraph is stored as `[ 1, 3, 2]` in the list of representatives.

For the 4-strand braid group quotient, each of the 648 cosets has a representative of length at most 9; that is, each coset has a member whose braid closure is an oriented link diagram with at most 9 crossings. Thus, by Theorem 1.1(c), each of these links is $t_3, \overline{t_4}$ equivalent to an unlink, and as the closure of every 4-strand braid is t_3 equivalent to one of these links, we have Theorem 3.2.

Theorem 3.2. *Every oriented link that can be represented as an oriented braid on at most 4 strands is $t_3, \overline{t_4}$ equivalent to an unlink.*

Remark 3.3. This result was announced previously [Che00, Theorem 5.1(b)] but we include our proof here to shed light on the methods used in Chapter 4. A version of Theorem 4.1 was also previously announced in weaker form as it excludes six $t_3, \overline{t_4}$ equivalence classes [Che00, Theorem 5.1(c)]. There are also oversights in the proof of [Che00, Theorem 5.1(c)], namely when changing from unoriented to oriented links. Our proof addresses these oversights.

Chapter 4

Proof of main theorem

In this chapter we build on the results of Chapter 3 to prove the main theorem, restated here.

Theorem 4.1. *The closures of all oriented 5-braids, except possibly for those $t_3, \overline{t_4}$ equivalent to the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$, are $t_3, \overline{t_4}$ equivalent to an unlink.*

In the case of 5-strand braids we start with 155,520 elements in Q_5, which has presentation

$$\begin{aligned} Q_5 = \langle \sigma_1, \sigma_2, \sigma_3, \sigma_4 \quad & |\sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2, \sigma_2\sigma_3\sigma_2 = \sigma_3\sigma_2\sigma_3, \sigma_3\sigma_4\sigma_3 = \sigma_4\sigma_3\sigma_4, \sigma_1\sigma_3 = \sigma_3\sigma_1, \\ & \sigma_1\sigma_4 = \sigma_4\sigma_1, \sigma_2\sigma_4 = \sigma_4\sigma_2, \sigma_1^3 = 1, \sigma_2^3 = 1, \sigma_3^3 = 1, \sigma_4^3 = 1\rangle \\ = B_5/\langle \sigma_1^3 \rangle^N & \end{aligned}$$

Of these, 68,192 of the cosets have a representative of length at most 11 and are thus handled by Theorem 1.1(c). The remaining 87,328 have a shortest representative of length between 12 and 20. Since the case for the closures of oriented braids on at most 4 strands has already been handled by Theorem 3.2, any cosets with a representative that misses one of the σ_i generators has already been shown to be $t_3, \overline{t_4}$ equivalent to an unlink. This is because missing a σ_1 or σ_4 means we have a disjoint union of an unknotted component with the closure of a 4-strand braid, while missing a σ_2 or σ_3 means we have the disjoint union of the closure of a 2-strand braid with the closure of a 3-strand braid. There are 5,718 such cosets with representative of length at least 12. This leaves 81,680 cosets that must still be considered.

We first show that all oriented links formed from the closures of 5-strand braids with at most 14 crossings are $t_3, \overline{t_4}$ equivalent to unlinks, and then use conjugacy classes to extend the result to all but one $t_3, \overline{t_4}$ equivalence class of links formed from the closure of 5-strand braids. The method used to analyze the closures of 5-strand braids is laid out in a flowchart in Figure 4.1. In these paragraphs we have covered the first three boxes of this flowchart.

Note that since the procedure above finds shortest-length representatives, reducing a coset representative in length means that the new element must represent a coset that the inductive process has shown is $t_3, \overline{t_4}$-equivalent to an unlink. That is, it cannot lie in a coset in which our shortest representative has higher length, because all elements of that coset have length at least that of the representative.

4.1 First pass elimination: cyclic permutation

We deal with the remaining 81,680 cases by an induction argument using previous results and properties of cosets. For the base case, we first show that all of the given coset representatives of length 12 have braid closures which can be reduced to the closures of braid words of length 11 or less using the t_3 and $\overline{t_4}$ moves, and thus their corresponding braid closures are $t_3, \overline{t_4}$ equivalent to an unlink. For the inductive step, it then suffices to show that for a given coset representative of length n, we can reduce the length by at least one by using $t_3, \overline{t_4}$ moves. This reduction implies the closure of the given representative is $t_3, \overline{t_4}$ equivalent to the closure of an element in a coset that has a shortest representative

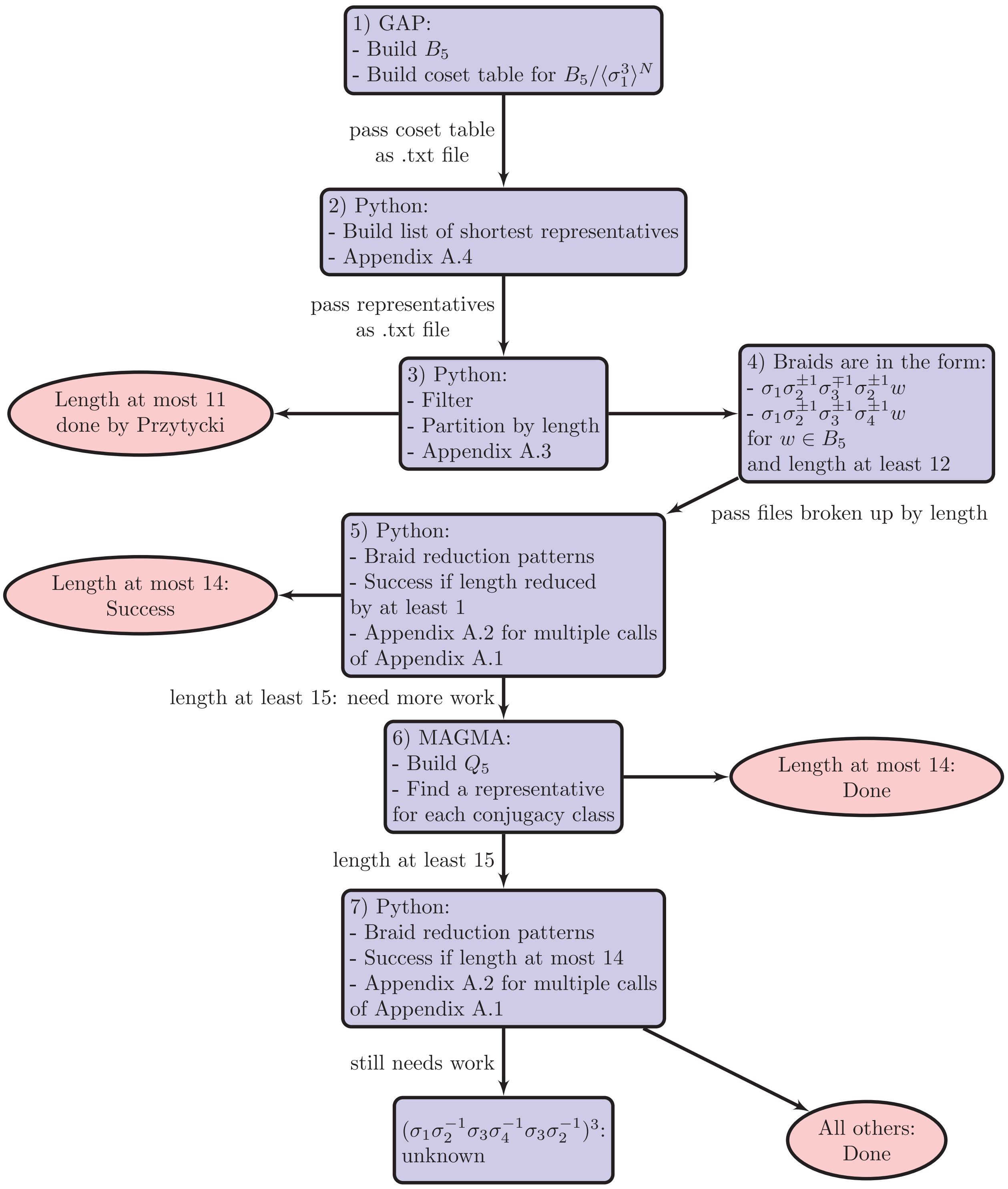

1) GAP:
- Build B_5
- Build coset table for $B_5/\langle\sigma_1^3\rangle^N$
pass coset table as .txt file
2) Python:
- Build list of shortest representatives
- Appendix A.4
pass representatives as .txt file
3) Python:
- Filter
- Partition by length
- Appendix A.3
Length at most 11 done by Przytycki
4) Braids are in the form:
- $\sigma_1\sigma_2^{\pm1}\sigma_3^{\mp1}\sigma_2^{\pm1}w$
- $\sigma_1\sigma_2^{\pm1}\sigma_3^{\pm1}\sigma_4^{\pm1}w$
for $w \in B_5$
and length at least 12
pass files broken up by length
5) Python:
- Braid reduction patterns
- Success if length reduced by at least 1
- Appendix A.2 for multiple calls of Appendix A.1
Length at most 14: Success
length at least 15: need more work
6) MAGMA:
- Build Q_5
- Find a representative for each conjugacy class
Length at most 14: Done
length at least 15
7) Python:
- Braid reduction patterns
- Success if length at most 14
- Appendix A.2 for multiple calls of Appendix A.1
still needs work
$(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$: unknown
All others: Done

Figure 4.1: Flowchart of 5-strand braid methods

of length at most $n-1$ which is $t_3, \overline{t_4}$ equivalent to an unlink by induction.

Because we are primarily concerned with the oriented knot (or link) that arises from a given braid word, without loss of generality we need only consider braids that begin with $\sigma_1^{\pm 1}$. This is because when closed to form a link, the braid $\sigma_i^{\pm 1}wu$ yields the same link as the braid $u\sigma_i^{\pm 1}w$ of the same length, where $u, w \in B_5$. Moreover, since t_3 and $\overline{t_4}$ moves can be replicated on the mirror image of any link, we can further restrict to braids beginning with σ_1. Additionally, we need only consider when the second element of the braid is $\sigma_2^{\pm 1}$, since $\sigma_3^{\pm 1}$ and $\sigma_4^{\pm 1}$ commute with σ_1 and thus can commute from the second position to the head of the braid and then cycle to the end, and $\sigma_1^{\pm 1}$ as the second element would allow a reduction using a Reidemeister II move or a t_3 move. Note a $\sigma_2^{\pm 1}$ must occur within the braid word by the logic in the preceding section. Similarly, commuting and reduction by a Reidemeister II move or a t_3 move imply that the cases where the third element is $\sigma_4^{\pm 1}$ or $\sigma_2^{\pm 1}$ are already covered. Thus we need only consider braids of the forms $\sigma_1\sigma_2^{\pm 1}\sigma_1^{\pm 1}w$ or $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\pm 1}w$, where $w \in B_5$.

The first of these two cases can further be reduced: if a braid is of the form $\sigma_1\sigma_2\sigma_1^{\pm 1}w$ or $\sigma_1\sigma_2^{-1}\sigma_1^{-1}w$, a Reidemeister III move allows this to be rewritten as $\sigma_2^{\pm 1}\sigma_1\sigma_2 w$ or $\sigma_2^{-1}\sigma_1^{-1}\sigma_2 w$, respectively. The leading $\sigma_2^{\pm 1}$ can then be cycled to the end of the braid, and we are left with a braid of the form $\sigma_1\sigma_2 w\sigma_2^{\pm 1}$ or $\sigma_1^{-1}\sigma_2 w\sigma_2^{-1}$. In the former case, this is a braid already under consideration; in the latter, the mirror image is a braid already under consideration. Therefore we can restrict our attention to braids in B_5 of the forms $\sigma_1\sigma_2^{-1}\sigma_1 w$ and $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\pm 1}w$, where the exponents in the latter form can be independently chosen.

By considering the fourth entry, we can narrow the possibilities down to $\sigma_1\sigma_2^{-1}\sigma_1\sigma_3^{\pm 1}w$, $\sigma_1\sigma_2\sigma_3^{-1}\sigma_2 w$, $\sigma_1\sigma_2^{-1}\sigma_3\sigma_2^{-1}w$, and $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\pm 1}\sigma_4^{\pm 1}w$ where the signs on the exponents are independently chosen. The first case comes from further examining $\sigma_1\sigma_2^{-1}\sigma_1 w$ for $w \in B_5$. If the first generator in w is $\sigma_1^{\pm 1}$ then either a t_3 or inverses can be used to simplify; if the first generator is a σ_2 a Reidemeister III move can be used followed by a t_3 reduction;

if the first generator is σ_2^{-1} then an (A1) move (see Section 4.3) can be used to move a σ_2 to the head, which can then be cycled to the end; and if w begins with a $\sigma_4^{\pm 1}$ then this can be commuted to the beginning of the braid then cycled to the end. The latter cases arise from $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\pm 1}w$. In this case, if w begins with $\sigma_1^{\pm 1}$ we can commute to get $\sigma_1\sigma_2^{\pm 1}\sigma_1^{\pm 1}\sigma_3^{\pm 1}w'$ for some $w' \in B_5$, which has already been considered. If w begins with $\sigma_3^{\pm 1}$, then a Reidemeister II move or a t_3 move can be used to reduce. If w begins with a $\sigma_2^{\pm 1}$ and the sign on σ_3 matches the sign on either occurrence of $\sigma_2^{\pm 1}$, then a Reidemeister III move can be used, followed by commuting, to push a σ_3 to the head of the braid. Thus if w begins with a $\sigma_2^{\pm 1}$, we need only consider when the signs on the σ_2's agree and are opposite to the sign on σ_3; i.e., the braid must begin with $\sigma_1\sigma_2\sigma_3^{-1}\sigma_2$ or $\sigma_1\sigma_2^{-1}\sigma_3\sigma_2^{-1}$.

This reduces the number of braids to consider from 81,610 down to 7,971 and covers the fourth box in the flowchart in Figure 4.1.

4.2 Braid reduction in Python

We wrote a program in Python that, upon input of a braid word from B_5, iteratively constructs conjugate and $t_3, \overline{t_4}$ equivalent braids using a set of replacement rules. If the program finds a braid of shorter length (which is $t_3, \overline{t_4}$ equivalent to an unlink by induction), then the input braid has braid closure that is $t_3, \overline{t_4}$ equivalent to an unlink. In Section 4.3, the six types of replacement rules (A1)-(A6) used by this program are identified and explained. The complete Python code can be found in Appendix A.4.

While ideally, the program would find every braid that is conjugate and $t_3, \overline{t_4}$ equivalent to the input braid, this is not computationally possible due to limitations on computer resources. In particular, allowing all commutation rewritings permitted within the braid group (i.e., rewritings of the form $\sigma_i\sigma_j \to \sigma_j\sigma_i$ where $|i - j| \geq 2$) significantly increases computation time, and allowing length increases by a t_3 move ($\sigma_i^{\pm 1} \to \sigma_i^{\mp 2}$) or a crossingincreasing Reidemeister II move (inserting $\sigma_i^{\pm1}\sigma_i^{\mp1}$) in general creates an infinite family of t_3 equivalent links, as the same location can be repeatedly expanded. As a compromise, targeted t_3 increases were included (see (A6)).

A targeted commuting function `lookLR` (within Appendix A.4) was also created to utilize commutation rules to enable the (A1) - (A6) replacement rules. If looking at a replacement rule of the form $\alpha u \beta \to v$ for $\alpha, \beta \in \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}^{\pm1}$ and $u, v \in B_5$, the function works as follows: first, the function tries to locate u as a subword of the braid word. Upon finding the subword u, the function looks at the braid entries prior to u to find the last occurrence of α before u. The function then attempts to use commutation rules to move α to immediately before u, so that αu is a subword of the new braid word and the new braid is equivalent to the original. If successful, the program then looks at the braid entries after αu to find the first occurrence of β. As before, the program then uses commutation rules to try to move β to immediately after αu so that $\alpha u \beta$ is a subword of the new braid word and the new braid word is equivalent to the original.

Using the replacement rules in Chapter 4.3, the program successfully reduced the length of the coset representatives in the constructed transversal of length 12, 13, and 14 by one, except for 2 representatives of length 14. Those representatives are:

(1) `1 2-3 2 1 4-3-2 1 4 3-2 3 4` (2) `1-2 3-2 1-4 3-2 1-4 3-2 3-4`

These representatives were reduced by hand, using the same replacement rules used in the Python program but with more leniency on commuting. A reduction for (1) is shown below; (2) can be handled similarly.

1 2 −3 2 1 4 −3 −2 1 4 3 −2 3 4	Initial braid (1) (14 crossings)
−3 2 1 4 −3 −2 1 4 3 −2 3 4 **1** **2**	Cyclic permutation of first two entries
−3 2 **4** **−3** **1** −2 1 4 3 −2 3 4 1 2	commuting
−3 2 4 −3 1 −2 **−1** **−1** 4 3 −2 3 4 1 2	t_3 expansion on second 1 (15 crossings)
−3 2 4 −3 1 −2 −1 **4** **3** **−1** −2 **1** **3** **2** **4**	commuting
−3 2 4 −3 **−2** **−1** **2** 4 3 **2** **−1** **−2** 3 2 4	two RIII moves
−3 **4** **2** −3 −2 −1 **4** **2** 3 2 −1 −2 3 2 4	commuting
−3 4 **−3** **−2** **3** −1 4 **3** **2** **3** −1 **3** **2** **−3** 4	three RIII moves
−3 4 −3 −2 3 −1 4 3 2 **−1** **3** 3 2 −3 4	commuting
−3 4 −3 −2 3 −1 4 3 2 −1 **−3** 2 −3 4	t_3 reduction (14 crossings)
−3 **4** −3 4 −3 −2 3−1 4 3 2 −1 −3 2	cycle last two entries to front
−4 **3** **−4** −2 3 −1 4 3 2 −1 −3 2	(A2) move at head (12 crossings)

When the 1,766 braids of length 15 were input into the program, all but 13 were able to be quickly (less than 4 minutes each) reduced in length by at least one. Similar results were found with the braids of length 16. We thus get Theorem 4.2 and have covered the first five boxes of the flowchart in Figure 4.1.

Theorem 4.2. *Any oriented link that can be realized as the closure of a 5-strand braid with at most 14 crossings is $t_3, \overline{t_4}$ equivalent to an unlink.*

4.3 Local braid replacement moves as a consequence of $t_3, \overline{t_4}$

In this section we examine the consequences of the t_3 and $\overline{t_4}$ moves on braid closures and construct replacement rules based on these consequences. The patterns herein are written for oriented 5-braids and their closures but can be extended to the closures of

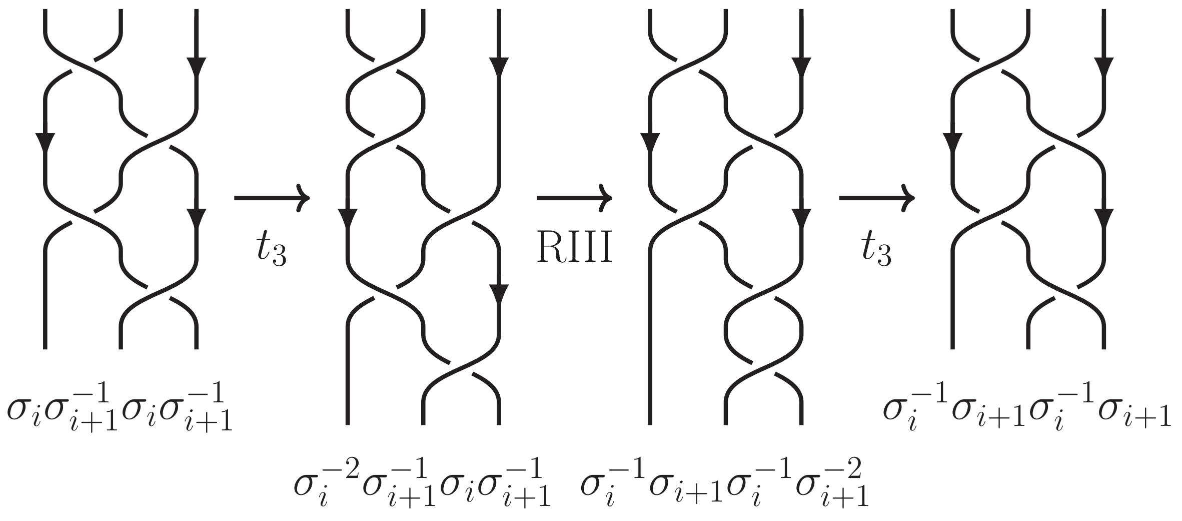

t_3
RIII
t_3
$\sigma_i\sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1}$
$\sigma_i^{-2}\sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1}$
$\sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1}\sigma_{i+1}^{-2}$
$\sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1}\sigma_{i+1}$

Figure 4.2: (A1) Example of consequence of t_3 in braids

oriented braids on 6 or more strands. These replacement rules are used in conjunction with Reidemeister moves in the proof in Section 4.2.

(A1) As a consequence of t_3 moves in conjunction with a Reidemeister III move we have, for $i = 1, 2, 3$:

$$\sigma_i\sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1} \leftrightarrow \sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1}\sigma_{i+1} \leftrightarrow \sigma_{i+1}\sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1} \leftrightarrow \sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1}\sigma_i$$

This follows from the (expansion) rewritings $\sigma_i^{\pm 1} \to \sigma_i^{\mp 2}$ due to a t_3 move, followed by one or two Reidemeister III moves and a t_3 collapse; expanding in the first, one of the middle two, or last position yields the three equalities. See Figure 4.3 for a visualization.

(A2) Due to rewriting (A1) and a t_3 move, the following replacement rules also apply for

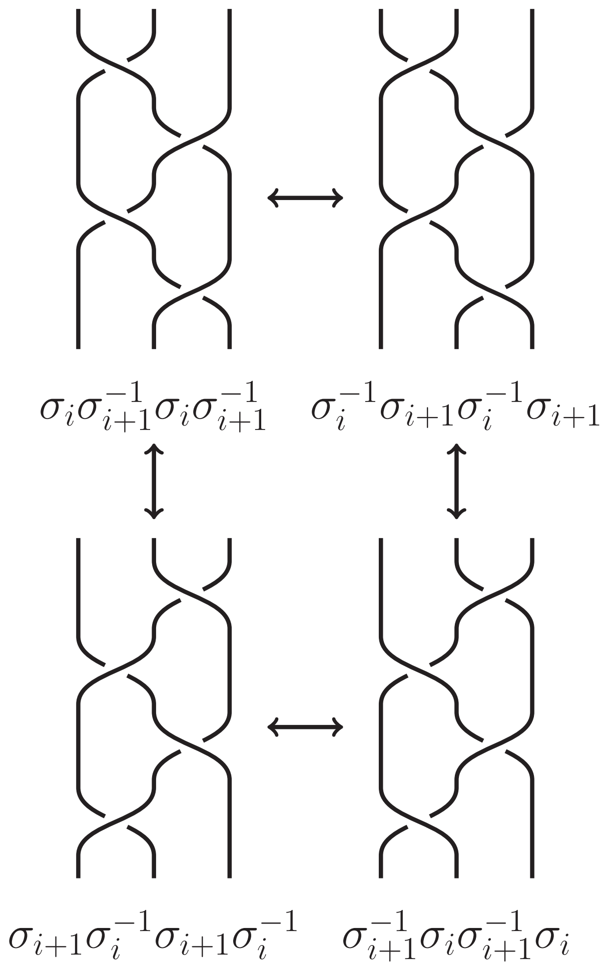

$\sigma_i\sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1}$
$\sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1}\sigma_{i+1}$
$\sigma_{i+1}\sigma_i^{-1}\sigma_{i+1}\sigma_i^{-1}$
$\sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}^{-1}\sigma_i$

Figure 4.3: (A1) Consequence of t_3 in braids

$i = 1, 2, 3$:

$$\sigma_i^{\pm1}\sigma_{i+1}^{\mp1}\sigma_i^{\pm1}\sigma_{i+1}^{\mp1}\sigma_i^{\pm1} \to \sigma_{i+1}^{\pm1}\sigma_i^{\mp1}\sigma_{i+1}^{\pm1}$$

$$\sigma_{i+1}^{\pm1}\sigma_i^{\mp1}\sigma_{i+1}^{\pm1}\sigma_i^{\mp1}\sigma_{i+1}^{\pm1} \to \sigma_i^{\pm1}\sigma_{i+1}^{\mp1}\sigma_i^{\pm1}$$

(A3) In the situation where σ_1 occurs precisely twice and σ_1^{-1} does not appear, $\overline{t_4}$ gives that the closure of the braid is equivalent to the closure of the same braid where σ_1 is replaced by σ_1^{-1} in both instances; see Figure 4.5. This follows as a $\overline{t_4}$ move in

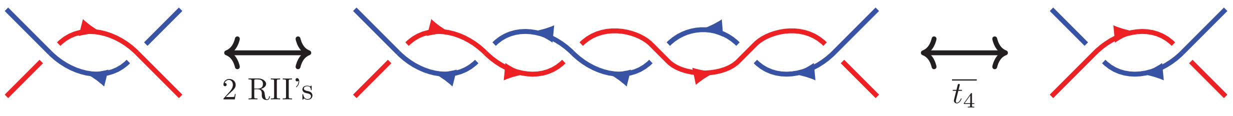


Figure 4.4: Utilizing $\overline{t_4}$ to change crossings on antiparallel bigons

conjunction with two Reidemeister II moves can be used on bigons with antiparallel orientation to change both crossings in the bigon. This equivalence is laid out in Figure 4.4. Similarly, if σ_1^{-1} occurs precisely twice and σ_1 does not appear, we can replace each σ_1^{-1} with σ_1. This gives us the following rewritings, where $w, u, v \in \{\sigma_2, \sigma_3, \sigma_4\}^{\pm 1*}$, i.e. w, u, and v are words over σ_2, σ_3, and σ_4 and their inverses:

$$w\sigma_1 u\sigma_1 v \leftrightarrow w\sigma_1^{-1}u\sigma_1^{-1}v$$

When working in B_5, if $\sigma_4^{\pm 1}$ appears precisely twice with no occurrences of $\sigma_4^{\mp 1}$, we can replace both $\sigma_4^{\pm 1}$ with $\sigma_4^{\mp 1}$, which gives the following rewritings with $w, u, v \in \{\sigma_1, \sigma_2, \sigma_3\}^{\pm 1*}$:

$$w\sigma_4 u\sigma_4 v \leftrightarrow w\sigma_4^{-1}u\sigma_4^{-1}v$$

(A4) To reduce computation time, a special case of the (A3) rewriting is also targeted for identification. In particular, if one of the patterns (A4.1)-(A4.4) appears and these are the only occurrences of $\sigma_1^{\pm 1}$ or $\sigma_4^{\pm 1}$, as in Figure 4.5, then a $\overline{t_4}$ move on the braid closure followed by a Reidemeister III move permits the braid to be destabilized (see Section 2.1). That is, a Reidemeister I move can be used to remove the sole $\sigma_1^{\pm 1}$ or $\sigma_4^{\pm 1}$ and thus reduce the braid index. Since the resulting braid is then a braid on 4 strands, the closure is $t_3, \overline{t_4}$ equivalent to an unlink by Theorem 3.2.

(A4.1) $\sigma_1^{\pm 1}\sigma_2^{\mp 1}\sigma_1^{\pm 1}\sigma_2^{\mp 1}$

(A4.2) $\sigma_2^{\pm 1}\sigma_1^{\mp 1}\sigma_2^{\pm 1}\sigma_1^{\mp 1}$

(A4.3) $\sigma_3^{\pm 1}\sigma_4^{\mp 1}\sigma_3^{\pm 1}\sigma_4^{\mp 1}$

(A4.4) $\sigma_4^{\pm 1}\sigma_3^{\mp 1}\sigma_4^{\pm 1}\sigma_3^{\mp 1}$

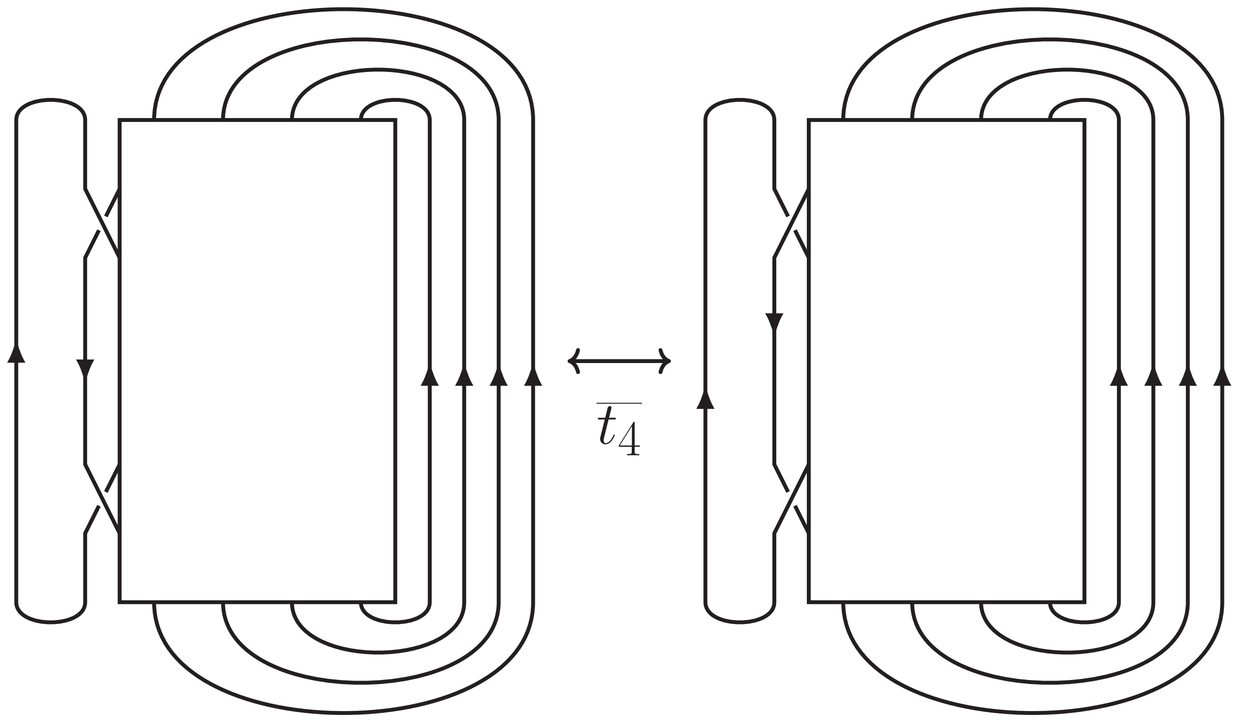


Figure 4.5: (A3) $\overline{t_4}$ move on braid closures

Note that without the alternating signs, the result still holds. When at least two neighboring signs agree, a Reidemeister III move can be used and so the program is already equipped to handle those patterns.

(A5) If we have precisely one occurrence each of σ_1 and σ_1^{-1}, then two Reidemeister II moves on the braid closure can be used to swap the signs on each; i.e., the σ_1 becomes a σ_1^{-1} and vice versa; see Figure 4.6. The same holds for σ_4 and σ_4^{-1} in B_5. This gives the following rewritings, where $w, u, v \in \{\sigma_2, \sigma_3, \sigma_4\}^{\pm 1*}$ and $x, y, z \in \{\sigma_1, \sigma_2, \sigma_3\}^{\pm 1*}$:

$$w\sigma_1 u\sigma_1^{-1}v \leftrightarrow w\sigma_1^{-1}u\sigma_1 v$$

$$x\sigma_4 y\sigma_4^{-1}z \leftrightarrow x\sigma_4^{-1}y\sigma_4 z$$

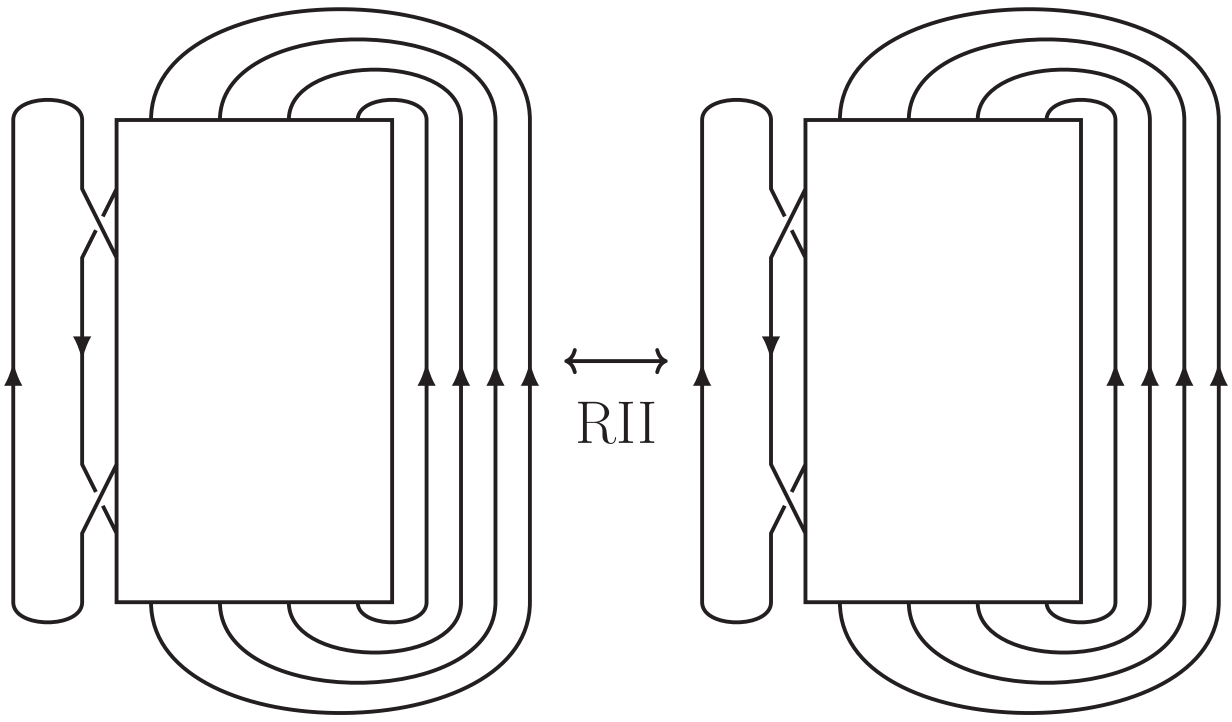


Figure 4.6: (A5) Reidemeister II move realized on braid closures

(A6) Targeted t_3 increases (i.e., rewrites of the form $\sigma_i^{\pm 1} \to \sigma_i^{\mp 2}$) are used to enable Reidemeister III moves, such as in the following examples. For a complete list, see the list `t3exps2` in Appendix A.4.

$$\sigma_1\sigma_2^{-1}\sigma_1\sigma_3\sigma_2^{-1}\sigma_1 \to \sigma_1\sigma_2^{-1}\sigma_1^{-2}\sigma_3\sigma_2^{-1}\sigma_1$$

$$\sigma_2\sigma_1^{-1}\sigma_2\sigma_3^{-1}\sigma_2 \to \sigma_2\sigma_1^{-1}\sigma_2^{-2}\sigma_3^{-1}\sigma_2$$

Rather than enforcing one or both RIII moves, this was kept as a single t_3 expansion to ensure more possibilities were examined.

These move sequences were all implemented in the code to search for braid length reductions.

4.4 5-strand braids of length at least 15

In order to handle the braids of length 15 and greater, we use conjugacy classes. Note that the closures of conjugate braids yield the same knot or link. This is because conjugating a braid is equivalent to a collection of Reidemeister II moves on a knot or link diagram. Using the Computational Algebra System MAGMA [BCP97], the 102 conjugacy classes of $Q_5 = B_5/\langle \sigma_1^3 \rangle^N$ are computed and representatives in the σ_i's are found in B_5, completing the sixth box of Figure 4.1.

Remark. In the code below, L is a permutation group isomorphic to Q_5, and $f : B_5 \to L$ a surjective group homomorphism.

```
magma> B5< a, b, c, d > := Group< a, b, c, d | a*c=c*a, a*d = d*a, b*d = d*b,
        a*b*a = b*a*b, b*c*b = c*b*c, c*d*c = d*c*d >;
magma> t3 := sub< B5 | a^3 >;
magma> T3 := t3^B5;
magma> f, L, K := CosetAction(B5, T3);
magma> cl := ConjugacyClasses(L);
```

Since MAGMA computes conjugacy classes through a permutation group, one element of the preimage of each class is found using the code `cl[i][3] @@ f;`. This element, a representative in B_5 of the ith conjugacy class, is not necessarily of shortest length, so an inductive argument as in Section 4.2 will not work here. For the conjugacy classes with a representative of length at most 14, we are done by Theorem 4.2; for the rest, it suffices to reduce the braid closure of the representative to the closure of a 5-strand braid of length at most 14 or to the closure of a braid on at most 4 strands. The representative output by MAGMA for a given conjugacy class is not necessarily unique – executed more than once, the same code will output different representatives, because MAGMA grabs an element in the preimage of the input conjugacy class at random. Due to this, the following results may vary by conjugation from other executions of the previous code snippets. In one execution of the code, the 102 representatives found in Appendix B were returned by MAGMA. Of these, 80 are of length at most 14, and the following 22 are of length 15 or more.

1. `[1,-2, 3,-4, 3,-2, 1,-2, 3,-4, 3,-2, 1,-2, 3,-4, 3,-2]`
2. `[1, 2, 1, 3,-4, 3,-2, 1, 3, 2, 4,-3, 2, 1, 4, 3,-2, 3,-4, 3]`
3. `[1, 2, 1, 4,-3,-2, 1,-4, 3,-2, 1, 3,-4, 3,-2, 1, 3,-2,-4,-3]`
4. `[1, 2, 1, 3,-4, 3,-2, 1, 3, 2, 4,-3, 2, 1, 4, 3,-2, 3,-4]`
5. `[1, 2, 1, 4,-3, 2, 1, 4,-3, 2, 1, 4,-3, 2, 1, 4,-3, 2, 4]`
6. `[1, 2, 1, 4,-3,-2, 1,-4, 3,-2, 1, 3,-4, 3,-2, 1, 3,-2,-4]`
7. `[1, 2,-4,-3, 2,-1, 2, 3,-4, 3,-2, 1, 3, 4,-2, 1,-3,-2,-4]`
8. `[2,-1, 2, 3,-4, 3, 2,-1, 2,-3, 2, 4,-3, 4,-2]`
9. `[2, 1, 3,-2,-4, 3,-2, 1,-2,-4, 3,-2, 1,-4, 3,-4]`
10. `[2,-1, 2, 3,-4, 3, 2,-1, 2, 3,-4, 3, 2,-1, 2, 3,-4]`
11. `[2,-1, 2,-3, 4,-3, 2,-1, 2,-3, 4,-3, 2,-1, 2,-3, 4]`
12. `[4, 3,-2,-1,-3, 2,-3, 4,-3, 2,-1, 2,-3, 4,-2]`
13. `[4, 3,-2, 3,-1, 2,-4, 3, 2,-1,-4, 3,-2, 3, 4]`
14. `[1,-2, 1,-3, 2,-3, 4,-3, 2,-1, 2,-3, 4,-3, 2]`
15. `[1, 2,-4,-3, 2, 1,-3,-2,-4, 3,-2, 1,-4, 3, 2]`
16. `[-1, 2, 3,-1,-4,-3, 2,-1,-3, 2, 4,-1,-3, 2,-4]`

17. `[1,-4,-3, 2,-1,-3, 2,-1,-4, 3, 2,-1,-4, 3, 2]`

18. `[1, 2, 3,-4,-3,-2, 1,-2,-3, 4,-3,-2, 1,-2, 3,-2]`

19. `[2,-1,-3, 2,-1,-3,-4,-3, 2,-1,-3, 2,-1,-3,-4,-3]`

20. `[1, 2,-1,-4, 3,-2, 1,-4, 3,-2, 1,-4, 3,-2, 3]`

21. `[4,-1, 2,-3, 4,-3, 2,-1, 2,-3, 2, 1, 4,-3,-2, 1]`

22. `[-4,-3, 2,-1, 2,-3, 4,-3,-2, 1,-2,-3, 4,-3, 2,-1]`

Note that many these representatives can be shortened using conjugation. Representatives 8, 12, and 13 can be shortened to length at most 14 by conjugation by the inverse of the leading element, followed by (commuting and) a reduction using inverses or a t_3 move; representative 21 can likewise be reduced by conjugating by the final element and commuting. Representative 14 can similarly be reduced by conjugating by the final element, then using an RIII move to enable a t_3 reduction. As a result of these observations, we compare the cyclic permutations of the conjugacy class representatives in the previous list with the coset table output using Todd-Coxeter to see which coset each fell into. This was accomplished using the `TracedCosetFpGroup` method in GAP, which automates tracing each word through the coset table built with the Todd-Coxeter algorithm and outputs the coset number. These coset numbers were then compared to the list of shortest representatives computed using the code in Appendix A.1. All but five are cyclically equivalent to an element of a t_3 equivalence class with a shortest representative of length at most 14; cosets 1, 2, 3, 10, and 11 in the above list were the exceptions. Cosets 2, 3, 10, and 11 were successfully reduced to braids of at most 14 crossings by using moves (A1)-(A6), completing the proof of Theorem 4.1.

Theorem 4.1. *The closures of all oriented 5-braids, except possibly for those $t_3, \overline{t_4}$ equivalent to the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$, are $t_3, \overline{t_4}$ equivalent to an unlink.*

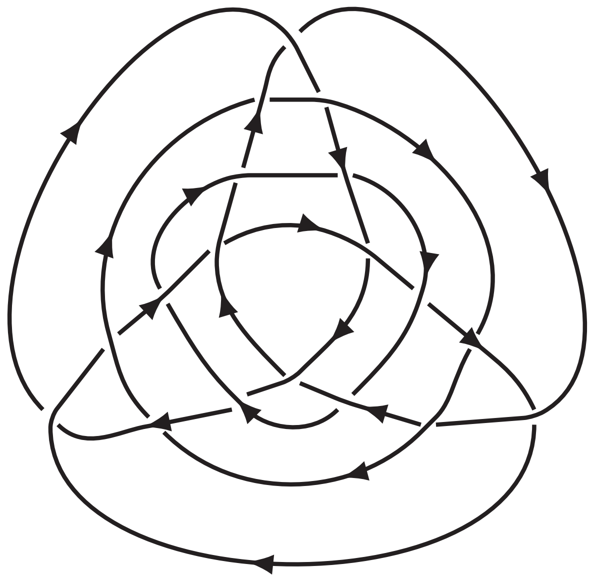


Figure 4.7: The closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$

Corollary 4.3. *If the closure of $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$ is $t_3, \overline{t_4}$ equivalent to an unlink, then the closure of every oriented 5-braid is $t_3, \overline{t_4}$ equivalent to an unlink.*

4.5 The braid $(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$

The closure of the remaining 5-strand braid $\beta = (\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^3$ is a 5-component link with 18 crossings. A diagram is given in Figure 4.7. To determine whether the closure of this braid is $t_3, \overline{t_4}$ equivalent to an unlink, we need to utilize other methods.

One of these methods uses the homology of the 3-fold cyclic cover of the knot complement. By using t_3 moves on the first four crossings of the closure of β, we obtain the 22-crossing knot that is the closure of $\beta' = \sigma_1^{-2}\sigma_2^{2}\sigma_3^{-2}\sigma_4^{2}\sigma_3\sigma_2^{-1}(\sigma_1\sigma_2^{-1}\sigma_3\sigma_4^{-1}\sigma_3\sigma_2^{-1})^2$. Using SnapPy [CDGW], we compute the 3-fold cyclic cover $M^{(3)}_{c(\beta')}$ of the complement of the closure $c(\beta')$ of β' and compute the first homology with $\mathbb{Z}$-coefficients. One should note that attempts were made to simplify this knot first using SnapPy; however, no crossing reductions were found.

```
In[1]: L = Link(braid_closure = [−1, −1, 2, 2, −3, −3, 4, 4, 3,
                   −2, 1, −2, 3, −4, 3, −2, 1, −2, 3, −4, 3, −2])
In[2]: M = L.exterior()
In[3]: covs = M.covers(3)              # Obtain the 3−fold covers
In[4]: M3 = covs[0]                    # Get the cyclic cover
In[5]: M3.homology()                   # Compute the first homology
Out[5]: Z/2 + Z/2 + Z/2 + Z/2 + Z/2 + Z/2 + Z/182 + Z/182 + Z
```

SnapPy yields that $H_1(M^{(3)}_{c(\beta')}, \mathbb{Z}) = (\mathbb{Z}/2)^6 \oplus (\mathbb{Z}/182)^2 \oplus \mathbb{Z}$. Then, using the short exact sequence from Section 2.3 we find that $H_1(M^{(3)}_{c(\beta')}, \mathbb{Z}/2) \cong (\mathbb{Z}/2)^8$. The first homology of the 3-fold cyclic branched cover is preserved by t_3 moves [Prz88a, Theorem 1] and $\overline{t_4}$ moves [Prz88b, Theorem 3.2]. Thus, as $\dim H_1(M^{(3)}_{U_n}, \mathbb{Z}/2) = 2(n-1)$ where U_n is the n-component unlink, we have that if $c(\beta')$ is $t_3, \overline{t_4}$ equivalent to an unlink, it must be equivalent to U_5, the unlink of 5 components. As $c(\beta')$ is t_3 equivalent to the closure of β, it follows that if the closure of β is $t_3, \overline{t_4}$ unlinkable, then it too must be equivalent to the 5-component unlink.

Chapter 5

Extensions

5.1 Extending to B_6

In Chapter 3, we proved the $t_3, \overline{t_4}$-conjecture for the closures of all oriented 4-braids and in Chapter 4 we proved the $t_3, \overline{t_4}$-conjecture the closures of all but one conjugacy class of oriented 5-braids. In this section, we extend those results to the closures of oriented 6-braids. Although the quotient $Q_6 = B_6/\langle\sigma_1^3\rangle^N$ is infinite by a result of Coxeter [Cox57], for a given braid word length there are only finitely many 6-strand braids. Thus we utilize the induction argument from 5-braids and begin by examining 6-braids with 12 crossings.

The logic in Section 4.1 regarding the head of these braids still holds. Since we are concerned with the links resulting from the closures of these braids, we again may assume without loss of generality that the braids begin with one of $\sigma_1\sigma_2^{-1}\sigma_1\sigma_3^{\pm 1}$, $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\mp 1}\sigma_2^{\pm 1}$, or $\sigma_1\sigma_2^{\pm 1}\sigma_3^{\pm 1}\sigma_4^{\pm 1}$. This reduces the number of 12 crossing 6-strand braids we need to consider from 10^{12} braids to 11×10^8 braids. Furthermore, since $\sigma_i^{\pm 1}\sigma_i^{\pm 1}$ may be reduced using either a t_3 move (if both exponents agree) or due to a Reidemeister II move (if the exponents have opposite signs), we have that if a given entry is $\sigma_i^{\pm 1}$ then the following entry must be $\sigma_j^{\pm 1}$ for $j \in \{1,2,3,4,5\}\backslash\{i\}$. This again reduces the number of 12-strand braids we need to consider down to 184,549,376, less than 0.02% of the original number.

To enumerate and simplify these braids, we wrote a program in Python to run within

SnapPy that (a) creates each of these braids, (b) tries to simplify the closure of each braid within SnapPy, and (c) only keeps the braid word of the simplified closure if the result has 12 crossings and is a braid on 6 strands. That is, if the braid closure can be simplified to at most 11 crossings or is the (connect sum or disjoint union of) closure of a braid on at most 5 strands, then Theorem 4.1 applies. The enumeration script is included in Appendix A.5. After this enumeration, approximately 13,577,568 braids are left and are then passed to the Python program in Appendix A.4, which was extended to work for 6-strand braids. These were all successfully reduced, leading to the following theorem.

Theorem 5.1. *All oriented links which result from the closure of a 6-strand braid with at most 12 crossings are* $t_3, \overline{t_4}$ *unlinkable.*

5.2 Extending the method to other local moves

By viewing a local move on diagrams as a local move on braids and their closures, the methods outlined in Chapter 3 can be extended to apply to moves other than t_3 and $\overline{t_4}$. That is, a system of replacement rules based on a given local move can be developed to iteratively reduce braid words and thus determine if the move is an unlinking move on the braid group, as long as a finite quotient of a braid group can be found.

One example to consider is the delta move shown in Figure 5.1. Within the braid group B_n with $n \geq 3$, this move gives rise to the relations $(\sigma_i \sigma_{i+1}^{-1})^3 = 1$ for $1 \leq i \leq n-2$. However, neither GAP nor MAGMA can compute the quotient group $B_3/\langle(\sigma_1 \sigma_2^{-1})^3\rangle^N$, and so we cannot get a transversal to begin rewrites on.

In extending this method to the 3-move, we again use the quotient $B_n/\langle\sigma_1^3\rangle^N$ of the nth braid group, but disregard moves that are consequences of $\overline{t_4}$ (i.e., move (A3) in Chapter 4.3). For oriented 4-strand braids, this extension is successfully able to reduce all 647 non-trivial coset representatives by at least one crossing and thus the closures of all 4-strand

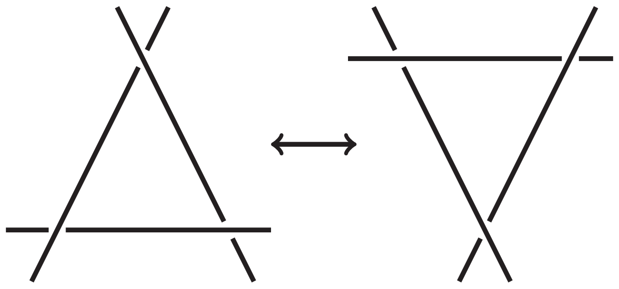


Figure 5.1: The delta move

braids are 3-move equivalent to an unlink. This reproduces a result of [Che00]. For 5-strand braids, we observe that each generator must appear at least twice: if any generator is missing, the braid closure in question is a split link of closures of braids on at most 4 strands; if any generator appears exactly once, the braid closure is a connect sum of braids on at most 4 strands. Unfortunately, an analog to the replacement rule (A3) cannot be realized on braids, as utilizing a $\overline{t_3}$ move in this situation results in a diagram which is not immediately realizable as a braid. The issue here is with the inherited orientation of the strands; an example is shown in Figure 5.2. Although the result is a diagram of the unknot, it is not immediately a braid diagram and, when attempting to preserve orientation, a conflict arises.

One should note that Chen wrote about the 3-move on 4- and 5-strand braids in [Che00]. As explained in Remark 3.3, the results in [Che00] regarding the 3-move on unoriented links are based on results regarding $t_3, \overline{t_4}$ on oriented links. Thus his results require further examination.

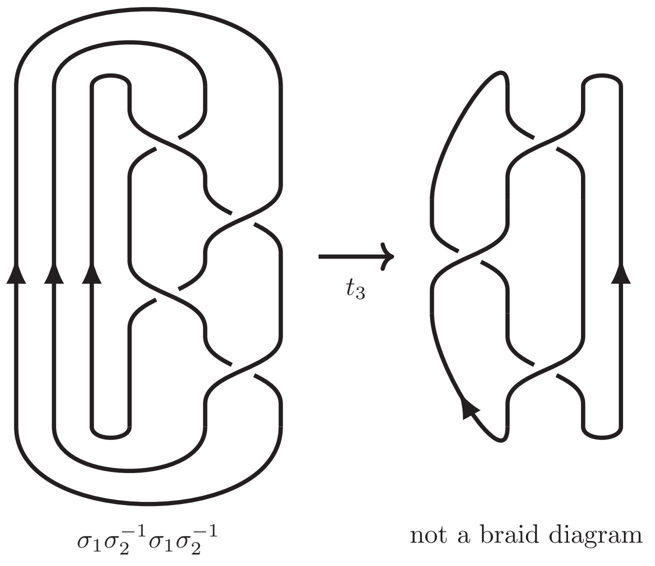


Figure 5.2: An $\overline{t_3}$ move resulting in a non-braid diagram

Appendix A

Python code

A.1 Coset Computation

This Python program builds a representative of shortest length for each of the 648 cosets in the group $B_4/\langle\sigma_1^3\rangle^N$, utilizing the coset table output by the `CosetTable` method in GAP. Modifications were made to build representatives for the group $B_5/\langle\sigma_1^3\rangle^N$.

```
import numpy as np
import sys
## generators
gens = [' 1', '-1', ' 2', '-2', ' 3', '-3']

#Read in coset table output from GAP into variable 'file'
data = []
for line in file:
    linedata = []
    for j in range(6):
        linedata.append(int(line[3+6*j:8+6*j]))
    linedata = np.asarray(linedata)
    data.append(linedata)
```

```
data = np.asarray(data)


## Create label output
cosetReps = np.empty(648, dtype = 'object')
cosetReps[0] = 'id'
for j in range(6):
    cosetReps[data[0][j]-1] = str(gens[j])
for i in range(1, 648):
    for j in range(6):
        if cosetReps[data[i][j]-1] == None:
            cosetReps[data[i][j]-1] = cosetReps[i]+gens[j]
```

A.2 First pass elimination (B5_firstpass.py)

This Python program runs on the output from the code in Appendix A.1 and saves only those beginning with the words from Section 4.1. It also partitions the saved representatives by length so that an induction argument can be utilized.

```
# -*- coding: utf-8 -*-
"""
Program to eliminate some braids: can reduce to braids that start
  with one of:
    - [ 1,-2, 1,+/-3, ...]
    - [ 1,+/-2,-/+3,+/-2, ...]
    - [ 1,+/-2,+/-3,+/-4, ...]
  and are of length at least 12
"""

fileloc = "C://Users//tenno//OneDrive//Documents//PythonScripts//"
```

```
def saveLine(line):
    if len(line) == 25: st = "12"
    elif len(line) == 27: st = "13"
    elif len(line) == 29: st = "14"
    elif len(line) == 31: st = "15"
    elif len(line) == 33: st = "16"
    elif len(line) == 35: st = "17"
    elif len(line) == 37: st = "18"
    elif len(line) == 39: st = "19"
    elif len(line) == 41: st = "20"
    with open(fileloc + "firstpass_elim_v3//reps" + st + ".txt", 'a') as f:
        f.write(line)
    f.close()

infile = open(fileloc + "b5_output_raw.txt", 'r')
outstring = ""
for line in infile:
    if (int(line[:2]) == 1) and (len(line[:-1]) > 22):
        if (int(line[2:4]) == -2) and (int(line[4:6]) == 1) and
           (int(line[7]) == 3):
            saveLine(line)
        elif (int(line[3]) == 2) and (int(line[5]) == 3) and
             (int(line[7]) == 4):
            saveLine(line)
        elif (int(line[2:4]) == 2) and (int(line[4:6]) == -3) and
             (int(line[6:8]) == 2):
            saveLine(line)
        elif (int(line[2:4]) == -2) and (int(line[4:6]) == 3) and
             (int(line[6:8]) == -2):
            saveLine(line)
```

```
infile.close()
```

A.3 Braid rewriting wrapper (B5_simplify.py)

This Python program reads the output files from the first pass elimination code (Appendix A.2) line by line – i.e., braid by braid – and runs the braid rewriting code (Appendix A.4) on each braid. It stores successes and failures separately: for successes, both the original and successfully reduced braid words are stored so that the program can be checked for accuracy; for failures, the final braid list formed by the braid rewriting code is stored for reference.

```
# -*- coding: utf-8 -*-
import braidreductions
import csv
import datetime
# Initialize lists
fails = [] # stores original braid word where program could not simplify
successes = [] # stores original and reduced braid

def toBraid(line):
    """Function to cast each line as a braid"""
    if line[-1] == "\n":
        line = line[:-1]
    length = len(line)
    if length%2 != 0:
        return False
    braid = []
    for i in range(0, length, 2):
        braid.append(int(line[i:i+2]))
    return braid
```

```
def toStr(braid, addLine):
    """recast a braidword as a string for saving"""
    if type(braid[0]) != int:
        out = ''
        for i in range(len(braid)):
            out.join(toStr(braid, False))
        return out
    string = ''
    for gen in braid:
        if gen > 0:
            string += " " + str(gen)
        else:
            string += str(gen)
    if addLine:
        string += '\n'
    return string

st = "15" #value between 12 and 20
file = open("C:\\Users\\tenno\\OneDrive\\Documents\\PythonScripts\\firstpass_elim\\reps" +
fileloc = "C:\\Users\\tenno\\OneDrive\\Documents\\PythonScripts\\v12Outputs_march5"

# Variable to keep track of progress
ct = 0

print("Start at " + str(datetime.datetime.now()))

for line in file:
    braid = toBraid(line)
    compbraid = braid[:]
    braidlist, tf = braidreductions.findthebraids(compbraid)
    if not tf: fails.append([toStr(braid, False), str(braidlist), '\n'])
    else: successes.append([toStr(braid, False), toStr(braidlist[0], True)])
    ct += 1
    if ct%50 == 0: print(ct)
file.close()

print("End at " + str(datetime.datetime.now()))

# Save fails, successes to files
with open(fileloc + '\\fails_nored_' + st + '.txt', 'w') as f:
    writer = csv.writer(f, delimiter = " ")
    writer.writerows(fails)
with open(fileloc + '\\successes_red_' + st + '.txt', 'w') as f:
    writer = csv.writer(f, delimiter = " ")
    writer.writerows(successes)
```

A.4 Braid rewriting (braidreductions.py)

This is the Python program that forms the core of the proof in Section 4.2. The main function is `findthebraids` which, upon input of a braid word, iteratively constructs braids whose closures are $t_3, \overline{t_4}$ equivalent to the closure of the input braid using the replacement rules from Section 4.3.

```
# -*- coding: utf-8 -*-

# Define global variables
braidlist = []
minlength = 100
startlength = 20
index = 0
fileloc = "//home//ubuntu//Documents//PythFiles//v13Outputs//"
```

```
class GetOutOfLoop( Exception ): pass

# Reduction replacement rules that are consequences of the Reidemeister III
    # move, followed by a Reidemeister II move or a t_3 move.
r3rewrites = [[[1,2,1,-2], [2,1]],              [[-2,1,2,1], [1,2]],
              [[2,1,-2,-1], [-1,2]],            [[1,-2,-1,-2], [-2,-1]],
              [[2,-1,-2,-1], [-1,-2]],          [[-2,-1,-2, 1], [-1,-2]],
              [[-1,-2,-1,2], [-2,-1]],          [[-1,2,1,2], [2,1]],
              [[2,1,2,-1], [1,2]],              [[1,2,-1,-2], [-2, 1]],
              [[-2,-1,2,1], [1,-2]],            [[-1,-2,1,2], [2,-1]],
              [[1, 2, 1, 2], [2, 1, -2]],       [[2, 1, 2, 1], [-2, 1, 2]],
              [[1, 2, -1, 2], [-2, 1, -2]],     [[-2, 1, 2, -1], [2, 1, 2]],
              [[1, -2, -1, 2], [-2, -1, -2]],   [[-2, -1, 2, -1], [1, -2, 1]],
              [[-1, -2, -1, -2], [-2, -1, 2]],[[-2, -1, -2, -1], [2, -1, -2]],
              [[-1, 2, 1, -2], [2, 1, 2]],      [[2, 1, -2, 1], [-1, 2, -1]],
              [[-1, -2, 1, -2], [2, -1, 2]],    [[2, -1, -2, 1], [-2, -1, -2]],
              [[2,3,2,-3], [3,2]],              [[-3,2,3,2], [2,3]],
              [[3,2,-3,-2], [-2,3]],            [[2,-3,-2,-3], [-3,-2]],
              [[3,-2,-3,-2], [-2,-3]],          [[-3,-2,-3, 2], [-2,-3]],
              [[-2,-3,-2,3], [-3,-2]],          [[-2,3,2,3], [3,2]],
              [[3,2,3,-2], [2,3]],              [[2,3,-2,-3], [-3, 2]],
              [[-3,-2,3,2], [2,-3]],            [[-2,-3,2,3], [3,-2]],
              [[2, 3, 2, 3], [3, 2, -3]],       [[3, 2, 3, 2], [-3, 2, 3]],
              [[2, 3, -2, 3], [-3, 2, -3]],     [[-3, 2, 3, -2], [3, 2, 3]],
              [[2, -3, -2, 3], [-3, -2, -3]],   [[-3, -2, 3, -2], [2, -3, 2]],
              [[-2, -3, -2, -3], [-3, -2, 3]],[[-3, -2, -3, -2], [3, -2, -3]],
              [[-2, 3, 2, -3], [3, 2, 3]],      [[3, 2, -3, 2], [-2, 3, -2]],
              [[-2, -3, 2, -3], [3, -2, 3]],    [[3, -2, -3, 2], [-3, -2, -3]],
              [[3,4,3,-4], [4,3]],              [[-4,3,4,3], [3,4]],
              [[4,3,-4,-3], [-3,4]],            [[3,-4,-3,-4], [-4,-3]],
              [[4,-3,-4,-3], [-3,-4]],          [[-4,-3,-4, 3], [-3,-4]],
                  [[-3,-4,-3,4], [-4,-3]],          [[-3,4,3,4], [4,3]],
                  [[4,3,4,-3], [3,4]],              [[3,4,-3,-4], [-4, 3]],
                  [[-4,-3,4,3], [3,-4]],            [[-3,-4,3,4], [4,-3]],
                  [[3, 4, 3, 4], [4, 3, -4]],       [[4, 3, 4, 3], [-4, 3, 4]],
                  [[3, 4, -3, 4], [-4, 3, -4]],     [[-4, 3, 4, -3], [4, 3, 4]],
                  [[3, -4, -3, 4], [-4, -3, -4]], [[-4, -3, 4, -3], [3, -4, 3]],
                  [[-3, -4, -3, -4], [-4, -3, 4]],[[-4, -3, -4, -3], [4, -3, -4]],
                  [[-3, 4, 3, -4], [4, 3, 4]],      [[4, 3, -4, 3], [-3, 4, -3]],
                  [[-3, -4, 3, -4], [4, -3, 4]],    [[4, -3, -4, 3], [-4, -3, -4]]]

# Replacements of the form (A1)
r3t3moves = [[[1, -2, 1, -2], [-1, 2, -1, 2], [2, -1, 2, -1], [-2, 1, -2, 1]],
             [[2, -3, 2, -3], [-2, 3, -2, 3], [3, -2, 3, -2], [-3, 2, -3, 2]],
             [[3, -4, 3, -4], [-3, 4, -3, 4], [4, -3, 4, -3], [-4, 3, -4, 3]]]

# Further consequences of (A1) and commuting
t32rewrites = [[[2, -1, 2, -1, -3, 2, -3], [1, -2, 1, 3, -2, 3]],
               [[2, -1, 2, -1, 3, -2, 3], [-1, 2, -1, -3, 2, -3]],
               [[-2, 1, -2, 1, -3, 2, -3], [1, -2, 1, 3, -2, 3]],
               [[-2, 1, -2, 1, 3, -2, 3], [-1, 2, -1, -3, 2, -3]],
               [[3, -2, 3, -2, -4, 3, -4], [2, -3, 2, 4, -3, 4]],
               [[3, -2, 3, -2, 4, -3, 4], [-2, 3, -2, -4, 3, -4]],
               [[-3, 2, -3, 2, -4, 3, -4], [2, -3, 2, 4, -3, 4]],
               [[-3, 2, -3, 2, 4, -3, 4], [-2, 3, -2, -4, 3, -4]],
               [[2, -3, 2, -3, -1, 2, -1], [3, -2, 3, 1, -2, 1]],
               [[2, -3, 2, -3, 1, -2, 1], [-3, 2, -3, -1, 2, -1]],
               [[-2, 3, -2, 3, -1, 2, -1], [3, -2, 3, 1, -2, 1]],
               [[-2, 3, -2, 3, 1, -2, 1], [-3, 2, -3, -1, 2, -1]],
               [[3, -4, 3, -4, -2, 3, -2], [4, -3, 4, 2, -3, 2]],
               [[3, -4, 3, -4, 2, -3, 2], [-4, 3, -4, -2, 3, -2]],
               [[-3, 4, -3, 4, -2, 3, -2], [4, -3, 4, 2, -3, 2]],
               [[-3, 4, -3, 4, 2, -3, 2], [-4, 3, -4, -2, 3, -2]],
               [[3, -2, 3, -1, 2, -1, 2], [-3, 2, -3, -1, 2, -1]],
               [[-3, 2, -3, -1, 2, -1, 2], [3, -2, 3, 1, -2, 1]],
               [[3, -2, 3, 1, -2, 1, -2], [-3, 2, -3, -1, 2, -1]],
               [[-3, 2, -3, 1, -2, 1, -2], [3, -2, 3, 1, -2, 1]],
               [[4, -3, 4, -2, 3, -2, 3], [-4, 3, -4, -2, 3, -2]],
               [[-4, 3, -4, -2, 3, -2, 3], [4, -3, 4, 2, -3, 2]],
               [[4, -3, 4, 2, -3, 2, -3], [-4, 3, -4, -2, 3, -2]],
               [[-4, 3, -4, 2, -3, 2, -3], [4, -3, 4, 2, -3, 2]],
               [[1, -2, 1, -3, 2, -3, 2], [-1, 2, -1, -3, 2, -3]],
               [[-1, 2, -1, -3, 2, -3, 2], [1, -2, 1, 3, -2, 3]],
               [[1, -2, 1, 3, -2, 3, -2], [-1, 2, -1, -3, 2, -3]],
               [[-1, 2, -1, 3, -2, 3, -2], [1, -2, 1, 3, -2, 3]],
               [[2, -3, 2, -4, 3, -4, 3], [-2, 3, -2, -4, 3, -4]],
               [[-2, 3, -2, -4, 3, -4, 3], [2, -3, 2, 4, -3, 4]],
               [[2, -3, 2, 4, -3, 4, -3], [-2, 3, -2, -4, 3, -4]],
               [[-2, 3, -2, 4, -3, 4, -3], [2, -3, 2, 4, -3, 4]]]

# Replacements of the form (A2)
t3rewrites = [[[1, -2, 1, -2, 1], [2, -1, 2]],
               [[-1, 2, -1, 2, -1], [-2, 1, -2]],
               [[2, -1, 2, -1, 2], [1, -2, 1]],
               [[-2, 1, -2, 1, -2], [-1, 2, -1]],
               [[2, -3, 2, -3, 2], [3, -2, 3]],
               [[-2, 3, -2, 3, -2], [-3, 2, -3]],
               [[3, -2, 3, -2, 3], [2, -3, 2]],
               [[-3, 2, -3, 2, -3], [-2, 3, -2]],
               [[3, -4, 3, -4, 3], [4, -3, 4]],
               [[-3, 4, -3, 4, -3], [-4, 3, -4]],
               [[4, -3, 4, -3, 4], [3, -4, 3]],
               [[-4, 3, -4, 3, -4], [-3, 4, -3]]]
```

```
# Specialized (A4) targets
t4barr3stabs4 = [[3, -4, 3, -4], [-3, 4, -3, 4],
                 [4, -3, 4, -3], [-4, 3, -4, 3]]
t4barr3stabs1 = [[2, -1, 2, -1], [-2, 1, -2, 1],
                 [1, -2, 1, -2], [-1, 2, -1, 2]]

# Replacements of the form (A6)
t3exps = [[[1, -2, 1, 3, -2, 1], [1, -2, -1, -1, 3, -2, 1]],
          [[-1, 2, -1, -3, 2, -1], [-1, 2, 1, 1, -3, 2, -1]],
          [[4, -3, 4, 2, -3, 4], [-4, 3, -4, -4, 2, -3, 4]],
          [[-4, 3, -4, -2, 3, -4], [-4, 3, 4, 4, -2, 3, -4]],
          [[1, -2, 3, 1, -2, 1], [1, -2, 3, -1, -1, -2, 1]],
          [[-1, 2, -3, -1, 2, -1], [-1, 2, -3, 1, 1, 2, -1]],
          [[4, -3, 2, 4, -3, 4], [4, -3, 2, -4, -4, -3, 4]],
          [[-4, 3, -2, -4, 3, -4], [-4, 3, -2, 4, 4, 3, -4]],
          [[1, -2, 1, -3, -2, 1], [1, -2, -1, -1, -3, -2, 1]],
          [[-1, 2, -1, 3, 2, -1], [-1, 2, 1, 1, 3, 2, -1]],
          [[4, -3, 4, -2, -3, 4], [-4, 3, -4, -4, -2, -3, 4]],
          [[-4, 3, -4, 2, 3, -4], [-4, 3, 4, 4, 2, 3, -4]],
          [[1, -2, -3, 1, -2, 1], [1, -2, -3, -1, -1, -2, 1]],
          [[-1, 2, 3, -1, 2, -1], [-1, 2, 3, 1, 1, 2, -1]],
          [[4, -3, -2, 4, -3, 4], [4, -3, -2, -4, -4, -3, 4]],
          [[-4, 3, 2, -4, 3, -4], [-4, 3, 2, 4, 4, 3, -4]]]

# More replacements of the form (A6)
t3exps2 = [[[2, -1, 2, -3, 2], [2, -1, -2, -2, -3, 2]],
           [[2, -3, 2, -1, 2], [2, -3, -2, -2, -1, 2]],
           [[-2, 1, -2, 3, -2], [-2, 1, 2, 2, 3, -2]],
           [[-2, 3, -2, 1, -2], [-2, 3, 2, 2, 1, -2]],
           [[3, -2, 3, -4, 3], [3, -2, -3, -3, -4, 3]],
           [[3 ,  −4,  3 ,  −2,  3] ,  [3 ,  −4,  −3,  −3,  −2,  3]] ,
           [[ −3 ,  2 ,  −3,  4 ,  −3],  [−3,  2 ,  3 ,  3 ,  4 ,  −3]],
           [[ −3 ,  4 ,  −3,  2 ,  −3],  [−3,  4 ,  3 ,  3 ,  2 ,  −3]],
           [[2 ,  1 ,  2 ,  −3,  2] ,  [2 ,  1 ,  −2,  −2,  −3,  2]] ,
           [[2 ,  3 ,  2 ,  −1,  2] ,  [2 ,  3 ,  −2,  −2,  −1,  2]] ,
           [[ −2 ,  −1,  −2,  3 ,  −2],  [−2,  −1,  2 ,  2 ,  3 ,  −2]],
           [[ −2 ,  −3,  −2,  1 ,  −2],  [−2,  −3,  2 ,  2 ,  1 ,  −2]],
           [[3 ,  2 ,  3 ,  −4,  3] ,  [3 ,  2 ,  −3,  −3,  −4,  3]] ,
           [[3 ,  4 ,  3 ,  −2,  3] ,  [3 ,  4 ,  −3,  −3,  −2,  3]] ,
           [[ −3 ,  −2,  −3,  4 ,  −3],  [−3,  −2,  3 ,  3 ,  4 ,  −3]],
           [[ −3 ,  −4,  −3,  2 ,  −3],  [−3,  −4,  3 ,  3 ,  2 ,  −3]],
           [[2 ,  −1,  2 ,  3 ,  2] ,  [2 ,  −1,  −2,  −2,  3 ,  2]] ,
           [[2 ,  −3,  2 ,  1 ,  2] ,  [2 ,  −3,  −2,  −2,  1 ,  2]] ,
           [[ −2 ,  1 ,  −2,  −3,  −2],  [−2,  1 ,  2 ,  2 ,  −3,  −2]],
           [[ −2 ,  3 ,  −2,  −1,  −2],  [−2,  3 ,  2 ,  2 ,  −1,  −2]],
           [[3 ,  −2,  3 ,  4 ,  3] ,  [3 ,  −2,  −3,  −3,  4 ,  3]] ,
           [[3 ,  −4,  3 ,  2 ,  3] ,  [3 ,  −4,  −3,  −3,  2 ,  3]] ,
           [[ −3 ,  2 ,  −3,  −4,  −3],  [−3,  2 ,  3 ,  3 ,  −4,  −3]],
           [[ −3 ,  4 ,  −3,  −2,  −3],  [−3,  4 ,  3 ,  3 ,  −2,  −3]]]

# Reidemeister III moves
r3basic = [[[1 ,2 ,1] ,  [2 ,1 ,2]] ,          [[1 ,2 , −1] ,  [ −2 ,1 ,2]] ,
           [[ −1 ,2 ,1] ,  [2 ,1 , −2]] ,          [[ −1 , −2 ,1] ,  [2 , −1 , −2]] ,
           [[1 , −2 , −1] ,  [ −2 , −1 ,2]] ,       [[ −1 , −2 , −1] ,  [ −2 , −1 , −2]] ,
           [[2 ,3 ,2] ,  [3 ,2 ,3]] ,          [[2 ,3 , −2] ,  [ −3 ,2 ,3]] ,
           [[ −2 ,3 ,2] ,  [3 ,2 , −3]] ,          [[ −2 , −3 ,2] ,  [3 , −2 , −3]] ,
           [[2 , −3 , −2] ,  [ −3 , −2 ,3]] ,       [[ −2 , −3 , −2] ,  [ −3 , −2 , −3]] ,
           [[3 ,4 ,3] ,  [4 ,3 ,4]] ,          [[3 ,4 , −3] ,  [ −4 ,3 ,4]] ,
           [[ −3 ,4 ,3] ,  [4 ,3 , −4]] ,          [[ −3 , −4 ,3] ,  [4 , −3 , −4]] ,
           [[3 , −4 , −3] ,  [ −4 , −3 ,4]] ,       [[ −3 , −4 , −3] ,  [ −4 , −3 , −4]]]
```

```
def find_sublists(seq, sublist):
    """Locates all occurrences of SUBLIST within input SEQ"""
    length = len(sublist)
    ixs = []
    for index, value in enumerate(seq):
        if value == sublist[0] and seq[index:index+length] == sublist:
            ixs.append([ index, index+length])
    return ixs, len(ixs) != 0

def replace_sublist(braid, st, st1, ixs):
    """Replaces sublist ST within BRAID with sublist ST1 by using input
    indexes IXS, output from FIND_SUBLISTS"""
    newbraids = []
    for i in range(len(ixs)):
        newBraid = []
        # Check indices are correct
        if braid[ixs[i][0]:ixs[i][1]] != st:
            print("ERROR!")
            newbraids.append(braid)
        else:
            newBraid = braid[:ixs[i][0]]
            for j in range(len(st1)):
                newBraid.append(st1[j])
            for k in range(ixs[i][1], len(braid)):
                newBraid.append(braid[k])
            newbraids.append(newBraid)
    return newbraids

def reset():
    """Resets global variables at start"""
    global braidlist
    global minlength
    braidlist = []
    minlength = 100

def updatebraidlist(braid):
    """Checks if input BRAID (or a cyclic permutation) is already in the
    global variable BRAIDLIST; if not, appends BRAID to BRAIDLIST"""
    global braidlist
    global minlength
    global startlength
    if len(braid) < minlength:
        minlength = len(braid)
        print("New min length is " + str(minlength))
    tfflag = False
    newBraid = braid[:]
    if newBraid in braidlist: tfflag = True
    for i in range(len(braid)-1):
        newBraid = cycle(newBraid)
        if newBraid in braidlist: tfflag = True
    if not tfflag: braidlist.append(braid)

def trymove(braid, st, st1, trytail = False, headlen = 0):
    """Tries to locate the sublist ST within BRAID; if located, replaces
    ST with ST1 in BRAID and updates the global variable BRAIDLIST. If
    TRYTAIL = TRUE, will look to the head of BRAID and cycle missing end
    entries of ST using commutation rules."""
    newBraid = braid[:]
    newBraids, tf = lookLRwrap(newBraid, st)
    if tf:
        for newbr in newBraids:
            newBraid = newbr[:]
            ixs, tf2 = find_sublists(newBraid, st)
            newBraids2 = replace_sublist(newBraid, st, st1, ixs)
            for i in range(len(newBraids2)):
                updatebraidlist(newBraids2[i])
    elif trytail:
        newBraids, tf = lookTailwrap(newBraid, st, headlen)
        if tf:
            for newbr in newBraids:
                newBraid = newbr[:]
                ixs, tf2 = find_sublists(newBraid, st)
                newBraids2 = replace_sublist(newBraid, st, st1, ixs)
                for i in range(len(newBraids2)):
                    updatebraidlist(newBraids2[i])

def findt4barr3stabs(braid):
    """Looks for the (A4) patterns within BRAID, and if these are the
    only occurrences of +/-1 or +/-4, returns TRUE (ie, using
    replacement (A1) results in only one occurrence of +/-1 or +/-4,
    thus the braid can be destabilized."""
    newBraid = braid[:]
    for i in range(len(t4barr3stabs4)):
        ixs, tfnew = find_sublists(newBraid, t4barr3stabs4[i])
        if not tfnew:
            newBraids, tfnew = lookLRwrap(newBraid, t4barr3stabs4[i])
            if len(newBraids) != 0: newBraid = newBraids[0]
            else:
                continue
        if tfnew and ((newBraid.count(4) == 2 and newBraid.count(-4) == 0) or
                      (newBraid.count(4) == 0 and newBraid.count(-4) == 2)):
            return newBraid, True
    for i in range(len(t4barr3stabs1)):
        ixs, tfnew = find_sublists(newBraid, t4barr3stabs1[i])
        if not tfnew:
            newBraids, tfnew = lookLRwrap(newBraid, t4barr3stabs1[i])
            if len(newBraids) != 0: newBraid = newBraids[0]
            else:
                continue
        if tfnew and ((newBraid.count(1) == 2 and newBraid.count(-1) == 0) or
                      (newBraid.count(1) == 0 and newBraid.count(-1) == 2)):
            return newBraid, True
    return braid, False

def reducet3(braid):
    """Uses t_3 moves to reduce the number of crossings"""
    i = 0
    while i < len(braid)-1:
        newBraid = braid[:]
        if newBraid[i] == newBraid[i+1]:
            newBraid[i] = -newBraid[i]
            newBraid.pop(i+1)
            updatebraidlist(newBraid)
        i += 1

def reduceinv(braid):
    """Uses a Reidemeister II move (ie, inverses in the braid group) to
    reduce the number of crossings"""
    i = 0
    while i < len(braid)-1:
        newBraid = braid[:]
        if newBraid[i] + newBraid[i+1] == 0:
            newBraid.pop(i+1)
            newBraid.pop(i)
            updatebraidlist(newBraid)
        i += 1

def cycle(braid):
    """Cyclically permutes the braid by appending the final entry
    to the front"""
    newbraid = [braid[-1]]
    for i in range(len(braid)-1):
        newbraid.append(braid[i])
    return newbraid

def swap(braid):
    """Commutes adjacent entries if permitted"""
    global usedswaps
    global numswaps
    newBraid = braid[:]
    for i in range(len(braid) - 1):
        if i not in usedswaps:
            if abs(abs(braid[i]) - abs(braid[i+1])) >= 2:
                newBraid[i] = braid[i+1]
                newBraid[i+1] = braid[i]
                updatebraidlist(newBraid)
                return newBraid, True
    return braid, False

def targetSwap(braid):
    """Looks for the initial entry in the braid towards the tail to commute
    the latter occurrence to the end so that after cycling, the braid can
    be reduced."""
    lookFor = [braid[0], -braid[0]]
    ind = max(loc for loc, val in enumerate(braid) if val in lookFor)
    if ind != len(braid) - 1:
        newbraid = braid[0:ind]
        for i in range(ind+1, len(braid)):
            if abs(abs(braid[i]) - abs(braid[0])) >= 2: newbraid.append(braid[i])
            else: return braid, True
        newbraid.append(braid[ind])
        updatebraidlist(newbraid)

def lookLRwrap(braid, substring):
    """A wrapper function for lookLR. Tries to locate the core (ie, all
    but the initial and final entries) of SUBSTRING within BRAID, and if
    successful, passes to lookLR."""
    newBraid = braid[:]
    outBraids = []
    tfs = False
    ixs, tf = find_sublists(newBraid, substring[1:-1])
    if tf:
        for ind in range(len(ixs)):
            newBraid = braid[:]
            outBraid, tf = lookLR(newBraid, substring, ixs[ind])
            if tf:
                outBraids.append(outBraid)
                tfs = True
    return outBraids, tfs

def lookLR(braid, substring, ixs):
    """When the core of SUBSTRING has already been located at IXS within
    BRAID: looks for the initial and final entries of SUBSTRING before
    and later, respectively, within BRAID and when permitted commutes
    those entries so that SUBSTRING occurs consecutively within BRAID."""
    newBraid = braid[:]
```

```
if ixs[1] == len(newBraid):
    return braid, False #center of substring ends at end of braid
lefttf = False
if newBraid[ixs[0]-1] == substring[0]:
    #initial entry of substring is immediately before the core
    lefttf = True
    tempBraid = newBraid[:]
else:
    try:
        #looks for last occurrence of substring[0] before the core appears
        leftix = max(loc for loc, val in enumerate(newBraid[:ixs[0]]) if
                    val == substring[0])
        tempBraid = newBraid[0:leftix]
        try:
            #tries to commute substring[0] to immediately before the core
            for j in range(leftix + 1, ixs[0]):
                if abs(abs(newBraid[j]) - abs(newBraid[leftix])) >= 2:
                    tempBraid.append(newBraid[j])
                else: raise GetOutOfLoop
            tempBraid.append(newBraid[leftix])
            for k in range(ixs[0], len(newBraid)):
                tempBraid.append(newBraid[k])
            lefttf = True
        except GetOutOfLoop: return braid, False #commutation failed
    except: return braid, False
        #substring[0] does not appear before the core
if lefttf:
    #substring[0:-1] occurs within the braid, possibly after commuting
    if tempBraid[ixs[1]] == substring[-1]:
        #final entry of substring is immediately after the core
        return tempBraid, True
        else:
            try:
                #looks for first occurrence of substring[-1] after the core
                rightix = min(loc for loc, val in enumerate(tempBraid[ixs[1]:])
                              if val == substring[-1])
                rightix += ixs[1]
                outBraid = tempBraid[:ixs[1]]
                outBraid.append(tempBraid[rightix])
                try:
                    for j in range(ixs[1], rightix):
                        if abs(abs(tempBraid[j]) - abs(tempBraid[rightix])) >=2:
                            outBraid.append(tempBraid[j])
                        else: raise GetOutOfLoop
                    for k in range(rightix+1, len(tempBraid)):
                        outBraid.append(tempBraid[k])
                    return outBraid, True
                except GetOutOfLoop: return braid, False #commutation failed
            except: return braid, False
                #substring[-1] does not occur after the core
    return braid, False

def lookTailwrap(braid, substring, headlen):
    """A wrapper function for lookTail. Locates the first HEADLEN entries of
    SUBSTRING within BRAID, and if successful, passes to lookTail."""
    newBraid = braid[:]
    outBraids = []
    tfs = False
    ixs, tf = find_sublists(newBraid, substring[:headlen])
    if tf:
        for ind in range(len(ixs)):
            newBraid = braid[:]
            outBraid, tf = lookTail(newBraid, substring, headlen, ixs[ind])
            if tf:
                outBraids.append(outBraid)
                tfs = True
    return outBraids, tfs

def lookTail(braid, substring, headlen, ixs):
    """The first HEADLEN entries of SUBSTRING occur consecutively within
        BRAID; this function tries to commute the final entries of SUBSTRING
        toward the head so that SUBSTRING occurs consecutively within BRAID."""
    newBraid = braid[:]
    if ixs[1] == len(braid):
        return braid, False
    if newBraid[ixs[1]:ixs[1] + len(substring) - headlen] == substring[headlen:]:
        return newBraid, True
    else:
        try:
            tempBraid = newBraid[:]
            for j in range(headlen, len(substring)):
                #find first occurrence of next entry of SUBSTRING
                ix = min(loc for loc, val in enumerate(tempBraid[ixs[1]+j-headlen:])
                         if val == substring[j])
                ix += ixs[1]+j-headlen
                #if possible, commute next entry forward
                tempBraid, tf2 = swapLocs(tempBraid, j+ixs[0], ix)
                if not tf2: return braid, False
            return tempBraid, True
        except: return braid, False
    return braid, False

def swapLocs(braid, ind1, ind2):
    """Tries to commute the entry at IND2 to occur before IND1"""
    newBraid = braid[:]
    tempBraid = newBraid[:ind1]
    tempBraid.append(newBraid[ind2])
    try:
        for i in range(ind1, ind2):
            if abs(abs(newBraid[i]) - abs(newBraid[ind2])) >= 2:
                tempBraid.append(newBraid[i])
            else: raise GetOutOfLoop
        for j in range(ind2+1, len(braid)):
            tempBraid.append(newBraid[j])
        return tempBraid, True
    except GetOutOfLoop: return braid, False
        #braid[ind2] couldn't commute past something between ind2 and ind1

def t4bar(braid):
    """Uses an \overline{t_4} move; ie, (A3)"""
    flag4 = False
    flag1 = False
    newBraid = braid[:]
    if braid.count(4) == 2 and braid.count(-4) == 0:
        newBraid[newBraid.index(4)] = -4
        newBraid[newBraid.index(4)] = -4
        updatebraidlist(newBraid)
        flag4 = True
    elif braid.count(-4) == 2 and braid.count(4) == 0 and not flag4:
        newBraid[newBraid.index(-4)] = 4
        newBraid[newBraid.index(-4)] = 4
        updatebraidlist(newBraid)
        flag4 = True
    newBraid = braid[:]
    if braid.count(1) == 2 and braid.count(-1) == 0:
        newBraid[newBraid.index(1)] = -1
        newBraid[newBraid.index(1)] = -1
        updatebraidlist(newBraid)
        flag1 = True
    elif braid.count(-1) == 2 and braid.count(1) == 0 and not flag1:
        newBraid[newBraid.index(-1)] = 1
        newBraid[newBraid.index(-1)] = 1
        updatebraidlist(newBraid)
        flag1 = True

def r2move(braid):
    """Uses (A2), an edge-specific Reidemeister II move in braids"""
    newBraid = braid[:]
    if braid.count(4) == 1 and braid.count(-4) == 1:
        ix = newBraid.index(-4)
        newBraid[newBraid.index(4)] = -4
        newBraid[ix] = 4
        updatebraidlist(newBraid)
    newBraid = braid[:]
    if braid.count(1) == 1 and braid.count(-1) == 1:
        ix = newBraid.index(-1)
        newBraid[newBraid.index(1)] = -1
        newBraid[ix] = 1
        updatebraidlist(newBraid)
    if braid.count(2) == 1 and braid.count(-2) == 1:
        ix = newBraid.index(-2)
        newBraid[newBraid.index(2)] = -2
        newBraid[ix] = 2
        updatebraidlist(newBraid)
    if braid.count(3) == 1 and braid.count(-3) == 1:
        ix = newBraid.index(-3)
        newBraid[newBraid.index(3)] = -3
        newBraid[ix] = 3
        updatebraidlist(newBraid)

def checkCnctSum(braid):
    """Checks if the braid is a connect sum"""
    newBraid = braid[:]
    if newBraid.count(2) == 1 and newBraid.count(-2) == 0: return True
    elif newBraid.count(2) == 0 and newBraid.count(-2) == 1: return True
    newBraid = braid[:]
    if newBraid.count(3) == 1 and newBraid.count(-3) == 0: return True
    elif newBraid.count(3) == 0 and newBraid.count(-3) == 1: return True
    return False

def checkAllGens(braid):
    """Checks that all generators occur within BRAID"""
    newBraid = braid[:]
    for i in range(1, 5):
        if newBraid.count(i) == 0 and newBraid.count(-i) == 0: return True
    return False

def destabilize(braid):
    """Checks if BRAID can be destabilized"""
    flag = False
    newBraid = braid[:]
    if newBraid.count(4) == 1 and newBraid.count(-4) == 0:
        newBraid.pop(newBraid.index(4))
        flag = True
    elif newBraid.count(4) == 0 and newBraid.count(-4) == 1:
        newBraid.pop(newBraid.index(-4))
```

```
            flag = True
        if newBraid.count(1) == 1 and newBraid.count(-1) == 0:
            newBraid.pop(newBraid.index(1))
            flag = True
        elif newBraid.count(1) == 0 and newBraid.count(-1) == 1:
            newBraid.pop(newBraid.index(-1))
            flag = True
        return newBraid, flag

def writeouts(fn, tf):
    """Writes results to file"""
    global braidlist
    global index
    global fileloc
    global shortlist
    with open(fileloc + 'debug_braidlist_' + fn + '.txt', 'w') as f:
        f.write(''.join(str(braidlist)))
    with open(fileloc + 'debug_shortlist_' + fn + '.txt', 'w') as f:
        f.write(''.join(str(shortlist)))

def tryallmoves(braid):
    """Tries all moves on BRAID"""
    br, tf = findt4barr3stabs(braid)
    if tf: return br, True
    br, tf = destabilize(braid)
    if tf: return br, True
    tf = checkCnctSum(braid)
    if tf: return br, True
    tf = checkAllGens(braid)
    if tf: return br, True
    t4bar(braid)
    r2move(braid)
    reducet3(braid)
    reduceinv(braid)
    targetSwap(braid)

    for i in range(len(r3rewrites)):
        trymove(braid, r3rewrites[i][0], r3rewrites[i][1])
    for i in range(len(t3rewrites)):
        trymove(braid, t3rewrites[i][0], t3rewrites[i][1])
    for i in range(len(t32rewrites)):
        trymove(braid, t32rewrites[i][0], t32rewrites[i][1])
    for i in range(len(r3basic)):
        trymove(braid, r3basic[i][0], r3basic[i][1])
        trymove(braid, r3basic[i][1], r3basic[i][0])
    for i in range(len(r3t3moves)):
        for j in range(len(r3t3moves[0])):
            for k in range(len(r3t3moves[0])):
                if k != j:
                    trymove(braid, r3t3moves[i][j], r3t3moves[i][k])
    for i in range(len(t3exps)):
        trymove(braid, t3exps[i][0], t3exps[i][1], True, 3)
    for i in range(len(t3exps2)):
        trymove(braid, t3exps2[i][0], t3exps2[i][1], True, 2)
    return braid, False

def findbraids(length):
    """Finds all braids of a given LENGTH within the braidlist"""
    global braidlist
    rets = []
    for braid in braidlist:
        if len(braid) == length: rets.append(braid)
    return rets

def rewritethisbraid(braid):
    """A wrapper function to try all rewrites on the braid"""
    global startlength
    curbraid = braid[:]
    br, tf = findt4barr3stabs(curbraid)
    if tf: return [br], True
    br, tf = destabilize(curbraid)
    if tf: return [br], True
    cycbraid = curbraid[:]
    for i in range(len(curbraid)):
        br, tf = tryallmoves(cycbraid)
        if tf: return [br], True
        if minlength < startlength: return findbraids(minlength), True
        cycbraid = cycle(cycbraid)
    return [br], False

def findthebraids(braid):
    """The main function"""
    global braidlist
    global minlength
    global index
    global timeout
    global startTime
    global startlength
    minlength = len(braid)
    startlength = len(braid)
    reset()
    index = 0
    updatebraidlist(braid)
    while index < len(braidlist):
        brlst, tf = rewritethisbraid(braidlist[index])
        if tf: return brlst, tf
        index += 1

    print("Ran out of braids...")
    return findbraids(minlength), False
```

A.5 B_6 enumeration script

This Python program is written to be read into SnapPy, thus defining the function `buildTheBraids`. This function is then run for each of the 11 combinations in Chapter 5.1, with input `n`$= 8$ and `count`$= 4$ when enumerating 12-crossing 6-strand braids. For example, one call takes the form `buildTheBraids([1,-2,1,3], 8, 4)` to enumerate all 6-strand braids of length 12 beginning with $\sigma_1\sigma_2^{-1}\sigma_1\sigma_3$, modulo immediate reductions due to inverses within the braid group and the t_3 move.

```
generators = [1, -1, 2, -2, 3, -3, 4, -4, 5, -5]

outfile = "//home//ubuntu//Documents//PythFiles//B6enum//1n21n3"
simpfilect = 0
simpfileind = 0
origfilect = 0
origfileind = 0

def init():
    global simpfilect
    global simpfileind
    global origfilect
    global origfileind

    simpfilect = 0
    simpfileind = 0
    origfilect = 0
    origfileind = 0

def writebrd(brd, count, flag = 0):
    global simpfilect
    global simpfileind
    global origfilect
    global origfileind

    if flag == 1:
        simpfileind = simpfileind + 1
        if simpfileind > 100000:
            simpfileind = 0
            simpfilect = simpfilect + 1
        with open(outfile + "len_" + str(count) + "_simpBrd_" + str(simpfilect)
                + ".txt", 'a+') as f:
            f.write(str(brd) + "\n")
        f.close()
    else:
        origfileind = origfileind + 1
        if origfileind > 100000:
            origfileind = 0
            origfilect = origfilect + 1
        with open(outfile + "len_" + str(count) + "_origBrd_" + str(origfilect)
                + ".txt", 'a+') as f:
            f.write(str(brd) + "\n")
        f.close()
```

```
def countGens(brd):
    flag = True
    for i in range(1, 6):
        if brd.count(i) + brd.count(-i) == 0:
            flag = False
    return flag

def reduceGens(ind):
    global generators

    reducedgenerators = generators[:]
    reducedgenerators.pop(reducedgenerators.index(ind))
    reducedgenerators.pop(reducedgenerators.index(-1*ind))
    return reducedgenerators

def buildTheBraids(braid, n, count = 0):
    global generators
    global outfile
    global simpfilect
    global simpfileind
    global origfilect
    global origfileind

    if n >= 1:
        if braid == []:
            print("initializing")
            init()

        else:
            if n == 2:
                flag = countGens(braid)
                if flag:
                    for gen in reduceGens(braid[-1]):
                        newBraid = braid[:]
                        newBraid.append(gen)
                        buildTheBraids(newBraid, n-1, count + 1)
            elif n == 1:
                flag = True
                for i in range(1, 6):
                    if braid.count(i) + braid.count(-i) == 1:
                        flag = False
                if flag:
                    for gen in reduceGens(braid[-1]):
                        newBraid = braid[:]
                        newBraid.append(gen)
                        buildTheBraids(newBraid, n-1, count + 1)
            else:
                for gen in reduceGens(braid[-1]):
                    newBraid = braid[:]
                    newBraid.append(gen)
                    buildTheBraids(newBraid, n-1, count + 1)
    else:
        newBraid = braid[:]
        flag=countGens(newBraid)
        if flag:
            L = Link(braid_closure = newBraid)
            if L.simplify():
                try:
                    braidInd = 0
                    brd = L.braid_word()
                    for i in range(len(brd)):
                        if abs(brd[i]) > braidInd:
                            braidInd = abs(brd[i])

                    if (len(L.crossings) > 11) or (braidInd > 3):
                        writebrd(brd, count, 1)
                except:
                    writebrd(newBraid, count)
            else:
                writebrd(newBraid, count)
```

Appendix B

Conjugacy classes of Q_5

Herein are the representatives for the 102 conjugacy classes of Q_5 constructed using MAGMA.

1. Identity
2. `[1, -2, 3, -4, 3, -2, 1, -2, 3, -4, 3, -2, 1, -2, 3, -4, 3, -2]`
3. `[4, -2, -3, 4, -3, 2]`
4. `[1, 2, 1, 3, -4, 3, -2, 1, 3, 2, 4, -3, 2, 1, 4, 3, -2, 3, -4, 3]`
5. `[1, 2, 1, 4, -3, -2, 1, -4, 3, -2, 1, 3, -4, 3, -2, 1, 3, -2, -4, -3]`
6. `[2, 1, 3, -4, 3, -2, 1, 3, 2, 4, -3, 2, 1, 4, 3, -2, 3, -4, 3]`
7. `[2, 1, 3, -2, 1, -4, 3, -2, 1, 3, 4, -3, 2, -1, -3, 2, 4, -3, 4]`
8. `[2, 3, 2, -1, -2, 3, -4, 3, -2, 1, 3, -2, 1, 3, -4, 3, -2, 3, -4]`
9. `[-1]`
10. `[1]`
11. `[2, 1, 4, -3, -2, 1, -4, 3, -2, 1, 3, -4, 3, -2, 1, 3, -2, -4, -3]`
12. `[2, 3, -1, -4, 3, -2, 3, -1, -2, -4, 3, 2, -1, 2, -3, -4, -3, 2]`

13. [1, 3]

14. [2, 1, 3, -2, -4, 3, -2, 1, -2, -4, 3, -2, 1, -2, -4, 3, 2, -4]

15. [4, -3, 2, 4, -3, 2, -1, 2, -3, 2, 4, -3, 4]

16. [1, 2, 1, -3, 2, 1, -3, 2]

17. [4, 3, 2, -1, 2, -3, 4, -3, 2, -1, 2, 3, 4]

18. [2, 1, 3, 4, -2, 3, 4, -2, 1, 3, -2, 1]

19. [4, -1, 2, -1, -3, -2, 1, -4, 3, -2, -1, -4, 3]

20. [2, -1, -3, -2, -4, 3, -2, -1, -4, -3, 2, -3, 4, -3]

21. [3, -1, 2, -1, -4, -3, 2, -3, 4, -3, -2, 1, -2, -3, 4]

22. [1, 2, -3, 4, -3, 2, 1, -3, 2, -3, 4, -3, 2, -3]

23. [2, -1, 2, -3, 2, -4, -3, 2, -1, 2, -3, 2, -4, -3]

24. [1, 2, -3, 2, -4, -3, 2, -1, 2, -3, 2, -4, -3, 2]

25. [2, -3, 2, -4, -3, 2, -1, 2, -3, 2, -4, -3, 2]

26. [2, 1, 3, 4, -2, 3, 4, -2, 1, 3, 2, -3, 4, -3]

27. [2, -3, 4, -3, 2, 1, -3, 2, -3, 4, -3, 2, -3]

28. [-2, 3, -4, 3, -2, 1, -2, 3, -4, 3, -2, 1, -2, 3, -4, 3, -2]

29. [2, 3, -4, 3, 2, -1, 2, 3, -4, 3, 2, -1, 2, 3, -4, 3, 2]

30. [1, 2, 3, -4, 3, -2, 1, 3, 4, -2, 3, 4, -2, 1, 3]

31. [-1, 2, -3, 2, 4, -1, 2, -3, 4, -2, 3, -1, -2, 3, -1]

32. [2, -1, 2, -3, 4, -3, 2, -1, 2, -3, 4, -3, -2, 1, -2, -3, 4]

33. [2, 4, -3, -2, -1, -4, 3, -2, 3, -1, -4, 3, -2]

34. [2, 1, 3, 4, -2, 3, 4, -2, 1, 3, 2, -3, 4]

35. [3, -1, -4, 3, -4]

36. [2, 1, 3, -4, 3, -2, 1, 3, 4, -2, 3, 4, -2, 1, 3, -2]

37. [2, -3, 2, -1, 2, 3, -4, -3, 2, -1, -3, 2, 4, -1, -3, -2]

38. [2, 1, 3, 2, -4, 3, 2, 1, -4, 3, 2]

39. [2, -1, 2, 3, -4, 3, -2, 1, -2, -3, 4, -3, -2, 1, -2]

40. [-2, -1, -3, -2, -4, 3, -2, -1, -4, -3, -2]

41. [1, 3, -4, 3, -4]

42. [2, 1, 3, -2, 1, -4, 3, -2, 3, 4, -3, -2, 1, -2, 3, -2]

43. [2, 3, -1, -2, 3, -4, 3, -2, 1, 3, -2, 1, -3, 4, -3, -2]

44. [2, 1, 3, 2, 4, -3, 2, 1, 4, 3, 2]

45. [2, -1, 2, 3, -4, 3, 2, -1, 2, -3, 4, -3, -2, 1, -2]

46. [-2, -1, -3, 4, -2, -3, 4, -2, -1, -3, -2]

47. [1, 4, -2, 3, -2, 1, -2, 3, 4, -2, -3, -2, 1, -2]

48. [2, 4, -3, 2, -1, 2, -3, 2, 4, -3, 2, 4, -1, 2, 3, -4]

49. [3, -4, -3, -2, 1, -2, -3, -4]

50. [2, -3, 2, -3, -4, -3, 2, -1, 2, -3, -4]

51. [2, -1, 2, -3, 2, 1, 4, 3, -2, 1, 3, 2, -4]

52. [3, 4, -2, 3, -1, -2, 3, -1, -2, -4, 3, 2, -1, 2]

53. [3, -4, -3, 2, -1, 2, 3, 4]

54. [2, 1, -3, -2, 1, -2, -3, 4, -3, 2, 1, -3, 2, -4]

55. [4, -2, -1, -3, 2, -1, -3, -2, -4, 3, -2, 1, -2]

56. [1, 2, -3, 2]

57. [4, 3, -2, 3, 4, -1, -2, 3, -2, 1, -2, -3, 4, -3, -2, 1]

58. [2, 3, -4, 3, -2, 1, 3, 4, -2, 1, -3, 4, -2, 1, -3, 4]

59. [-1, -2, 3, 4, -2, 3, -2, 1, -2, 3, 4]

60. [2, 3, -1, -2, -4, 3, -2, 1, -4, 3, -2, 1, 3, -2, -4, 3, -4]

61. [-3, -4, -3, -2, 1, -2, 3, 4]

62. [-1, -3, 2, -1, -3, 2, 4, -1, -3, 2, -4]

63. [1, -3, 2, 1, -4, -3, 2, 1, -3, 2, 4]

64. [2, -1, 2, -3, 4, -3, 2, -1, 2, 3, -4]

65. [3, 4, -2, 3, -1, -2, 3, -1, -2, -4, 3]

66. [1, -3, 2, 1, 4, -3, 2, 1, -3, 2, -4, -3]

67. [4, -3, 2, -1, 2, 3, -4, 3, 2, -1, 2]

68. [-2, -1, -4, 3, -2, 3, 4, -1, -2, 3]

69. [2, -1, -2, -3, -2, -1, -4, 3, -2, -1]

70. [2, 1, 3, 4, -2, -3, 4, -3, -2, 1, -2, 3, 4, -2, 3, -1]

71. [1, 2, 1, -3, 2, 1, 4, -3, 4, -2, 1, -2, 3, -2, -1, -4]

72. [2, -1, 2, -3, 4, -3, 2, -1, 2, -3, 4, -3, -2, -1]

73. [-2, 3, -1, -4, 3, -2, 3, -4, 3]

74. [3, -2, 1, 3, 4, -3, 2, 1]

75. [1, 3, -4, 3, 2, 1]

76. [2, -1, 2, -3, 2, -1, 2, -4, -3, 2, -1, 2]

77. [2, 1, 3, -2, -4, 3, -2, 1, -2, -4, 3, -2, 1, -4, 3, 2]

78. [2, -1, -3, 2, -1, -3, -4, -3, 2, -1, -3, 2, -1, -4, -3, 2]

79. [2, 1, 3, 2, 4, -1, -3, 2, 1, 4, 3, 2, -4, -3]

80. [2, -1, 2, 3, -4, 3, 2, -1, 2, 3, -4, 3, -2, 1, -2, -3, -4]

81. [2, -1, 2, -3, -2, 1, -2, -4, -3, -2, 1, -2]

82. [3, 4, 3, -1]

83. [1, 2, 1, 3, -2, -4, 3, -2, 1, -2, -4, 3, -2, 1, -4, 3, 2]

84. [1, 2, -1, -3, 2, -1, -3, -4, -3, 2, -1, -3, 2, -1, -4, -3, 2]

85. [2, 1, -3, 4, -3, 2, 1, -3, 4, -3, 2, -3, -4]

86. [2, -1, 2, 3, -4, 3, -2, 1, -2, 3, -4, 3, -2, 1, -2, -3, -4]

87. [1, 3, 4, 3]

88. [4, -3, 2, 1, 4, 3, 2]

89. [1, 2, 3, -1, -2, -4, 3, 2, -1, 2, -3, 4, -3, -2, 1, -2]

90. [1, -2, 1, 3, -2, 3, -1, -4, -3, 2, -1, -3, 4, -2]

91. [3, -1, -2, 3, -4, 3, 2, -1, 2, -3, 2, 1, -4]

92. [1, 2, 3, 4, -1, -3, 2, -3]

93. [2, 4, -1, 2, -3, 4, -2, 3, -1, -2, 3, -1, -2]

94. [2, -1, 2, -3, 2, -1, 2, -4, 3, -2, 1, -2, -4, 3]

95. [3, -4, 3, -2, 1, -2, -4, -3]

96. [3, 4, -2, 1, -2, -3, 4, -3]

97. [2, 4, 3, -2, 1, -2, 3, -4, 3, -2, 1, 3, 2, -4]

98. [2, 1, 3, -2, 1, 3, 2, -4, 3, 2, -1, 2, -4]

99. [4, -3, 2, -1, -3, 2, 4, -1, -3, -2]

100. [1, -4, -3, 2, -1, -3, 2, 4, -1, -3, 4]

101. [1, 3, 2, 1, 4, -3, 2, 1, 4, -3, 2, -3, 4]

102. [1, 3, -2, 1, 3, -4, 3, -2, 1, 3, -2, -4, 3, -2, 1, -4]

Bibliography

[Ale23] James Alexander, *A lemma on a system of knotted curves*, Proceedings of the National Academy of Sciences of the United States of America **9** (1923), 93–95.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3–4, 235–265, Computational algebra and number theory (London, 1993). MR MR1484478

[CDGW] Marc Culler, Nathan M. Dunfield, Matthias Goerner, and Jeffrey R. Weeks, *SnapPy, a computer program for studying the geometry and topology of* 3-*manifolds*, http://snappy.computop.org.

[Che00] Qi Chen, *The 3-move conjecture for 5-braids*, Knots in Hellas '98 (2000), 36–47.

[Cox57] H. S. M. Coxeter, *Factor groups of the braid group*, Proc. Fourth Canadian Math. Congress (1957), 95–122.

[CT36] H.S.M. Coxeter and J.A. Todd, *A practical method for enumerating cosets of a finite abstract group*, Proceedings of the Eidenburgh Mathematical Society **5** (1936), 26–34.

[DP02] Mieczyslaw K. Dabkowski and Jozef H. Przytycki, *Burnside obstructions to the Montesinos-Nakanishi 3-move conjecture*, Geometry and Topology **6** (2002), 355–360.

[DS11] S. Duzhin and M. Shkolnikov, *Bipartite knots*, arxiv (2011), 1–8.

[GAP17] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.8.7*, 2017.

[Kir] R. Kirby, *Problems in low-dimensional topology*, https://math.berkeley.edu/~kirby/.

[Nak90] Yasutaka Nakanishi, *On Fox's congruence classes of knots*, Osaka Journal of Mathematics **27** (1990), 207–215.

[Prz88a] Jozef Przytycki, *Plans' theorem for links: an application of t_k moves*, Canad. Math. Bull. **31** (1988), 325–327.

[Prz88b] Jozef H. Przytycki, *t_k moves on links*, Contemporary Math **78** (1988), 615–656.

[Prz90] ______, *The t_3, $\overline{t_4}$ moves conjecture for oriented links with matched diagrams*, Mathematical Proceedings of the Cambridge Philosophical Society **108** (1990), 55–61.

[Prz93] ______, *Elementary conjectures in classical knot theory*, Quantum Topology (1993), 292–320.

[Rol03] Dale Rolfsen, *Knots and links*, American Mathematical Society, 2003.