An Implementation of the Solution to the Conjugacy Problem on Thompson's Group V

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Rachel Naleczy

Annandale-on-Hudson, New York
May, 2018

Abstract

We describe an implementation of the solution to the conjugacy problem in Thompson's group V as presented by James Belk and Francesco Matucci in 2013. Thompson's group V is an infinite finitely presented group whose elements are complete binary prefix replacement maps. From these we can construct closed abstract strand diagrams, which are certain directed graphs with a rotation system and an associated cohomology class. The algorithm checks for conjugacy by constructing and comparing these graphs together with their cohomology classes. We provide a complete outline of our solution algorithm, as well as a description of the data structures which store closed abstract strand diagrams and contain methods to simplify and compare them. The final conjugacy checking program runs in $\mathcal{O}(n^3)$ time.

Contents

Dedication

This project is dedicated to all of my past, present, and future teachers in mathematics and computer science. In particular,

Randy Nellist, Laura Bobbin, Russ Byer, Tammy Carter, Christine "Lenny" Lenhard, Christy Giancursio, Jennifer "Vibbs" Vibber, Lauren Rose, Amir Barghi, John Cullinan, Maria Belk, Steve Simon, Keith O'Hara, Rebecca Thomas, and Sven Anderson;

My wonderful advisors Jim Belk and Robert McGrail; and

Neil Chodorow, who I credit with unearthing my passion for mathematics for the first time when I was 6 years old. Thank you for showing me even then how much fun math can be. I did know at the time that I would become a math major 12 years later but I am grateful that I did.

Acknowledgments

For playing an invaluable role in the production of this project, I would like to emphatically thank and acknowledge the following.

My family, for loving me, supporting my education, and indulging my crazy whims and passions for as long as I can remember (and probably before then as well);

All of my friends and all of the Bard community, for telling me it would get done every time I was convinced it never would, for helping me debug my code when I could not stare at it any longer, for reminding me to have fun every once in a while, and for your continuous love and support (shout-out to the recipients of my daily-senior-project-update snapchats, especially those who responded every day);

Bard '13 graduate Nabil Hossain, for providing a model for my work, inspiration for me to push past the hard parts, and precedence for the success of my project;

My professors at Bard College, for helping me build such a strong foundation in the fields of mathematics and computer science, for opening up a world of opportunity for me at Bard and beyond, and to Keith O'Hara in particular, whose Object Oriented Programming class convinced me to add computer science as second major, and Stefan Méndez-Diez, for ensuring that my project is devoid of any numerical error whatsoever;

Coffee shops everywhere, for caffeinating me and providing me with vibrant and pleasant places to work;

My advisors, Bard Professors Jim Belk and Robert McGrail, for meeting with me every week (sometimes multiple times), sharing your expertise about Thompson's groups and algorithmic group theory, being a fantastic sounding board for my ideas, providing direction when I got lost, reading my write-up over and over again, and making my experience writing a senior thesis at Bard enjoyable, rewarding, and absolutely unforgettable.

Introduction

In 1965, Richard Thompson introduced the groups F, T, and V by way of unpublished, handwritten notes. Now referred to as Thompson's groups, they are all infinite, finitely presented groups concerned with dyadic subdivisions and homeomorphisms on different geometric spaces. In particular, V consists of all homeomorphisms on the Cantor set determined by complete binary prefix replacement maps. For a comprehensive overview of Thompson's groups, the reader is referred to Canon, Floyd, and Parry [3]

Mathematician Max Dehn introduced the conjugacy problem in 1911 as one of three fundamental algorithmic decision problems in the study of infinite groups [5]. The conjugacy problem is the problem of determining whether any two given elements of a group are conjugate. Although it is not solvable in general [11], solutions to the conjugacy problem do exist within certain groups.

There are three known solutions to the conjugacy problem in V, the first of which was proved by Higman in 1974 [7]. Salazar-Diaz gave a second solution in 2006 [14]. We will focus on the solution introduced by Belk and Matucci [2] in 2013. Their solution includes the use of certain directed graphs called **closed abstract strand diagrams**, each of which has a rotation system and an associated cohomology class. These strand diagrams can be reduced using certain reduction moves and compared in order to decide whether two elements are conjugate. According

to their solution, two elements of V are conjugate if and only if their reduced closed abstract strand diagrams are **equivalent**. This means that they are isomorphic in a way that preserves the associated cohomology class.

In this project, we make this solution precise and present an implementation of this solution as an application. We also show that our algorithm runs in $\mathcal{O}(n^3)$ time where n is the sum of the length of the two elements for which we are checking conjugacy.

Given two elements of V, our application constructs the corresponding tree diagrams and converts them into abstract strand diagrams. We implement data structures to store, compose, and reduce tree diagrams and abstract strand diagrams, and present algorithms with analysis of runtime for all of the relevant operations. We then close each abstract strand diagram and include a data structure to store and reduce closed abstract strand diagrams.

To determine whether these closed abstract strand diagrams are equivalent, our implementation includes an isomorphism checker and a cohomology checker; once two strand diagrams have been found to be isomorphic, we check whether their associated cohomology classes are the same. It can happen that the number of self-isomorphisms grows linearly with the size of the strand diagram, so we may need to construct as many as n isomorphisms, but we believe this to be exceptionally rare.

We use the computation of matrix rank, which intially takes $\mathcal{O}(n^\omega)$ time for matrix size n and matrix multiplication coefficient ω. We also require, however, as many as n rank computations of a slightly modified version of the original matrix, which runs in $\mathcal{O}(n^2)$ time [6]; this is the source of our cubic worst case. The upper bound on our average case, however, is somewhat quicker at $\mathcal{O}(n^{2.495})$ [15]. We conjecture that the true average case for pairs of conjugate elements is $\mathcal{O}(n^\omega)$, since most often the first isomorphism found between their closed abstract strand diagrams will preserve the cohomology class and there will be no need to construct any further isomorphisms. For pairs of elements that are not conjugate, we believe that we will be able to determine that they are not conjugate once their closed abstract strand diagrams are reduced and separated into connected components; this can be done in $\mathcal{O}(n \log n)$ time.

In 2013, Hossain, McGrail, Belk, and Matucci proved the existence of a linear time algorithm to solve the conjugacy problem on Thompson's group F and released a Java implementation of their solution [8]. As far as we know, ours is the first implementation of the solution to the conjugacy problem in V using strand diagrams. We release a Java implementation of our solution algorithm as a web application and an executable JAR file. We also release our source code in the following GitHub repository:https://github.com/rnales/ConjugacyV. We hope that researchers studying Thompson's groups will use and improve upon our results in further study.

Chapter 1 provides the relevant background about conjugacy and Thompson's group V, as well as definitions which will be used throughout the paper. Chapter 2 outlines our implementation of data types to store elements of V as tree diagrams and strand diagrams, as well as algorithms for all of the necessary operations on these objects. Chapter 3 provides a detailed explanation of Belk and Mattuci's solution to the conjugacy problem [2], including descriptions of closed abstract strand diagrams and an example of determining whether two elements of V are conjugate using this method from start to finish. Chapter 4 describes our implementation of the remainder of the solution to the conjugacy problem as described in Chapter 3, as well as analysis of runtime. Finally, Chapter 5 concludes the project and draws attention to open questions and future work.

1
Background

This chapter is an exposition of what is known about conjugacy and Thompson's group V. We begin be giving an overview of decision problems on finitely presented groups, including the conjugacy problem, in Section 1.1. Then we will describe prefix replacement maps, which are the elements of V, in Section 1.2. Section 1.3 introduces Thompson's group V as well as tree diagrams, including how to reduce and compose them. To conclude we discuss converting elements of V to abstract strand diagrams in Section 1.4.

1.1 The Conjugacy Problem

Definition 1.1.1. In any group G with a generating set S, a **word** is a finite sequence of elements of S and their inverses. For any word w, if there are no instances of adjacent pairs xx^{-1} for all $x \in S$ then we say that w is a **reduced word**.

Note that the product of w is an element of G, and every element of G can be represented by a word in S. Additionally, a single element of a group G can be represented by multiple words.

Example 1.1.2. Consider the dihedral group D_3 with identity r_0 and generating set $S = \{r_1, s_1\}$. Then $r_1 s_1$ is a word in S shown in Figure 1.1.1, and $s_1 r_1^{-1}$ is also a word in S shown in Figure 1.1.2. Note that both words represent the same element in D_3.

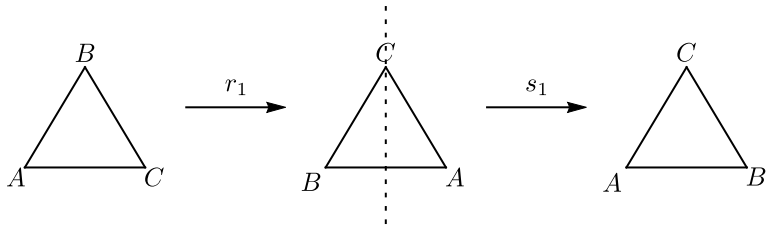


Figure 1.1.1. The element in D_3 (far right) formed from the product of the word $r_1 s_1$

Definition 1.1.3. A **decision problem** is a problem which asks for a "yes-or-no" answer to a specific question.

Max Dehn was a German-born American mathematician who is best known for his work in geometry, topology, and geometric group theory. In particular, he identified three fundamental decision problems for finitely presented groups [5]. Dehn posed these problems in 1911, and they are often referred to today as Dehn's Decision Problems.

Definition 1.1.4. Given a finitely presented group G with generating set S, the **word problem** is the descision problem of determining whether two words over S reperesent the same element in G.

Definition 1.1.5. Given a finitely presented group G with generating set S, the **conjugacy problem** is the decision problem of determing whether two words over S represent conjugate elements in G.

Definition 1.1.6. The **isomorphism problem** is the decision problem of determining whether two given finite group presentations represent isomorphic groups.

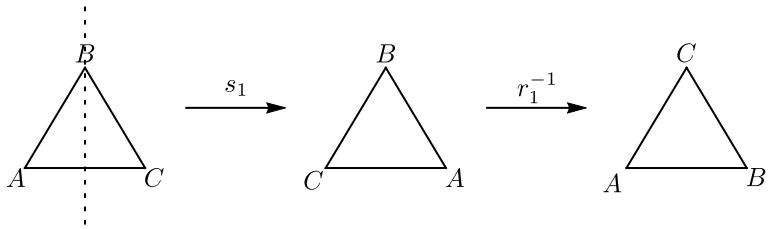


Figure 1.1.2. The element in D_3 (far right) formed from the product of the word $s_1 r_1^{-1}$

It is known that these problems do not have general solutions, but rather can have solutions on specific groups [11]. Note that we ask the word and conjugacy problems about single groups, whereas we ask the isomorphism problem about (at least) two groups at once. Dehn originally posed the isomorphism problem about all groups; it has since been proven that not all finite group presentations represent isomorphic groups. Today we instead ask the isomorphism problem about classes of finitely presented groups [1, 12].

Definition 1.1.7. If a decision problem has a solution on a group G, we say that the problem is **decidable** on G.

In 1974, Higman proved that the conjugacy problem is decidable on Thompson's group V and provided the first known solution [7]. A second solution was given by Salazar-Diaz in 2006 [14]. Belk and Matucci provided a solution in 2013 using strand diagrams, which will be the primary subject of this text [2].

We will now discuss the conjugacy problem in detail.

Definition 1.1.8. Given a group G and elements $g, h \in G$, if there exists an element $k \in G$ such that $h = k^{-1}gk$ then we say that g and h are **conjugate**. We refer to k as the **conjugator** from g to h.

Note that conjugacy is an equivalence relation.

Example 1.1.9. Consider the symmetric group S_3 and the elements $(12), (23) \in S_3$. Observe that

$$(23) = (123)(12)(132).$$

Since $(132)^{-1} = (123)$, we conclude that (12) and (23) are conjugate in S_3.

Example 1.1.10. Consider $(12), (123) \in S_3$. We construct the following table, in which the left column shows every element $s \in S_3$ and the right column shows (12) conjugated by s:

s	(1)	(12)	(13)	(23)	(123)	(132)
$s^{-1}(12)s$	(12)	(12)	(23)	(13)	(13)	(23)

Figure 1.2.1. Seven iterations of the Cantor set

Observe that there are no elements in S_3 which, when used to conjugate (12), result in (123). We conclude that (12) and (123) are not conjuate in S_3.

Definition 1.1.11. A **conjugacy class** is an equivalence class of elements of G under the conjugacy relation.

Note that each element $g \in G$ belongs to exactly one conjugacy class.

Example 1.1.12. Recall Example 1.1.10. The bottom row of the table shows every element in S_3 which is conjugate to (12). Thus the set $\{(12), (23), (13)\}$ forms a conjugacy class of S_3.

Note that determining whether two elements of a given group are conjugate is equivalent to determining whether those two elements belong to the same conjugacy class.

We will now discuss the Cantor set as motivation for defining elements of V.

1.2 The Cantor Set and Binary Prefix Codes

Consider the closed line segment $[0, 1]$. Now remove the middle third, i.e. the open interval $(\frac{1}{3}, \frac{2}{3})$, leaving the two line segments $[0, \frac{1}{3}]$ and $[\frac{2}{3}, 1]$. Imagine iteratively removing the open middle third from all of the remaining line segments. The result is a fractal in one dimension which we call the **Cantor set**, shown in Figure 1.2.1.

We can represent these segments as decimals in base 3, considering all decimals along the line segment $[0, 1]$. When we remove the first middle third, we are removing all decimals whose first digit *must* be 1 (since .1 is equivalent to $.0\overline{2}$), leaving $[0, .1]$ and $[.2, 1]$. In the next iteration, we remove decimals whose second digit *must* be 1 (since $.01 = .00\overline{2}$), leaving $[0, .01]$, $[.02, .1]$, $[.2, .21]$, and $[.22, 1]$. If we were to consider infinite iterations of this, we would remove all decimals containing the digit one and be left with only decimals containing 0's and 2's. Thus, another

way to express the Cantor set the set of all infinite sequences of only 0's and 2's, i.e. $\{0,2\}^\infty$. There is a natural bijection between this set and the set $\{0,1\}^\infty$, so we often refer to the Cantor set as the set of all infinite binary sequences.

We will now introduce some notation and terminology which we will use throughout our discussion of binary sequences.

Definition 1.2.1. If α and β are binary sequences, we write $\alpha < \beta$ if α is a prefix of β.

Definition 1.2.2. A set of binary sequences $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ is a **complete binary prefix code** if for every $\beta \in \{0,1\}^\infty$ there exists exactly one $i \in \{0,1,\ldots,n-1\}$ such that $\alpha_i < \beta$. We say that a complete binary prefix code is **ordered** if α_{i+1} is greater than α_i for all $0 \leq i < n$.

Example 1.2.3. The set $\{000, 001, 01, 10, 11\}$ is an ordered complete binary prefix code. The set $\{00, 001, 01\}$ is not a complete binary prefix code since there is no sequence α in this set such that $\alpha < 1011000....$

We now introduce prefix prelacement maps. Thompson's group V, which will be the subject of Section 1.3, is the group of all such functions.

Definition 1.2.4. Let $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ and $\{\beta_0, \beta_1, \ldots, \beta_{n-1}\}$ be ordered complete binary prefix codes and let ϕ be a bijective function between them. We define $f : \{0,1\}^\infty \to \{0,1\}^\infty$ such that for $\omega \in \{0,1\}^\infty$

$$f(\omega) = \begin{cases} \phi(\alpha_0)\psi & \text{if } \omega = \alpha_0\psi \\ \phi(\alpha_1)\psi & \text{if } \omega = \alpha_1\psi \\ \vdots & \vdots \\ \phi(\alpha_{n-1})\psi & \text{if } \omega = \alpha_{n-1}\psi \end{cases}$$

We call f a **prefix replacement map**. Additionally, we call the ordered list $[\alpha_0, \alpha_1, \ldots, \alpha_{n-1}]$ the **domain code** and we call $[\beta_0, \beta_1, \ldots, \beta_{n-1}]$ the **range code**.

We begin our indices at 0 rather than 1 for ease of implementation, to be discussed in Chapter 2.

Note that we can order the range code as follows:

$$[\phi(\alpha_0), \phi(\alpha_1), \ldots, \phi(\alpha_{n-1})]$$

We refer to the list of subscripts of corresponding β_i's as the **permutation code**.

Given a domain code, range code, and permutation code, the corresponding prefix replacement map is uniquely determined.

Example 1.2.5. Consider the two ordered complete binary prefix codes $\{00, 01, 1\}$ and $\{0, 10, 11\}$, with bijection

$$\begin{array}{lcl} \phi(00) & = & 0 \\ \phi(01) & = & 11 \\ \phi(1) & = & 10 \end{array}$$

The domain code is $[00, 01, 1]$, the range code is $[0, 10, 11]$, the permutation code is $[0, 2, 1]$, and the corresponding prefix replacement map is the function f defined as follows:

$$\begin{array}{lcl} f(00\omega) & = & 0\omega \\ f(01\omega) & = & 11\omega \\ f(1\omega) & = & 10\omega \end{array}$$

Then $f(0100111010...) = 1100111010...$, $f(00001011...) = 0001011...$, and $f(1101010...) = 10101010...$.

Example 1.2.6. Consider the prefix replacement map f in Example 1.2.5. This map is derived from domain code $[00, 01, 1]$, range code $[0, 10, 11]$, and permutation code $[0, 2, 1]$. However, we could derive this same map from domain code $[00, 01, 10, 11]$, range code $[0, 100, 101, 11]$ and permutation code $[0, 2, 3, 1]$.

Example 1.2.6 shows us that given a prefix replacement map, the domain code, range code, and permutation code are *not* uniquely determined. This will be important for us in Sections 1.3.1 and 1.3.2.

Returning to the relationship between the Cantor set and binary sequences, we wil now discuss how to represent a prefix replacement map using the middle thirds Cantor set.

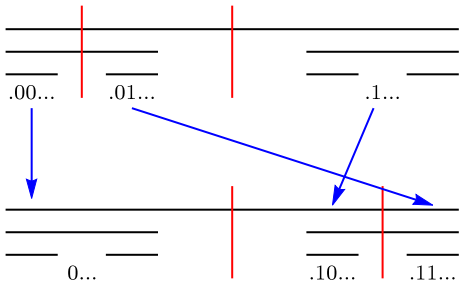


Figure 1.2.2. Two dydadic subdvisions of the Cantor set and a bijection between their dyadic intervals

Definition 1.2.7. Given a finite binary prefix α, we define I_α as the set of all binary sequences β such that $\alpha < \beta$. We call I_α a **dyadic interval**. A **dyadic subdivision** is a partition of the Cantor set into finitely many dyadic intervals.

For example, if C is the middle thirds representation of the Cantor set, then $I_0 = C \cap [o, \frac{1}{3}]$ and $I_1 = C \cap [\frac{2}{3}, 1]$. Together I_0 and I_1 form a dyadic subdivision of the Cantor set. In general, there is one dyadic subdivision on the Cantor set for each complete binary prefix code.

Two dyadic subdivisions of the Cator set are shown in Figure 1.2.2. Observe the top partition. The first dyadic interval is I_{00}, the second is I_{01}, and the third is I_1. Now observe the bottom partition. The first dyadic interval is I_0, the second is I_{10}, and the third is I_{11}.

Given these dyadic subdivisions of the Cantor set with the same number of dyadic intervals, we can construct the following bijective map f between them,

$$\begin{array}{rcl} f(I_{00}) & = & I_0 \\ f(I_{01}) & = & I_{11} \\ f(I_1) & = & I_{10} \end{array}$$

as shown in Figure 1.2.2. This bijective map is precisely the prefix replacement map given in Example 1.2.5.

1.3 Thompson's Group V

Definition 1.3.1. Thompson's group V is the group of all binary prefix replacement maps under function composition.

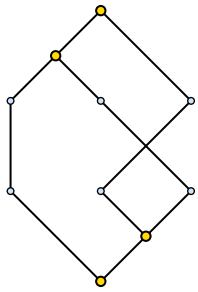


Figure 1.3.1. A tree diagram representing the element of V with domain code $[00, 01, 1]$, range code $[0, 10, 11]$, and permutation code $[2, 0, 1]$

We may refer to composition of two elements of V as **multiplication** of two elements of V, but these operations are equivalent in all respects. It is not obvious that the composition of two prefix replacement maps is also a prefix replacement map; we will discuss this in Section 1.3.2.

We remark that the **inverse** of an element of V is precisely the inverse of the prefix replacement map which defines it. We can obtain this by swapping the domain code with the range code and inverting the permutation code as if it were a member of the symmetric group S_n, where n is the length of the permutation.

Example 1.3.2. Consider once again the element of V presented in Example 1.2.5. The inverse of this element has domain code $[0, 10, 11]$, range code $[00, 01, 1]$ and permutation code $[0, 1, 2]$.

A **tree diagram** is a visual representation of an element of V using two binary trees. We call the one on top the **upper tree**, the leaves of which correspond to the binary prefixes which make up the domain code of the element. Likewise, we call the one on the bottom the **lower tree**, the leaves of which correspond to the binary prefixes which make up the range code. We draw lines between the leaves of the upper tree and the lower tree to represent the permutation code.

Example 1.3.3. Recall the element of V from Example 1.2.5 with domain code $[00, 01, 1]$, range code $[0, 10, 11]$, and permutation code $[0, 2, 1]$. Figure 1.3.1 shows this element as a tree diagram.

1.3.1 Reduction in Thompson's Group V

Central to our discussion of reduction in Thompson's group V is the occurrence of pairs of binary sequences called a carets.

Definition 1.3.4. Given a binary prefix α, a **caret** is is the pair $\alpha 0, \alpha 1$. We call the $\alpha 0$ the **left child** and $\alpha 1$ the **right child** of the caret. In a domain or range code, if a left child and corresponding right child appear in that order, we say that this is an **ordered caret**.

Example 1.3.5. Consider the domain code $[00, 0100, 0101, 011, 1]$. The prefixes 0100 and 0101 form an ordered caret.

Suppose we have an element of V with domain code containing the caret $\alpha 0, \alpha 1$ and range code containing the caret $\beta 0, \beta 1$, for binary prefixes α and β. A **reduction** occurs when $\alpha 0$ maps to $\beta 0$ and $\alpha 1$ maps to $\beta 1$ in the permutation code. We perform the reduction in the domain by replacing the two prefixes of the caret with the single prefix α, and in the range by replacing the two prefixes of the caret with the single prefix β. We must also modify the permutation by removing the element which formerly corresponded to the left children of the reduced carets and subtracting 1 from every element greater than the removed element.

Example 1.3.6. Consider the element with domain code $[0, 100, 101, 11]$, range code $[00, 01, 10, 11]$ and permutation $[3, 0, 1, 2]$. The caret $100, 101$ occurs in the domain code at indices 1 and 2. Similarly, we have the caret $00, 01$ occurring at indices 0 and 1 in the range code. Checking the permutation, we see that 100 maps to 00 and 101 maps to 01, so a reduction can occur here. The greatest common prefix of 100 and 101 is 10, and the greatest common prefix of 00 and 01 is 0. Thus the reduced element has domain code $[0, 10, 11]$, range $[0, 10, 11]$ and permutation $[2, 0, 1]$. This reduction is shown in Figure 1.3.2.

Note that once we determine that a reduction is necessary, we need several pieces of information to go about reducing the element. We need the indices of the left and right child of the caret to be reduced in the domain code, as well as the greatest common prefix of those two sequences. Similarly, we need the indices of the left and right child of the caret to be reduced in

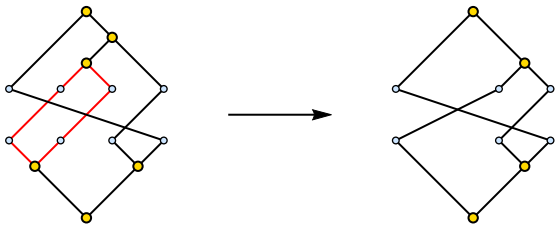


Figure 1.3.2. Reduction of the element of V with domain code $[0, 100, 101, 11]$, range code $[00, 01, 10, 11]$ and permutation $[3, 0, 1, 2]$

the range code, as well as the greatest common prefix of those two sequences. Finally, we need to know which element connects the right children of these two carets so that we can remove it from the permutation and decrement all elements greater than this value.

A representation of an element of V is **reduced** if there are no possible reductions to perform. Each element of V has a unique reduced form.

Definition 1.3.7. For a given reduced element of V, we will call the number of binary prefixes in the domain code the **size** of the element.

Note that the size of an element of V is also equal to the number of binary prefixes in the range code and the number of whole numbers in the permutation code.

Example 1.3.8. Recall the element of V presented in Example 1.2.5 which has domain code $[00, 01, 1]$, range code $[0, 10, 11]$, and permutation code $[0, 2, 1]$. We can see that this element is reduced. The size of this element is 3.

1.3.2 Composition in Thompson's Group V

Recall that we multiply two elements of V using function composition. The following example depicts one way of doing this.

Example 1.3.9. Consider the following two elements of V:

$$\begin{array}{lcl} f(00\omega) & = & 100\omega \\ f(01\omega) & = & 0\omega \\ f(10\omega) & = & 101\omega \\ f(11\omega) & = & 11\omega \end{array} \qquad \begin{array}{lcl} g(0\omega) & = & 000\omega \\ g(100\omega) & = & 01\omega \\ g(101\omega) & = & 001\omega \\ g(11\omega) & = & 1\omega \end{array}$$

Then the composition $g \circ f$ is defined by the following prefix replacement map, which send elements of the domain code of f to elements of the range code of g:

$$\begin{aligned}(g \circ f)(00\omega) &= 01\omega \\ (g \circ f)(01\omega) &= 000\omega \\ (g \circ f)(10\omega) &= 001\omega \\ (g \circ f)(11\omega) &= 1\omega\end{aligned}$$

This is a relatively simple example because the prefixes in the range code of f correspond to the prefixes in the domain code of g. We can use this idea to multiply other elements of V. Given two elemnts $f, g \in V$ for which the range code of f does not necessarily match the domain code of g, we can "un-reduce" them by adding carets to them until they do match.

Example 1.3.10. Consider the following two elements of V:

$$\begin{aligned}f(00\omega) &= 10\omega \\ f(01\omega) &= 11\omega \\ f(1\omega) &= 0\omega\end{aligned} \qquad \begin{aligned}g(00\omega) &= 0\omega \\ g(01\omega) &= 11\omega \\ g(1\omega) &= 10\omega\end{aligned}$$

Then the range code of f is $[0, 10, 11]$ and the domain code of g is $[00, 01, 1]$. In order to "un-reduce" f, we will first add the caret with left child 00 and right child 01 to the range code, and modify the domain code accordingly. Doing so yields the following equivalent representation of f:

$$\begin{aligned}f(00\omega) &= 10\omega \\ f(01\omega) &= 11\omega \\ f(10\omega) &= 00\omega \\ f(11\omega) &= 01\omega\end{aligned}$$

Similarly, we can add the caret with left child 10 and right child 11 to the domain code of g, modifying the range code accordingly, to obtain the following equivalent representation of g:

$$\begin{aligned}g(00\omega) &= 0\omega \\ g(01\omega) &= 11\omega \\ g(10\omega) &= 100\omega \\ g(11\omega) &= 101\omega\end{aligned}$$

Now that the the range code of f is equivalent to the domain code of g, we can compose g with f to obtain the following element of V:

$$\begin{aligned}(g \circ f)(00\omega) &= 100\omega \\ (g \circ f)(01\omega) &= 101\omega \\ (g \circ f)(10\omega) &= 0\omega \\ (g \circ f)(11\omega) &= 11\omega\end{aligned}$$

In general, we compose elements of V using the fact that there is not a uniquely determined domain code, range code, and permutation code for a given prefix replacement map. Given $f \in V$ with domain code $[\alpha_0, \alpha_1, \ldots, \alpha_{n-1}]$, range code $[\beta_0, \beta_1, \ldots, \beta_{n-1}]$, and permutation code $[p_0, p_1, \ldots, p_{n-1}]$, and $g \in V$ with domain code $[\gamma_0, \gamma_1, \ldots, \gamma_{n-1}]$, range code $[\lambda_0, \lambda_1, \ldots, \lambda_{n-1}]$, and permutation code $[q_0, q_1, \ldots, q_{n-1}]$, we obtain the composition $g \circ f$ by stragically adding carets to f and g until the range code of f matches the domain code of g, modifying their permutation codes accordingly. Once we acheive this matching, we use function composition to obtain $g \circ f$.

In the following section we introduce certain directed graphs called abstract strand diagrams as an alternative representation for elements of V. We will use abstract strand diagrams in Chapter 3 when we describe the solution to the conjugacy problem as presented by Belk and Matucci [2].

1.4 Abstract Strand Diagrams

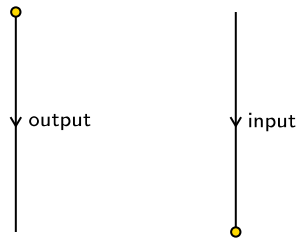
Definition 1.4.1. A **directed graph** is a 4-tuple $G = (V, E, s, t)$ such that

- V is a set of vertices,
- E is a set of edges,
- the function $s : E \to V$ indicates a source vertex for each edge, and
- the function $t : E \to V$ indicates a target vertex for each edge.

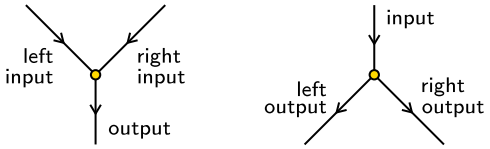
Note that our definition for directed graphs allows for loops and multiple edges.

Definition 1.4.2. Let $G = (V, E)$ be a directed graph. For each vertex $v \in V$, let ρ_v denote the counterclockwise order of the edges connectd to v. Then the set $\{\rho_v \mid v \in V\}$ is called a **rotation system** for G.

Definition 1.4.3. An **abstract strand diagram** is a finite acyclic directed graph with a rotation system that has the following properties:



(a) A source vertex (left) and a sink vertex (right)



(b) A merge vertex (left) and a split vertex (right)

Figure 1.4.1. The four types of vertices in an abstract strand diagram

1. The graph has exactly one univalent **source** with an outgoing edge and exactly one univalent **sink** with an incoming edge,

2. Every other vertex is either

   (a) a **split**, a trivalent vertex with one incoming edge and two outgoing edges, or

   (b) a **merge**, a trivalent vertex with two incoming edges and one outgoing edge.

We equivalently refer to edges as **strands**.

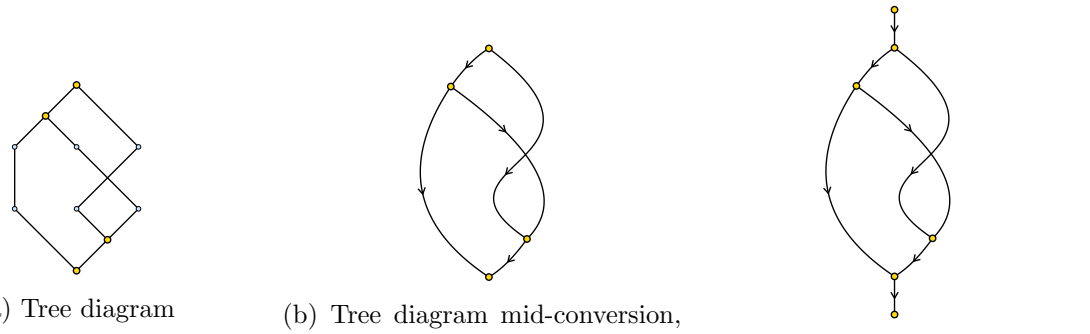
The four types of vertices (source, sink, split, and merge) are shown in Figure 1.4.1.

We show the procedure to obtain an abstract strand diagram from a tree diagram in Figure 1.4.2. The top of each caret becomes a vertex; in the upper tree we have splits, and in the lower tree we have merges. The edges of the tree diagram become the edges of the graph, all of which are directed downard. Finally, we connect the topmost vertex to a new vertex, the source, and the bottommost vertex to another new vertex, the sink.

1.4.1 Reduction of Abstract Strand Diagrams

There are two moves that we use to reduce abstract strand diagrams, both of which are shown in Figure 1.4.3. Both types involve one split and one merge.

A Type I reduction occurs whenever the left output of a split a becomes the left input of a merge b, and the right output of a becomes the right input of b. After the reduction, both a and b are deleted, as well as the two edges that had connected them, leaving one edge in their place.



(a) Tree diagram

(b) Tree diagram mid-conversion, drawn as directed graph

(c) Tree diagram after converstion to abstract strand diagram

Figure 1.4.2. An example of converting a tree diagram to a strand diagram

A Type II reduction occurs when the output of a merge a becomes the input of a split b. After the move takes place, both a and b are deleted as well as the strand that had connected them, leaving two edges in their place.

1.4.2 Composition of Abstract Strand Diagrams

Suppose that given two abstract strand diagrams f and g we wish to obtain their composition $g \circ f$. To do so we "concatenate" them by placing g below f and connecting them in the following way:

1. Remove the sink of f
2. Remove the source of g
3. Connect the new bottommost vertex of f (which will always be a merge) to the new topmost vertex of g (which will always be a split)
4. Reduce the resulting abstract strand diagram

The above procedure is shown in Figure 1.4.4

Figure 1.4.3. The two moves to reduce abstract strand diagrams: Type I (left) and Type II (right).

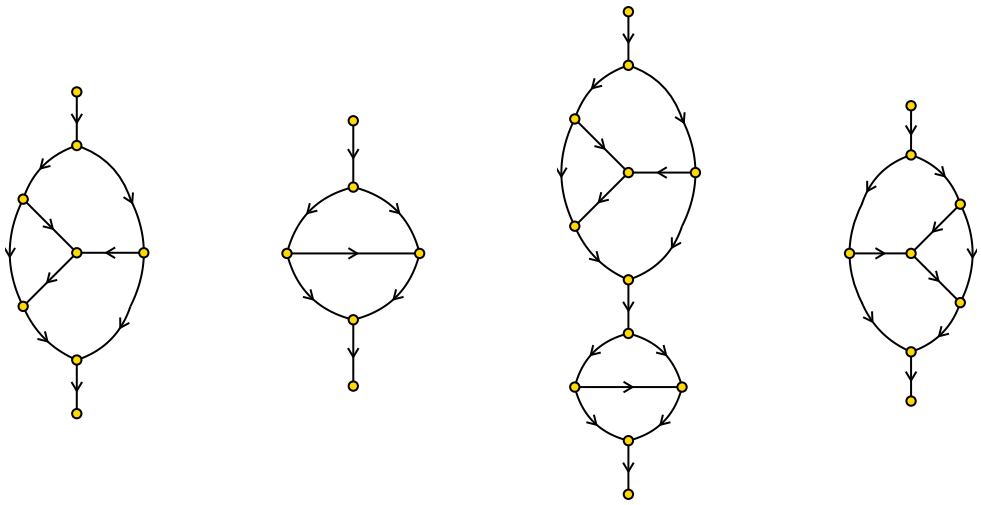


Figure 1.4.4. Composition of two abstract strand diagrams. From left to right, we have f, g, $g \circ f$, and $g \circ f$ reduced

2
Implementing Thompson's Group V

Now that we have introduced Thompson's group V we will present our implementation. We will first describe how to store complete binary prefix replacement maps, in a class that we call `TreeDiagram`, as well as the necessary operations on these objects. We will then outline the conversion of a `TreeDiagram` object to an `AbstractStrandDiagram` object and describe the structures we created to store and reduce abstract strand diagrams.

2.1 Implementing Tree Diagrams in Thompson's Group V

We implement Thompson's group V in a class called `TreeDiagram` using three basic structures, which are (1) an `Array` of binary prefixes to store the domain code, (2) an `Array` of binary sequences to store the range code, and (3) an `Array` of whole numbers to store the permutation code. We will make use of the `Array` operations `get`, `set`, `add`, and `remove`, all of which run in constant time.

Example 2.1.1. Recall the binary prefix replacement map from Example 1.2.5. The instance of this element using our implementation would be a `TreeDiagram` t with t`.domain` $= [00, 01, 1]$, t`.range` $= [0, 10, 11]$, and t`.permutation` $= [0, 2, 1]$.

In our code and our algorithms we refer to the "domain code" as simply the `domain` for simplicity. The same applies to the "range code" and the "permutation code," which we will henceforth refer to as the `range` and the `permutation` respectively.

The runtime to initialize a `TreeDiagram` is negligible.

2.1.1 Implementing the Inverse Operation

We calculate the inverse of a `TreeDiagram` using Algorithm 1. The algorithm proceeds by switching the `domain` with the `range`, which happens on lines 1 and 2. Then we traverse through the `permutation` and switch each entry with its position in the array on lines 5 and 6. This operation executes in linear time $\mathcal{O}(n)$ where n is the length of the permutation, owing to the loop (while) on line 4.

Input: `TreeDiagram` x
Output: `inverse of` x

```
1 inverseDomain = x.range
2 inverseRange = x.domain
3 i = 0
4 while i < x.permutation.size() do
  |  // invert permutation code
5 |  p = x.permutation.get(i)
6 |  inversePerm.set(p) = i
7 end
  // the inverse of x has domain code, range code, and permutation code
  inverseDomain, inverseRange, and inversePerm, respectively
```

Algorithm 1: Algorithm to determine the inverse of a `TreeDiagram`

We remark that in our final conjugacy checking program this method is not used.

2.1.2 Implementing Reduction

Our method for reducing a `TreeDiagram` will begin by creating working copies of the `domain`, `range`, and `permutation` of the object. We will modify these `Array`s dynamically as we reduce since, as we will see, performing reductions can create yet more redexes. At the end of this method, the `domain`, `range`, and `permutation` objects that we have created will reperesnt the reduced element.

Recall that three conditions must be met for a reduction to occur in an `TreeDiagram`:

1. Two whole numbers $p1, p2$ must sit in indices $i, i+1$ in the `permutation`,
2. The prefixes in the `domain` at indices $i, i+1$ must form an ordered caret (see Definition 1.3.4), and
3. The prefixes in the `range` at indices $p1, p2$ must form an ordered caret (see Definition 1.3.4).

Input: `permutation, domain, range, and index` i `for a TreeDiagram`
Output: `a ReductionInfo object` r

```
1  p1 = permutation.get(i)
2  p2 = permutation.get(i + 1)
3  if p1 == p2 − 1 then
4  |  dom1=domain.get(i)
5  |  dom2=domain.get(i + 1)
6  |  if isOrderedCaret(dom1,dom2) then
7  |  |  ran1=range.get(p1)
8  |  |  ran2=range.get(p2)
9  |  |  if isOrderedCaret(ran1,ran2) then
10 |  |  |  r.isReduction=true
11 |  |  |  r.p2 = p2
12 |  |  |  r.domIdx = i
13 |  |  |  r.domGCP = dom1.getGCP(dom2)
14 |  |  |  r.lastDigit=getLastDigit(domGCP)
15 |  |  |  r.ranIdx = p1
16 |  |  |  r.ranGCP = ran1.getGCP(ran2)
17 |  |  |  return r
18 r.isReduction=false
19 return r
```

Algorithm 2: Algorithm for the helper method `isReduction` called on line 5 of Algorithm 3, which stores all of the needed information to perform a reduction in a `ReductionInfo` object r

We look for these conditions by traversing incrementally through each `Array`, begining at position 0 (the first slot). This process takes place in the helper method `isReduction` which is called on line 5 of Algorithm 3 and shown in detail in Algorithm 2. As shown, it takes as input the `permutation`, `domain`, `range`, and current position i of `TreeDiagram`.

We look at the `permutation` first; let $p2$ be the entry at position $i+1$. Then if the entry at position i is $p1 = p2 - 1$ then the first condition is met. These operations occur on lines 1-3

Next we look at the `domain`. It is the positions i and $i+1$ in the `permutation` which correspond to the elements of the `domain`, so we check the binary prefixes in positions i and $i+1$ to see if they form an ordered caret. We perform these checks on lines 4-6, then move on to look at the `range`. The entries $p1$ and $p2$ correspond to the elements of the `range`, so we check the binary prefixes at positions $p1$ and $p2$ to see if they form an ordered caret. This occurs on lines 7-9. If this condition is met, then we know that a reduction must take place.

Recall from Section 1.3.1 that once all three conditions are met then several piece of information are needed to perform the reduction:

1. The entry $p2$ from the permutation code,
2. The index i of the ordered caret in the domain,
3. The greatest common prefix of the ordered caret in the domain,
4. The index $p1$ of the ordered caret in the range, and
5. The greatest common prefix of the ordered caret in the range.

The aforementioned helper method, `isReduction`, as well as a sub-class within the `TreeDiagram` class, `ReductionInfo`, stores that information for use in reduction; this occurs on lines 10-16 in Algorithm 2.

Returning to Algorithm 3, lines 8-10 invoke three more helper methods, the first of which on line 8 performs the appropriate reduction on the permutation code. The method `reducePerm()` is detailed in Algorithm 4. We incrementally traverse the `permutation` and check for two conditions:

1. If the current entry is equal to $p2$ then we must delete this entry. This happens on line 4 .
2. If the entry is greater than $p2$, we must decrement it by 1. This happens on line 6.

This method is linear on the size of the `permutation` due to the loop (while) on line 2.

```
Input: TreeDiagram x
Output: x in reduced form
1  i = 0
2  boolean foundReduction
3  while i < x.permutation.size()−1 do
       // check permutation, domain, and range at i for potential reductions
4      foundReduction = false
5      r = x.isReduction(x.permutation, x.domain, x.range, i)
6      if r.isReduction then
7          foundReduction = true
8          x.permutation = reducePerm(x.permutation, r) // see Algorithm 4
9          x.domain = reduceDomain(x.domain, r) // see Algorithm 5
10         x.range = reduceRange(x.range, r)
       // if no reduction occurred, increment i to keep looking
11     if !foundReduction then
12         i++
       // if a reduction occurred in the right child, decrement i to see if
       another reduction was created
13     else if i > 0 && r.lastDigit.equals("1") then
14         i−−
       // else a reduction occurred in the left child, so we should leave i as
       is to check this spot again
15 end
```

Algorithm 3: Algorithm to reduce a `TreeDiagram`

The `reduceDomain()` helper method, called on line 9 of Algorithm 3, performs the appropriate reduction on the `domain`. Algorithm 5 details this method. We incrementally traverse the indices of the `domain` while checking for two conditions:

1. If the current index is equal to $domIdx$, then we replace the entry with $domGCP$. This happens on line 3.

2. If the current index is equal to $domIdx + 1$, then we delete the entry from the `domain`, which happens on line 5.

This algorithm is linear on the size of the domain due to the while loop on line 2.

The `reduceRange()` helper method, called on line 10 of Algorithm 3, performs the appropriate reduction on the `range`. Similarly to reducing the `domain`, we incrementally traverse the indices of the `range` while checking for two conditions:

Input: `permutation and ReductionInfo object` r
Output: `reduced permutation`

```
1  j = 0
2  while j <permutation.size do
3  |  current = permutation.get(j)
4  |  if current == r.p2 then
5  |  |  permutation.remove(j)
6  |  else if current > r.p2 then
7  |  |  permutation.set(j,current-1)
8  |  else
9  |  |  j++
10 end
11 return permutation
```

Algorithm 4: Algorithm for `reducePerm()` helper method called on line 8 of Algorithm 3

1. If the current index is equal to $p1$, then we replace the entry with the $ranGCP$

2. If the current index is equal to $p1 + 1$, then we delete the entry from the `range`

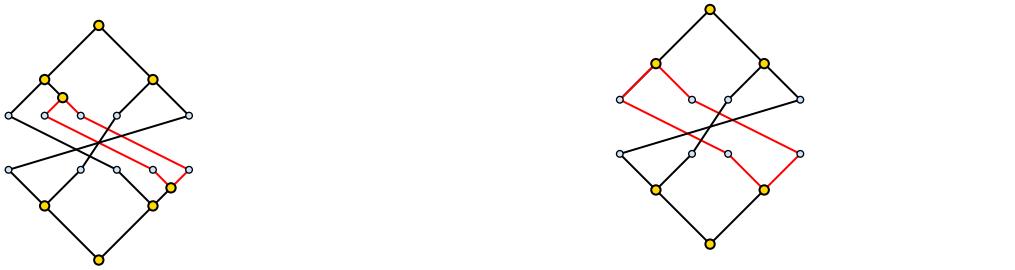
The algorithm for this method proceeds in the same way as Algorithm 5 and also runs in linear time.

Returning to Algorithm 3, we then determine where to next look for a reduction. Three cases follow:

Input: `domain and ReductionInfo object` r
Output: `reduced domain`

```
1  j = 0
2  while j <domain.size() do
3  |  if j == r.domIdx then
4  |  |  domain.set(j, r.domGCP)
5  |  else if j == r.domIdx + 1 then
6  |  |  domain.remove(j)
7  |  j++
8  end
9  return domain
```

Algorithm 5: Algorithm for `reduceDomain()` helper method, which is called on line 9 of Algorithm 3



(a) Before reduction, index $i = 1$

(b) After reduction, new reduction created at index $i = 0$

Figure 2.1.1. An example of a reduction in a right child of a caret giving rise to a subsequent reduction

Case 1: No reduction took place

In this case, we simply increment i to check the next possible location for a reduction in the permutation. This happens on line 11.

Case 2: A reduction occurred in the right child of a caret

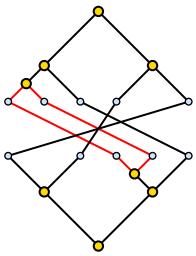
In this case, we have created the possibility for another reduction to take place at the previous location. We therefore must decrement i by 1 so as to check the previous location for a new reduction. This occurs on line 13. See Figure 2.1.1 for an example of this case.

Case 3: A reduction occurred in the left child of a caret

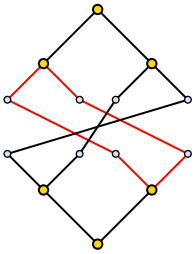
In this case, we have created the possibility for another reduction to take place at this location. We therefore do not increment i so as to check this location again for a new reduction. See Figure 2.1.2 for an example of this case.

When we have finished traversing each `Array`, all necessary reductions will have taken place and the resulting `Arrays` representing the `domain`, `range`, and `permutation` will accurately reflect the reduced `TreeDiagram`.

Reducing a `TreeDiagram` is quadratic in n, where n is the length of the original permutation. The outer loop (while) on line 3 of Algorithm 3 executes a number of times linear in n. However, when a reduction is found we must reduce the domain, range, and permutation, all of which also happen in linear time, hence, the $\mathcal{O}(n^2)$ runtime of this algorithm. We remark that in our final conjugacy checking program this method is not used.



(a) Before reduction, index $i = 0$



(b) After reduction, new reduction created at index $i = 0$

Figure 2.1.2. An example of a reduction in a left child of a caret giving rise to a subsequent reduction

2.1.3 Implementing Composition

Recall that given $f, g \in V$, we obtain the composition $g \circ f$ by (1) "un-reducing" them until the range of f is equal to the domain of g and (2) composing the permutations of the un-reduced elements. The "un-reduced" domain of f, the "un-reduced" range of g, and the composition of permutations represent the composition $g \circ f$. Refer to Section 1.3.2 for a full explanation of composition in V.

Our method to obtain $g \circ f$ given `TreeDiagrams` f and g will begin by incrementally traversing both the `range` of f and the `domain` of g and performing "un-reductions" until they are equal to each other.

Similarly to the reduction process, we begin by initializing a whole number i at 0 to compare the binary prefixes at position i in both the `range` of f and the `domain` of g, which we call `rangefCurrent` and `domaingCurrent`. One of two cases then takes place:

Case 1: The prefixes `rangefCurrent` and `domaingCurrent` are not equal (Algorithm 6 line 5)

We must create a caret in either f or g. Since `rangefCurrent` and `domaingCurrent` are not equal, it is necesarily the case that one is the prefix of the other. Then two more cases emerge:

Subcase 1: The prefix `rangefCurrent` is shorter than `domaingCurrent` (Algorithm 6 line 6)

We conclude that `rangefCurrent` is a prefix of `domaingCurrent`, so we must create a caret in f. We have the position for the caret in the `range` of f, which is i. We must

Input: `two elements of` V, f `and` g
Output: `the composition of` f `and` g, $g * f$

```
1  i = 0
2  while i < f.range.size() do
3  |  rangefCurrent = f.range.get(i)
4  |  domaingCurrent = g.domain.get(i)
5  |  if !rangefCurrent.equals(domaingCurrent) then
6  |  |  if rangefCurrent.getLength() < domaingCurrent.getLength() then
7  |  |  |  rangefIdx = i
8  |  |  |  domainfIdx = f.permutation.indexOf(i)
9  |  |  |  f.range = addCaret(f.range, rangefIdx)
10 |  |  |  f.permuutation = addCaret(f.permutation, domainfIdx)
11 |  |  |  f.domain = addCaret(f.domain, domainfIdx)
12 |  |  else
13 |  |  |  domaingIdx = i
14 |  |  |  rangegIdx = g.permutation.get(i)
15 |  |  |  g.domain = addCaret(g.domain, domaingIdx)
16 |  |  |  g.permutation = addCaret(g.permutation, domaingIdx)
17 |  |  |  g.range = addCaret(g.range, rangegIdx)
18 |  |  end
19 |  else
20 |  |  i++
21 end
22 newPermutation = composePerms(f.permutation, g.permutation)
23 result = ElementOfV(f.domain, g.range, newPermutation)
24 return reult
```

Algorithm 6: Algorithm to compose two elements of V

use the `permutation` to determine the index of the corresponding binary prefix in the `domain`, which we do on line 8 of Algorithm 6. We use the helper method `addCaret` to alter the `domain`, `range`, and `permutation` accordingly (lines 9- 11). Algorithm 7 outlines this method for the `range`, but it works the same way for the `domain` and `permutation`. Each of these calls takes $\mathcal{O}(n)$ time for current size n of the element, owing to the loop (while) on line 5.

In the `domain` and `range`, we take the current prefix α and create a caret with left child $\alpha 0$ (line 2) and right child $\alpha 1$ (line 3). The `leftChild` takes the place i of the original prefix α (line 7) and we insert the `rightChild` into position $i + 1$ (line 9). The `add` method which is called on line 9 of Algorithm 7 creates a new `Array` entry

at the specified position. As a result, any subsequent prefixes are shifted right from their former position p to $p+1$.

Input: `range of a TreeDiagram and position` i `to add caret`
Input: `updated range with added caret`

```
1  α = range.get(i)
2  leftChild = α0
3  rightChild = α1
4  int j = 0
5  while j<range.size() do
6  |  if j==i then
7  |  |  range.set(j,leftChild)
8  |  else if j==i+1 then
9  |  |  range.add(j,rightChild)
10 |  j++
11 end
12 return domain
```

Algorithm 7: Algorithm for `addCaret()` helper method used in Algorithm 6

Subcase 2: The prefix `domaingCurrent` is shorter than `rangefCurrent` (Algorithm 6 line 12)

In this case, we must create a caret in g. We have the position for the caret in the `domain` of g, which is i. We must use the permutation to determine the index of the corresponding binary prefix in the `range`, which we do on line 14 of Algorithm 6. We use the helper method `addCaret` to alter the `domain`, `range`, and `permutation` accordingly on lines 15-17.

Case 2: The two binary prefixes are equal (Algorithm 6 line 19)

This is the desired case. We simply increment i to check the next position in the `range` of f and the `domain` of g.

The outer loop on line 2 executes n times, and each `addCaret` method call also takes linear time. Thus our composition algorithm runs in $\mathcal{O}(n^2)$ time, where n is the size of f plus the size of g. We remark that in our final conjugacy checking program this method is not used.

2.2 Implementing Abstract Strand Diagrams

In order to construct an abstract strand diagram we require a data type called `Strand` to represent edges (which we will equivalently refer to as "strands"), as well as a data type called `Vertex` to represent vertices. Once we have these implementations, an `AbstractStrandDiagram` is essentially a `DoublyLinkedList` of `Vertex` and `Strand` objects.

Throughout our implementation, we use the following terms to describe the roles of incoming and outgoing edges at a given vertex:

- Input: parent
- Left input: left parent
- Right input: rigth parent
- Output: child
- Left output: left child
- Right output: right child

which we discuss more in the following section.

2.2.1 Implementing the Strand Class

Any edge in an abstract strand diagram is connected to two vertices. Since the edges are directed, we refer to these as `beginVertex` and `endVertex`. In addition to knowing this information, each instance of a `Strand` stores the role it plays at its begin vertex (either `child`, `lchild`, or `rchild`) as well as the role it plays at its end vertex (either `parent`, `lparent`, or `rparent`). We use an enumerated data type called `Role` to store this set of constants. For a visual representation of a `Strand` object refer to Figure 2.2.1.

2.2.2 Implementing the Vertex Class

Each instance of a `Vertex` knows which type it is: `source`, `sink`, `split`, or `merge`. We use an enumerated data type called `Type` to store this set of constants. Additionally, each `Vertex`

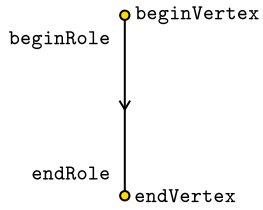
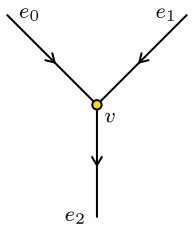


Figure 2.2.1. A `Strand` object stores data about its vertices and the role it plays at each vertex

stores a `Dictionary` with entries `Role`: `Strand`. A source, for example, would store a dictionary containing only one entry for which the key is `child`. A merge, however, would store a dictionary containing three entries, for which the keys are `lparent`, `rparent`, and `child`. Lastly, each vertex stores a reference to the `Node` which contains it in the `DoublyLinkedList` which defines the `AbstractStrandDiagram`.

Example 2.2.1. Observe the `Vertex` v below of type `merge`



The dictionary of strands for v is the following:

$$\{\texttt{lparent} : e_0,\ \texttt{rparent} : e_1,\ \texttt{child}, e_2\}$$

2.2.3 Converting a Tree Diagram into an Abstract Strand Diagram

Recall that a `TreeDiagram` object uses a list of binary prefixes to store the `domain`, a list of binary prefixes to store the `range`, and a list of whole numbers to store the `permutation`. In a tree diagram, the top of each caret comes with an associated binary prefix, as shown in Figure 2.2.2. In the upper tree, each of these will become a split vertex. Using these associated binary prefixes will aid in our construction of the corresponding `AbstractStrandDiagram`. Similary, we use the caret prefixes in the lower tree to construct the merge vertices in the abstract strand diagram. With

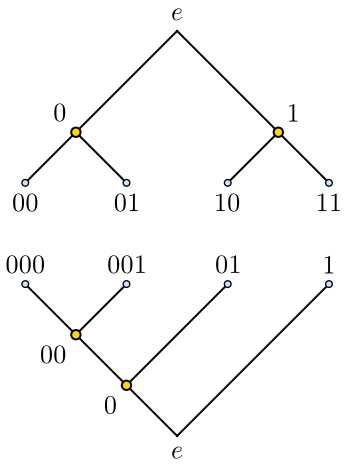
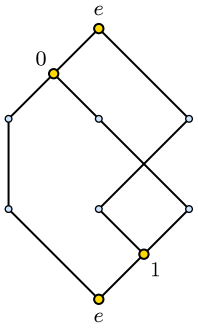


Figure 2.2.2. Each vertex in a tree diagram has an associated binary prefix

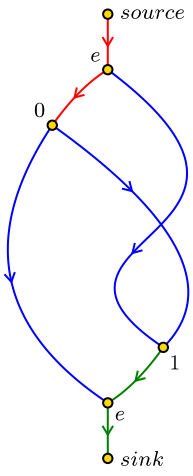
this information, we convert a `TreeDiagram` to an `AbstractStrandDiagram` in the following steps, as shown in Algorithm 8. Refer Figure 2.2.3 for an example of this algorithm at work.

1. **Create source and sink vertices**

   Because they are not represented in the original tree diagram, the source vertex and sink vertex do not have associated binary prefixes in the same way that split vertices and merge vertices do. We create them on lines 1 and 2 of Algorithm 8 by passing the appropriate `Type` to the `Vertex` constructor. We will use the strings "source" and "sink" as their prefix identifiers, respectively, and deal with them as special cases below. We create a



(a) The `TreeDiagram` with `domain` [00,01,1], `range` [0,10,11], and `permutation` [0,2,1]



(b) The corresponding `AbstractStrandDiagram`

Figure 2.2.3.

Input: `one element` $f \in V$`, i.e. a TreeDiagram object`
Output: `the abstract strand diagram representing` f

```
1  source = Vertex(sink)
2  sink = Vertex(sink)
3  DoublyLinkedList vertices
4  vertices.add(source)
5  vertices.add(sink)
6  createSplits()
   // see Algorithm 9
7  createMerges()
   // see Algorithm 10
8  connectSplitsAndMerges()
   // see Algorithm 11
9  result = AbstractStrandDiagram(vertices)
10 return result
```

Algorithm 8: Algorithm to convert an element of V in `TreeDiagram` form to an `AbstractStrandDiagram`

`DoublyLinkedList` called `vertices` and add these two vertices to it on lines 3-5, which we will use at the end of this procedure to construct the resulting `AbstractStrandDiagram`. Assume that all other `Vertex` objects created in the following steps are also added to this `vertices`.

2. **Use domain to create and connect all split vertices**

We show our method to create all necessary `split` type `Vertex` objects in Algorithm 9. Refer to Figure 2.2.3b for an example of this method; after this step we will have created the red `Strands` and the `Vertex` objects labeled *source*, e, and 0.

We need to obtain the set of binary prefixes which correspond to each split vertex in the upper tree of the given element. We initialize this `Set` on line 1 and call it `splitPrefixes`. Each binary prefix in the `domain` corresponds to a leaf in the upper tree, and each 0 or 1 in the prefix indicates location in the left or right child of a caret, respectively. The `Set` type does not allow duplicates; therefore, if we include all of the proper prefixes for all of the prefixes in the `domain`, then we will obtain the desired `Set` of `splitPrefixes`. We include e to represent the empty string in every set, since the empty string is a prefix of

all binary prefixes. We compute the proper prefixes for each sequence b in `domain` and add them to `splitPrefixes` on lines 2-3.

Now we can create one vertex for each such prefix identifier. We also create a `Dictionary` called `splits` with entries $prefix$:`Vertex` so that we can access the correct `Vertex` as needed throughout the rest of the algorithm. This occurs on lines 5-9.

The next step is to connect each `split` type `Vertex` to its parent `Vertex`. For the `Vertex` whose identifier is e, we know that the parent `Vertex` is always the `source` type `Vertex` since this is the case in all non-empty abstract strand diagrams. Furthermore, we know that the role of the connecting `Strand` s at the `source` type `Vertex` is `child` and the role of s at the `split` is `parent`. We address this case on lines 14-16 in Algorithm 9. For any other `split` type `Vertex` v with binary prefix b, the parent `split` type `Vertex` w is that which is identified with the largest proper prefix of b (lines 18-19). We know that the role of the connecting `Strand` s at v is `parent`. If the last digit of b is a 0, then the role of s at w is `lchild` (line 20). Else (if the final digit of b is a 1) the role of s at w is `rchild` (line 23). With this information, we can attach our `Strand` and `Vertex` objects accordingly (lines 24-25).

Algorithm 9 runs in quadratic time, owing to the loop (for) on line 11 and the `Dictionary` lookup on line 19.

3. **Use range to create and connect all merge vertices**

We show our method to create all necessary `merge` type `Vertex` objects in Algorithm 10. Refer to Figure 2.2.3b for an example of this method; after this step we will have created the green `Strands` and the `Vertex` objects labeled *sink*, e, and 1.

We need to obtain the set of binary prefix codes which correspond to each merge vertex in the lower tree of the given element. We initialize this `Set` on line 1 and call it `mergePrefixes`. Each binary prefix in the `range` corresponds to a leaf in the lower tree, and each 0 or 1 in the prefix indicates location in the left or right child of a caret,

Input: a `TreeDiagram` f
Output: `the split vertices to include in the AbstractStrandDiagram for` f

```
1  Set<String> splitPrefixes
2  for b in f.domain do
3  |  splitPrefixes.add(b.getProperPrefixes())
4  end
5  Dictionary<String, Vertex> splits
6  for b in splitPrefixes do
7  |  v = Vertex(b)
8  |  splits.put(b, v)
9  |  vertices.add(v)
10 end
11 for b in splits.keys do
12 |  v = splits.get(b)
13 |  Strand s
14 |  if b == e then
15 |  |  v.addStrand(s, parent)
16 |  |  source.addStrand(s, child)
17 |  else
18 |  |  parent = getPrefix(b)
19 |  |  w = splits.get(parent)
20 |  |  if lastDigit(b) == 0 then
21 |  |  |  r = lchild
22 |  |  else
23 |  |  |  r = rchild
24 |  |  w.addStrand(r, s)
25 |  |  v.addStrand(parent, s)
26 end
```

Algorithm 9: Algorithm for `createSplits()` helper method to create the `split` vertices for an `AbstractStrandDiagram` from a `TreeDiagram`

respectively. Therefore, if we include all of the proper prefixes for each prefix in the `range`, then we will obtain the desired set. We compute the proper prefixes for each sequence b in `range` and add them to `mergePrefixes` on lines 2-3.

Subsequently, we create one merge vertex for each such prefix identifier. We also create a `Dictionary` called `merges` with entries $prefix$:`Vertex` so that we can access the correct `Vertex` as needed throughout the rest of the algorithm. This occurs on lines 5-9.

The next step is to connect each `merge` type `Vertex` to its child `Vertex`. For the `Vertex` whose identifier is e, we know that the child `Vertex` is always the `sink` type `Vertex` since this is the case in all non-empty abstract strand diagrams. Furthermore, we know that the role of the connecting `Strand` s at the `sink` type `Vertex` is `parent` and the role of s at the `merge` is `child`. We address this case on lines 14-16. For any other `merge` type `Vertex` v with binary prefix identifier b, the child `split` type `Vertex` w is that which is identified with the largest proper prefix of b (lines 19-20). We know that the role of the connecting `Strand` s at v is `child`. If the last digit of b is a 0, then the role of s at w is `lparent` (line 22). Else (if the final digit of b is a 1), the role of s at w is `rparent` (line 24). With this information, we can attach our `Strand` and `Vertex` objects accordingly (lines 25-26).

Similarly to Algorithm 9, Algorithm 10 runs in quadratic time.

4. **Use permutation to create strands which connect split vertices and strand vertices**

We show our method to connect all `split` type `Vertex` objects and `merge` type `Vertex` objects in Algorithm 11. Refer to Figure 2.2.3b for an example of this method; after this step we will have created the blue `Strand`s.

Now we have created every `Vertex` that we need, but not every `Strand`. The only vertices that remain unconnected are the leaves of the upper tree, which are splits, and the leaves of the lower tree, which are merges. The `permutation` tells us which child of which `split` type `Vertex` connects to which parent of which `merge` type `Vertex`. We begin

Input: one TreeDiagram f
Output: the merge vertices to include in the AbstractStrandDiagram for f

```
1  Set<String> mergePrefixes
2  for b in f.range do
3  |  mergePrefixes.add(b.getProperPrefixes())
4  end
5  Dictionary<String, Vertex> merges
6  for b in mergePrefixes do
7  |  v = Vertex(b)
8  |  merges.put(b, v)
9  |  vertices.add(v)
10 end
11 for b in merges.keys do
12 |  v = merges.get(b)
13 |  Strand s
14 |  if b == e then
15 |  |  sink.addStrand(parent,s)
16 |  |  v.addStrand(child,s)
17 |  else
18 |  |  else
19 |  |  |  child = getPrefix(b)
20 |  |  w = merges.get(child)
21 |  |  if lastDigit(b)==0 then
22 |  |  |  r = lparent
23 |  |  else
24 |  |  |  r=rparent
25 |  |  v.addStrand(child, s)
26 |  |  w.addStrand(r, s)
27 |  end
28 end
```

Algorithm 10: Algorithm for `createMerges()` helper method to create the `merge` vertices for an `AbstractStrandDiagram` from a `TreeDiagram`

by iterating through the `domain`. The greatest proper prefix of each binary prefix *dom* in the `domain` will give us the prefix identifier *splitId* for the current `split` type `Vertex` v (lines 3-5). Likewise, the greatest proper prefix of each binary prefix *ran* in the `range` will give us the prefix identifier *mergeId* for the current `merge` type `Vertex` w (lines 12-14).

We use the last digit of each prefix identifier to determine what role the newly created `Strand` will play at each vertex. If the last digit of the prefix identifier for the currrent `split` type `Vertex` v is a 0, then the begin role of our `Strand` s at v is `lchild` (line 8). Else (if the last digit is 1) the begin role of s at v is `rchild` (line 10). Similarly, if the last digit of the prefix identifier for the current `merge` type `Vertex` w is a 0, then the end role of our `Strand` s at w is `lparent` (line 17). Else (if the last digit is 1) the end role of s at w is `rparent` (line 19). With this information we can attach our `Strand` and `Vertex` objects accordingly (lines 21-22).

Algorithm 11 runs in quadratic time, owing to the loop (while) on line 2 and various `Dictionary` lookups within the loop.

5. **Use completed vertices list to create new abstract strand diagram**

   Returning to Algorithm 8, we simply call the `AbstractStrandDiagram` constructor on line 9. We pass `vertices` as an argument, which contains all of the necessary information to define the `AbstractStrandDiagram`.

Algorithm 8 runs in $\mathcal{O}(n^2)$ time for `TreeDiagram` length n, owing to quadratic runtime of helper methods `createSplits()`, `createMerges()`, and `connectSplitsAndMerges()`.

2.2.4 Reduction of Abstract Strand Diagrams

Each reduction move of an `AbstractStrandDiagram` involves one `split` type `Vertex` and one `merge` type `Vertex`. To search an `AbstractStrandDiagram` for possible reductions, we will examine each `split` type `Vertex` to see if its surrounding vertices meet the stipulations for a reduction to take place.

Input: a `TreeDiagram` f
Output: `final list of connected vertices to form the AbstractStrandDiagram for` f

```
1  domIdx = 0
2  while domIdx<domain.size() do
3  |  dom = domain.get(domIdx)
4  |  splitId = largestProperPrefix(dom)
5  |  v = splits.get(splitId)
6  |  last = getLastDigit(dom)
7  |  if last == 0 then
8  |  |  beginRole = lchild
9  |  else
10 |  |  beginRole = rchild
11 |  ranIdx=permutation.get(domIdx)
12 |  ran = range.get(ranIdx)
13 |  mergeId = largestProperPrefix(ran)
14 |  w = merges.get(mergeId)
15 |  last = getLastDigit(ran)
16 |  if last == 0 then
17 |  |  endRole = lparent
18 |  else
19 |  |  endRole = rparent
20 |  Strand s
21 |  v.addStrand(beginRole, s)
22 |  w.addStrand(endRole, s)
23 |  domIdx++
24 end
```

Algorithm 11: Algorithm for helper method `connectSplitsAndMerges()` which uses the `permutation` from a `TreeDiagram` to connect each `split` type `Vertex` and `merge` type `Vertex` of the corresponding `AbstractStrandDiagram`

We present our reduction method in Algorithm 12. On line 1 we initialize a `Stack` called `splitsToCheck`, which initially includes every `split` type `Vertex` in the `AbstractStrandDiagram`. We remark that the decision to examine `split` vertices instead of `merge` vertices was arbitrary.

As discussed in Section 1.4.1, there are two cases which can result in a reduction taking place. We describe them below.

Case 1: Type I Reduction

All vertices, strands, and names in this case are shown in Figure 2.2.4 and also correspond to the variable names used in Algorithm 12. Given a split vertex `split` with left child

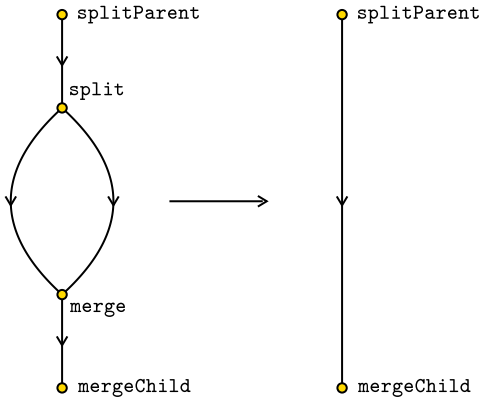


Figure 2.2.4. Requirements and procedure for a type I reduction to take place

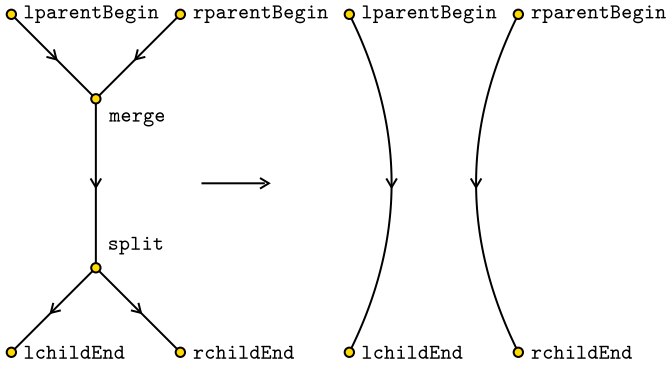


Figure 2.2.5. Requirements and procedure for a type II reduction to take place

Input: `one AbstractStrandDiagram` f
Output: f `in reduced form`

```
1  splitsToCheck = getSplitsToCheck()
2  while splitsToCheck.isNotEmpty() do
3  |  split = splitsToCheck.pop()
4  |  splitParent = split.getStrand(parent).getBeginVertex()
5  |  lchildEnd = split.getStrand(lchild).getEndVertex()
6  |  rchildEnd = split.getStrand(rchild).getEndVertex()
7  |  lchildEndRole = split.getStrand(lchild).getEndRole
8  |  rchildEndRole = split.getStrand(rchild).getEndRole()
9  |  if lchildEnd == rchildEnd && lchildEndRole == lparent && rchildEndRole ==
   |  rparent then
10 |  |  reductionI() // see Algorithm 13
11 |  |  continue
12 |  else if splitParent.type == merge then
13 |  |  reductionII() // see Algorithm 14
14 |  |  continue
15 end
```

Algorithm 12: Algorithm to reduce an `AbstractStrandDiagram`

strand `lchildStrand`, left child vertex `lchildEnd`, right child strand `rchildStrand`, and right child vertex `rchildEnd`, there are three requirements to be met for a type I reduction to take place.

1. The vertex `lchildEnd` and the vertex `rchildEnd` must be equal,
2. The end role of `lchildStrand` at `lchildEnd` must be `lparent`, and
3. The end role of `rchildStrand` at `rchildEnd` must be `rparent`.

We check for these conditions on line 9 of Algorithm 12. If all three of these conditions are met, then we proceed with the reduction.

The helper method for executing a type I reduction is shown in Algorithm 13. We will change the name of `lchildEnd` to `merge` for ease of understanding. We grab the parent `Strand` of `splitParent`, which we call `splitParentStrand`, on line 1, and we grab the child `Vertex` of `merge`, which we call `mergeChild`, on lines 2-3. This allows us to connect the `Vertex` called `splitParent` to `mergeChild` using `SplitParentStrand` on line 5.

Now we can delete `split`, `merge`, and the three unused `Strand`s (lines 6). Lastly, we grab any adjacent `split` type `Vertex` objects and add them to `splitsToCheck` to ensure that we find any newly created reductions (lines 7-10).

Input: `one AbstractStrandDiagram` f
Output: f `after performing a type I reduction`

```
1  splitParentStrand = split.getStrand(parent)
2  mergeChildStrand = merge.getStrand(child)
3  mergeChild = mergeChildStrand.getEndVertex()
4  endRole = mergeChildStrand.getEndRole()
5  mergeChild.addStrand(endRole, splitParentStrand)
6  delete(split,merge,lchildStrand, rchildStrand,mergeChildStrand)
7  if splitParent.type == split then
8  |  splitsToCheck.push(splitParent)
9  if mergeChild.type == split then
10 |  splitsToCheck,push(mergeChild)
```

Algorithm 13: Algorithm for `reductionI()` helper method, called on line 10 of Algorithm 12

Due to the absence of any repeated structures, Algorithm 13 runs in constant time.

Case 2: Type II Reduction

Note that all vertices, strands, and names in this case are represented in Figure 2.2.5 and also correspond to the variable names used in Algorithm 12. Given the `Vertex` called `split` with parent `Vertex` called `splitParent`, if the type of `splitParent` is `merge` then a type II reduction can take place. We check for these conditions on line 12 of Algorithm 12.

The method for executing a type II reduction is shown in Algorithm 14. We now change the name of `splitParent` to `merge` for ease of understanding. We will use the the left parent `Strand` of `merge` to connect `lparentBegin` to `lchildEnd` (lines 2-5), and we will use the right parent `Strand` of `merge` to connect `rparentBegin` to `rchildEnd` (lines 6-9). Now we can delete `merge`, `split`, and the three unused `Strand`s (line 10). Lastly, we grab any adjacent `split` type `Vertex` objects and add them to `splitsToCheck` to ensure that we find any newly created reductions (lines 11-18).

Input: `one AbstractStrandDiagram` f
Output: f `after performing a type II reduction`

```
1  merge = splitParent
2  lparentStrand = merge.getStrand(lparent)
3  lchildStrand=split.getStrand(lchild)
4  leftEndRole = lchildStrand.getEndRole()
5  lchildEnd.addStrand(leftEndRole, lparentStrand)
6  rparentStrand = merge.getStrand(rparent)
7  rchildStrand=split.getStrand(rchild)
8  rightEndRole = rchildStrand.getEndRole()
9  rchildEnd.addStrand(rightEndRole, rparentStrand)
10 delete(merge,split,lchildStrand, rchildStrand,splitParentStrand)
11 if lparentBegin.type == split then
12 |  splitsToCheck.push(lparentBegin)
13 if rparentBegin.type == split then
14 |  splitsToCheck.push(rparentBegin)
15 if lchildEnd.type == split then
16 |  splitsToCheck.push(lchildEnd)
17 if rchildEnd.type == split then
18 |  splitsToCheck.push(rchildEnd)
```

Algorithm 14: Algorithm for `reductionII()` helper method, called on line 13 of Algorithm 12

Due to the absense of any repeated structures, Algorithm 14 runs in constant time.

Owing to the loop (while) on line 2, Algorithm 12 runs in $\mathcal{O}(n)$ time for n `Vertex` objects. Note that reduction of an `AbstractStrandDiagram` is more efficient than reduction of a `TreeDiagram` by an entire factor of n.

3
The Solution to the Conjugacy Problem

Given the strand diagram for an element of V we can **close** it. To do so, we remove the source vertex and the sink vertex, and connect the child vertex of the source to the parent vertex of the sink with a single connecting strand. This procedure is shown in Figure 3.0.1. The resulting structure is called a **closed abstract strand diagram**.

When we close an abstract strand diagram, we need a way to keep track of how it was closed. In Figure 3.0.1 we do so using a blue circle called a **puncture**. In order to make this concept rigorous, we use what is called a **cohomology class**. Defining cohomology classes will be the subject of Sections 3.1 and 3.2.

Definition 3.0.2. A **closed abstract strand diagram** is a finite directed topological graph with a rotation system and an associated cohomology class. Every vertex is either a split or a merge. Closed abstract strand diagrams can also contain dircted edges with no vertices called **free loops**.

We will describe free loops in detail in Section 3.3.

We reduce closed abstract strand diagrams in the same way that we reduce abstract strand diagrams. There are, however, two added complications. The first is that as we reduce we must continue to keep track of the cohomology class of the closed abstract strand diagram. The

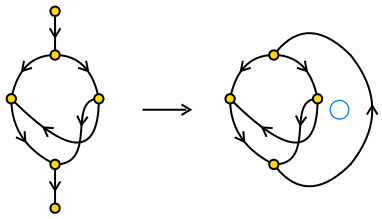


Figure 3.0.1. How to close an abstract strand diagram and obtain a closed abstract strand diagram

second is that we must add a third reduction move to address free loops when they arise. We will describe how to obtain the unique reduced form of a given closed abstract strand diagram in Section 3.3.

Belk and Matucci prove that two elements of V are conjugate if and only if their reduced closed abstract strand diagrams are isomorphic in a way that preserves their cohomology class [2]; we will describe this theorem in detail in Section 3.4.

Most of the content of this chapter comes from Belk and Matucci [2], but we have expanded upon it to include previously unexplained specifics, particularly concerning keeping track of the cohomology class as we reduce.

3.1 Cohomology on Directed Graphs

Definition 3.1.1. Let $G = (V_G, E_G, s_G, t_G)$ and $H = (V_H, E_H, s_H, t_H)$ be directed graphs. An **isomorphism** between G and H is a pair (ϕ_V, ϕ_E) where $\phi_V : V_G \to V_H$ and $\phi_E : E_G \to E_H$ are bijections satisfying the following conditions:

1. An edge $e \in E_G$ connects vertices $v, w \in V_G$ if and only if $\phi_E(e) \in E_H$ connects $\phi_V(v), \phi_V(w) \in V_H$

2. For any $e \in E_G$,

   - $s_H\big(\phi_E(e)\big) = \phi_V\big(s_G(e)\big)$
   - $t_H\big(\phi_E(e)\big) = \phi_V\big(t_G(e)\big)$

In other words, the source and target of each edge e on G must correspond to the source target of $\phi_E(e)$ on H, respectively.

For closed abstract strand diagrams in particular, we impose the additional restriction that the isomorphism must preserve the rotation system, i.e. for any edge $e \in E_G$ and connected vertex $v \in V_G$, the role of e at v must be equal to the role of $\phi_E(e)$ at $\phi_V(v)$. In other words, the bijection ϕ_E must preserve the role that each strand plays at each vertex.

We remark that for any vertex $v \in V_G$, the type of v will be equal to the type of $\phi_V(v)$; in other words, the bijection ϕ_V preserves the type of each vertex.

Example 3.1.2. Consider the closed abstract strand diagrams $G = (V_G, E_G)$ and $H = (V_H, E_H)$ shown in Figure 3.1.1, where $V_G = \{v_0, v_1, v_2, v_3\}$, $E_G = \{e_0, e_1, e_2, e_3, e_4, e_5\}$, $V_H = \{w_0, w_1, w_2, w_3\}$, and $E_H = \{e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$. The following bijections $\phi_V : V_G \to V_H$ and $\phi_E : E_G \to E_H$ form an isomorphism between G and H:

$$\begin{array}{lcl} \phi_V(v_0) & = & w_3 \\ \phi_V(v_1) & = & w_2 \\ \phi_V(v_2) & = & w_1 \\ \phi_V(v_3) & = & w_0 \end{array} \qquad \begin{array}{lcl} \phi_E(e_0) & = & d_1 \\ \phi_E(e_1) & = & d_0 \\ \phi_E(e_2) & = & d_3 \\ \phi_E(e_3) & = & d_2 \\ \phi_E(e_4) & = & d_5 \\ \phi_E(e_5) & = & d_4 \end{array}$$

Here is an isomorphism between G and H as directed graphs:

$$\begin{array}{lcl} \phi_V(v_0) & = & w_0 \\ \phi_V(v_1) & = & w_1 \\ \phi_V(v_2) & = & w_2 \\ \phi_V(v_3) & = & w_3 \end{array} \qquad \begin{array}{lcl} \phi_E(e_0) & = & d_0 \\ \phi_E(e_1) & = & d_1 \\ \phi_E(e_2) & = & d_2 \\ \phi_E(e_3) & = & d_3 \\ \phi_E(e_4) & = & d_4 \\ \phi_E(e_5) & = & d_5 \end{array}$$

The above is *not* an isomorphism of closed abstract strand diagrams G and H, since it does *not* preserve the rotation system.

We now turn to defining cohomology classes on directed graphs. We do so using cocycles, which we obtain by assigning integer values to edges. We can compare cocycles using cohomology, an equivalence relation. The equivalence classes under cohomology are called cohomology classes.

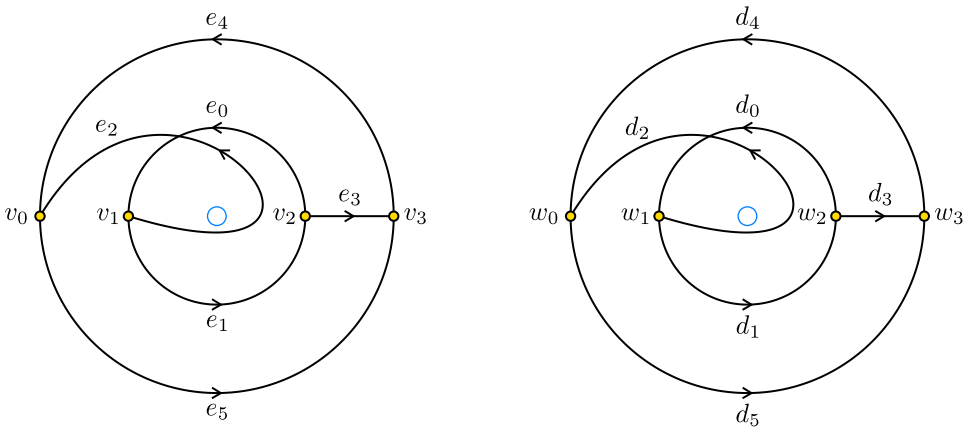


Figure 3.1.1. Two closed abstract strand diagrams which are isomorphic

Definition 3.1.3. Given a directed graph $G = (V, E)$, the vector space $\mathbb{R}^E$ is the set of all functions $f : E \to \mathbb{R}$. Elements of $\mathbb{R}^E$ are called **cocycles**.

We will write cocycles using the following three notations:

1. As described in Definition 3.1.3, we can write cocycles as a functions $f : E \to \mathbb{R}$.

2. We can describe $\mathbb{R}^E$ as the set of all formal linear combinations of edges, and thus may write cocycles as linear combinations of edges.

3. Given m edges, assign a number $\{0, 1, \ldots, m-1\}$ to each edge. We can then write cocycles as $m \times 1$ vectors where for $i \in \{0, 1, \ldots, m-1\}$ the entry at row i corresponds to the value of f at e_i.

Example 3.1.4. Consider a graph with edge set $E = \{e_0, e_1, e_2, e_3, e_4\}$. Consider the following element $b \in \mathbb{R}^E$:

$$\begin{array}{rcl} b(e_0) & = & 0 \\ b(e_1) & = & 2 \\ b(e_2) & = & 0 \\ b(e_3) & = & -4 \\ b(e_4) & = & 3 \end{array}$$

We can equivalently express b as the linear combination

$$b = 2e_1 - 4e_3 + 3e_4$$

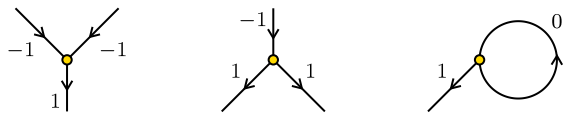


Figure 3.1.2. How to assign coefficients to edges in a coboundary of a vertex

or the vector

$$\vec{b} = \begin{bmatrix} 0 \\ 2 \\ 0 \\ -4 \\ 3 \end{bmatrix}$$

Definition 3.1.5. Given a directed graph $G = (V, E)$, the vector space $\mathbb{R}^V$ is the set of all functions $f : V \to \mathbb{R}$.

Our notations on $\mathbb{R}^V$ reflect our notations in $\mathbb{R}^E$. Note that V is a basis for $\mathbb{R}^V$.

Definition 3.1.6. Let $G = (V, E)$ be a directed graph with m vertices and n edges such that $V = \{v_0, v_1, \ldots, v_{m-1}\}$ and $E = \{e_0, e_1, \ldots, e_{n-1}\}$. We define the **coboundary** of a vertex v as $\vec{\delta v} = \lambda_0 e_0 + \lambda_1 e_1 + \cdots + \lambda_{n-1} e_{n-1}$ where

$$\lambda_i = \begin{cases} -1 & \text{if } e_i \text{ is an incoming edge to } v, \\ 1 & \text{if } e_i \text{ is an outgoing edge from } v, \\ 0 & \text{if } e_i \text{ is not connected to } v \text{ or forms a loop at } v, \end{cases}$$

as shown in Figure 3.1.2. Let $b \in \mathbb{R}^V$ be a formal linear combination of vertices, say $b = \alpha_0 v_0 + \alpha_1 v_1 + \cdots + \alpha_{m-1} v_{m-1}$ for $\alpha_j \in \mathbb{R}$. We define the **coboundary** of b as

$$\vec{\delta b} = \alpha_0 \vec{\delta v_0} + \alpha_1 \vec{\delta v_1} + \cdots \alpha_{m-1} \vec{\delta v}_{m-1}$$

We define the **coboundary map** as the linear function $\delta : \mathbb{R}^V \to \mathbb{R}^E$ which satisifies the above conditions for all $b \in \mathbb{R}^V$.

A cocycle $\vec{a} \in \mathbb{R}^E$ is called a **coboundary** if $\vec{a} = \vec{\delta b}$ for some $b \in \mathbb{R}^V$.

Example 3.1.7. Consider the left closed abstract strand diagram in Figure 3.1.3. The coboundary of v_2 is

$$\vec{\delta v_2} = e_0 - e_1 + e_3$$

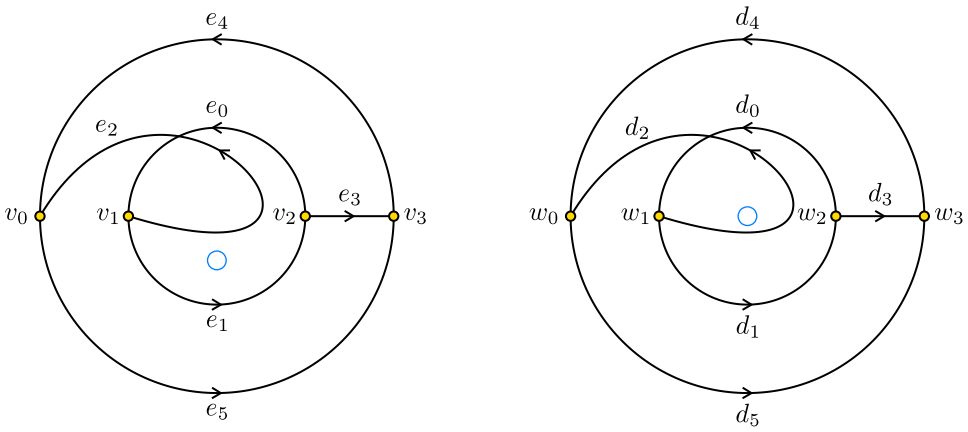


Figure 3.1.3. The two closed abstract strand diagrams above are isomorphic but not cohomologous

We can also depict $\delta\vec{v}_2$ as a matrix where row i corresponds to λ_i. Thus

$$\delta\vec{v}_2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Definition 3.1.8. For a given coboundary map δ and two cocycles $\vec{a}$ and $\vec{b}$, we say that $\vec{a}$ and $\vec{b}$ are **cohomologous** if $\vec{a} - \vec{b}$ is a coboundary.

We note that cohomology is an equivalence relation. The corresponding equivalence classes are called **cohomology classes**.

When we draw a directed graph on the plane, we can indicate a cohomology class by drawing a puncture. Given a puncture, you can draw a path from the puncture to infinity to obtain a cocycle of that graph in the following manner.

The coefficient of an edge e which the path does not cross is 0. If the path croses an edge e in a counterclockwise direction, then it's coefficient is 1. If the path crosses e in a clockwise direction, then then its coefficient is -1. If the path crosses e multiple times, we add or subtract 1 the corresponding number of times. Figure 3.1.4 shows these cases. Any two cocyles obtained this way are cohomologous.

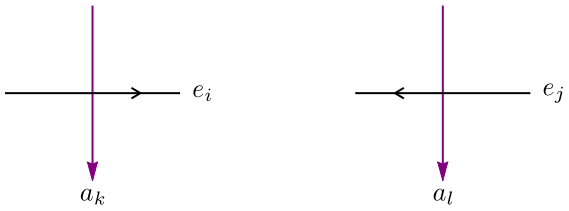


Figure 3.1.4. The edge e_i crosses the path a_k in a counterclockwise direction, so we would add 1 to the coefficient of e_i (left). The edge e_j crosses the path a_l in a clockwise direction, so we would subtract 1 from the coefficient of e_j (right).

Example 3.1.9. Figure 3.1.5 shows two paths labeled $\vec{b_1}$ and $\vec{b_2}$. The path $\vec{b_1}$ represents the cocycle $e_1 + e_5$ and the path $\vec{b_2}$ represents the cocycle $e_0 + e_3 + e_5$. Alternatively, we have

$$\vec{b_1} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \vec{b_2} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Notice that

$$\vec{b_2} - \vec{b_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

which is precisely the coboundary $\delta\vec{v}_2$. In general, dragging a cocycle path across one vertex changes the cocycle by the coboundary of that vertex. Since $\vec{b_2} - \vec{b_1} = \delta\vec{v}_2$ and $\delta\vec{v}_2$ is a coboundary, we conclude that $\vec{b_1}$ and $\vec{b_2}$ belong to the same cohomology class (they are cohomologous).

3.2 The Coboundary Matrix

Definition 3.2.1. Given a directed graph as described in Definition 3.1.6, we can construct an $n \times m$ matrix as follows:

$$\begin{bmatrix} \delta\vec{v}_1 & \delta\vec{v}_2 & \dots & \delta\vec{v}_m \end{bmatrix}$$

We call this matrix the **coboundary matrix**.

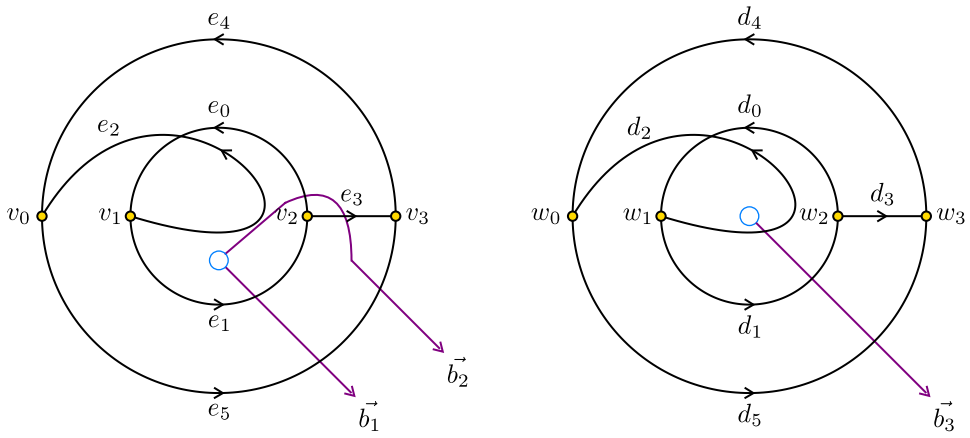


Figure 3.1.5. Two directed graph with associated cohomology classes indicated punctures, as well as cocycles $\vec{b_1}$ and $\vec{b_2}$ (left) and $\vec{b_3}$ (right).

Example 3.2.2. Consider the graph at left in Figure 3.1.5. The coboundary matrix is

$$\begin{bmatrix} 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$

Note that the graph at right in Figure 3.1.5 has the same coboundary matrix. This is true because they are isomorphic graphs.

Theorem 3.2.3. *Let G and H be two isomorphic directed graphs with coboundary matrix M. Let $\vec{a}$ be a cocycle from G and $\vec{b}$ be a cocycle from H. Then $\vec{a}$ and $\vec{b}$ are cohomologous if and only if the vector $\vec{a} - \vec{b}$ is a linear combination of the columns in M.*

This follows immediately from the fact that the range of a linear transformation is the column space of the associated matrix. In particular, the column space of the coboundary matrix is precisely the set of cocycles that are coboundaries.

Corollary 3.2.4. *Given coboundary matrix M, the cocycles $\vec{a}$ and $\vec{b}$ are cohomologous if and only if the augmented matrix*

$$\left[M \quad | \quad \vec{a} - \vec{b} \right]$$

has the same rank as the original matrix M.

Example 3.2.5. Consider the two isomorphic closed abstract strand diagrams in G (left) and H (right) in Figure 3.1.5. Note that their coboundary matrix is equal to that in Example 3.2.2. Observe the two cocycles

$$\vec{b_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \vec{b_3} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Then

$$\vec{b_1} - \vec{b_3} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

We compute that the rank of the augmented matrix

$$\left[\begin{array}{cccc|c} 0 & 1 & -1 & 0 & 1 \\ 0 & -1 & 1 & 0 & -1 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 1 \\ 1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 1 & 0 \end{array}\right]$$

is 4, whereas the rank of the original coboundary matrix is 3. Thus $\vec{b_1}$ and $\vec{b_3}$ are not cohomologous.

We say that two closed abstract strand diagrams are **equivalent** if their graphs are isomorphic and they belong to the same cohomology class. Example 3.2.5 exhibits the importance of the cohomology class associated with each closed abstract strand diagram. Even though they are isomorphic, we do not consider them equivalent since their cocycles are not cohomologous.

Example 3.2.6. Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be the directed graphs at left and right respectively in Figure 3.2.1. The following bijections $f_1 : V_G \to V_H$ and $g_1 : E_G \to E_H$ form an

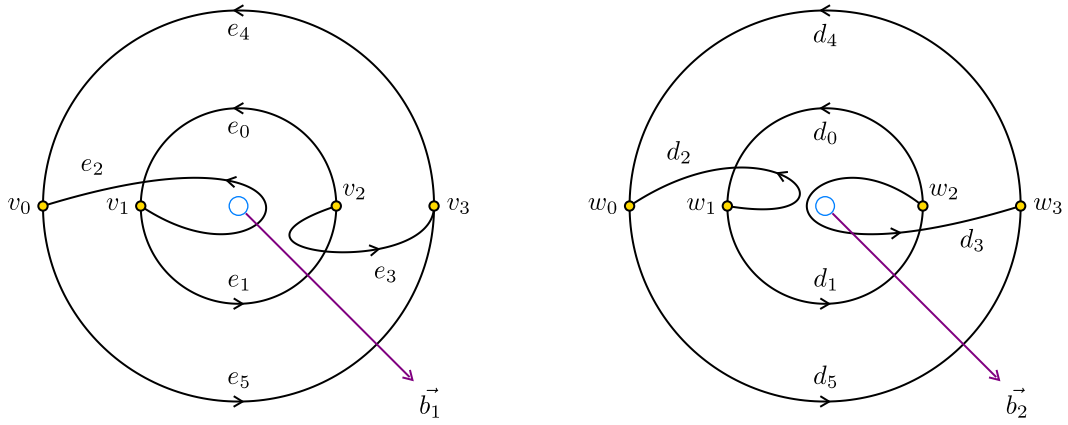


Figure 3.2.1. Two closed abstract strand diagrams with more than isomorphism between them

isomorphism between G and H:

$$\begin{array}{lcl} f_1(v_0) & = & w_0 \\ f_1(v_1) & = & w_1 \\ f_1(v_2) & = & w_2 \\ f_1(v_3) & = & w_3 \end{array} \qquad \begin{array}{lcl} g_1(e_0) & = & d_0 \\ g_1(e_1) & = & d_1 \\ g_1(e_2) & = & d_2 \\ g_1(e_3) & = & d_3 \\ g_1(e_4) & = & d_4 \\ g_1(e_5) & = & d_5 \end{array}$$

We construct the coboundary matrix

$$\begin{bmatrix} 0 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$

which has rank 3. We then compute that

$$\vec{b_1} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \vec{b_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\vec{b_1} - \vec{b_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

The augmented matrix

$$\left[\begin{array}{cccc|c} 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & -1 \\ -1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -1 & 0 \end{array}\right]$$

has rank 4, which would lead us to conclude that $\vec{b_1}$ and $\vec{b_2}$ are not cohomologous. However, the following bijections $f_2 : V_G \to V_H$ and $g_2 : E_G \to E_H$ form another isomorphism between G and H:

$$\begin{array}{lcl} f_2(v_0) = w_3 \\ f_2(v_1) = w_2 \\ f_2(v_2) = w_1 \\ f_2(v_3) = w_0 \end{array} \qquad \begin{array}{lcl} g_2(e_0) = d_1 \\ g_2(e_1) = d_0 \\ g_2(e_2) = d_3 \\ g_2(e_3) = d_2 \\ g_2(e_4) = d_5 \\ g_2(e_5) = d_4 \end{array}$$

We have the same coboundary matrix but must modify $\vec{b_2}$ to correspond with the current isomorphism. The edge d_1 now corresponds to row 0, the edge d_0 corresponds to row 1, etc. so

$$\vec{b_2} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

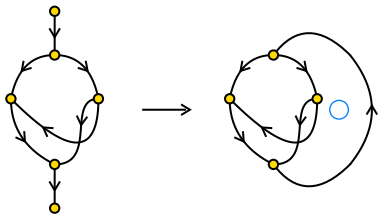
which gives us

$$\vec{b_1} - \vec{b_2} = \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

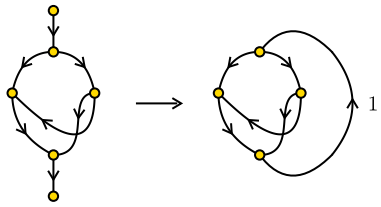
and the augmented matrix

$$\left[\begin{array}{cccc|c} 0 & -1 & 1 & 0 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & -1 & 1 \end{array}\right]$$

which has rank 3. Since the rank of the coboundary matrix is equal to the rank of the augmented matrix above, we conclude that $\vec{b_1}$ and $\vec{b_2}$ are indeed cohomologous.



(a) A closed abstract strand diagram with the cohomology class indicated by a puncture



(b) A closed abstract strand diagram with the cohomology class indicated by c-values

Figure 3.3.1. Two ways to close an abstract strand diagram and represent a cohomology class

Example 3.2.6 shows two closed abstract strand diagrams G and H, with cocycles $\vec{a}$ and $\vec{b}$ respectively, may have multiple isomorphisms between them, some of which preserve the cohomology class of $\vec{a}$ and $\vec{b}$ and some of which do not.

3.3 Reduction of Closed Abstract Strand Diagrams

The reduction moves on closed abstract strand diagrams are the same as the moves for regular abstract strand diagrams with two added complications. The first is that as we reduce we must continue to keep track of the associated cohomology class.

Recall that when we close an abstract strand diagram, we keep track of how we close it using a puncture. Another way to do this is by assigning an integer value 1 to the newly drawn edge; both of these methods are shown in Figure 3.3.1. We call this value the **cutting value** or **c-value** for that strand. The c-value for every other strand is initialized at 0, and these values together define the cocycle for the closed abstract strand diagram. Figure 3.3.2 shows two equivalent drawings of a closed abstract strand diagram with cocycle

$$\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

As we reduce and the edges of closed abstract strand diagram change, we must reassign the c-values of each edge accordingly. The rules for this reassignment after each reduction move follow from considering a path that cuts through each edge the indicated number of times. The importance of these rules is to continue to keep track of the cohomology class accurately as we reduce.

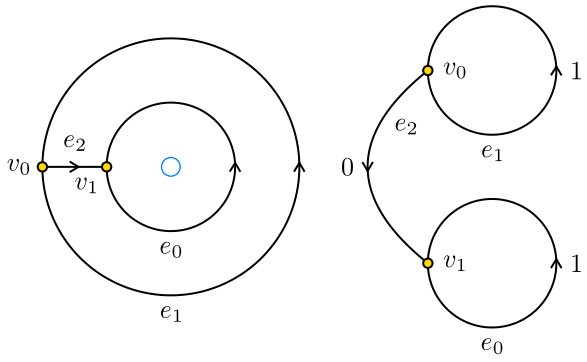


Figure 3.3.2. Two equivalent representations of the same close abstract strand diagram

The second complication is that as we reduce a closed abstract strand diagram, free loops can arise.

Definition 3.3.1. A **free loop** is an edge which forms a loop within a closed abstract strand diagram and contains no vertices.

We will describe how free loops can arise from the vertex deletions in type I and type II reduction moves.

A **type I** reduction can only occur in a closed abstract strand diagram if the c-values for the left child and right child strands of the split vertex equal. Following a type I reduction move, the c-values of the remaining strands must be reassigned according to the values of the deleted strands. This reduction move is shown in Figure 3.3.3a.

We must consider one special case of a type I reduction, shown in Figure 3.3.3b. If the parent vertex of the split is equal to the merge vertex at the end of its left child and right child strands, then the deletion of these two vertices leads to a free loop. The c-value of the free loop is $i + j$.

A **Type II** reduction proceeds in exactly the same way in closed abstract strand diagrams as it does in regular abstract strand diagrams. Following a type II reduction, however, the c-values of the remaining strands must be reassigned according to the values of the deleted strands. A typical case for this reduction move is shown in Figure 3.3.4.

There are five special cases that we must consider which differ from the typical case.

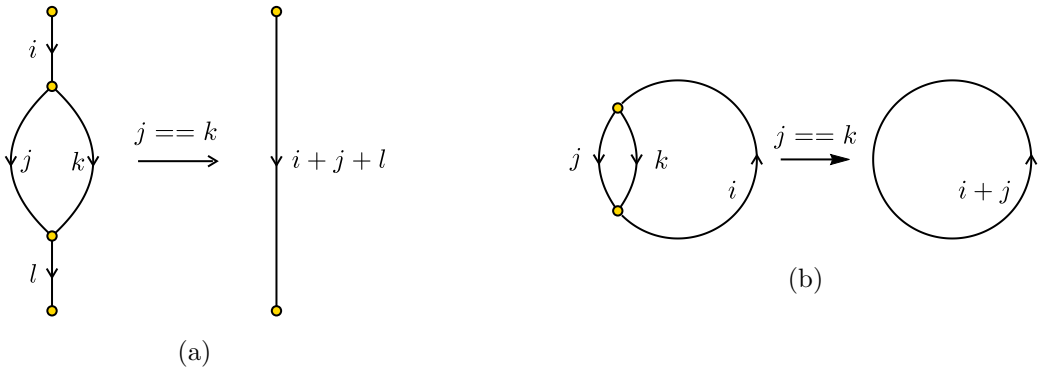


Figure 3.3.3. The two cases in which a type I reduction can occur

1. The right parent of the merge is equal to the right child of the split: This case is shown in Figure 3.3.5a .

2. The left parent of the merge is equal to the left child of the split: This case is shown in Figure 3.3.5b. Note that this case and the case described in item 1 are not mutually exclusive and may both occur in the same reduction.

3. The right parent of the merge is equal to the left child of the split: This case is shown in Figure 3.3.6a.

4. The left parent of the merge is equal to the right child of the split: This case is shown in Figure 3.3.6b.

5. The right parent of the merge is equal to the left child of the split *and* the left parent of the merge is equal to the right child of the split: This case is shown in Figure 3.3.7a.

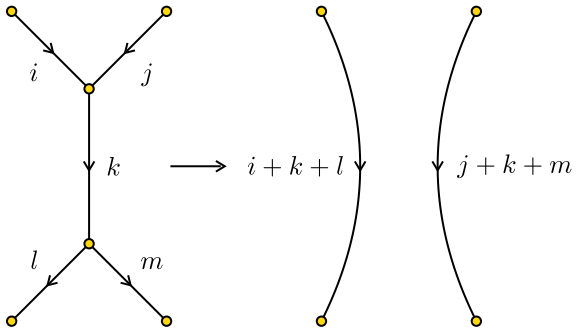


Figure 3.3.4. One case in which a type II reduction can occur

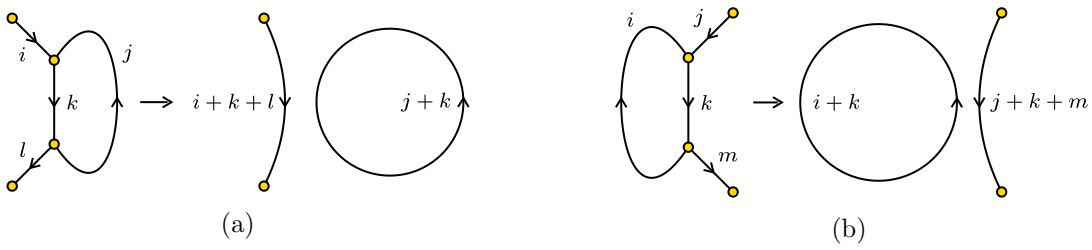


Figure 3.3.5. Two special cases of a type II reduction

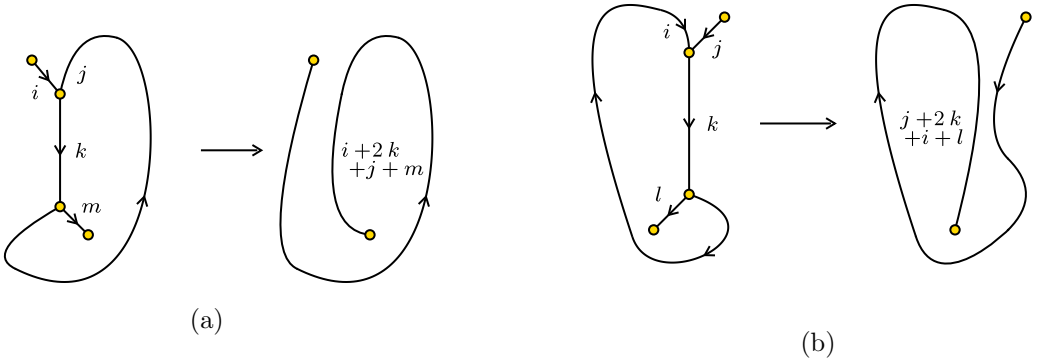


Figure 3.3.6. Two special cases of a type II reduction

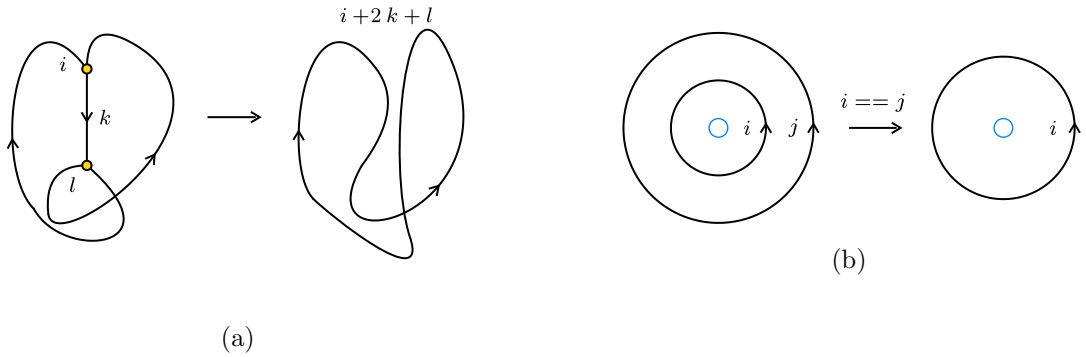


Figure 3.3.7. A special case of the type II reduction (left) and the type III reduction (right)

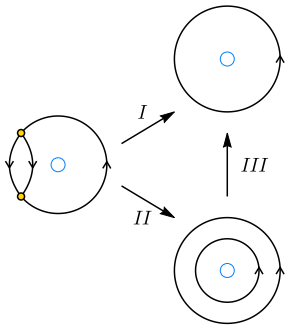


Figure 3.3.8. A closed abstract strand diagram with multiple possible reduction moves

These rules come from considering a path that cuts through each edge the indicated number of times.

We remark that in a closed abstract strand diagram, a type II reduction can result in multiple connected components within the diagram. Each component is itself a closed abstract strand diagram.

The third move, a **Type III** reduction, combines two free loops which have the same c-value into just one loop with the same value. This reduction is shown in Figure 3.3.7b. Sometimes there are multiple possible reduction moves to perform on a given closed abstract strand diagram, which can result in two different graphs. The type III reduction move ensures that despite this, closed abstract strand diagrams each have a unique reduced form. One example of this is shown in Figure 3.3.8

A closed abstract strand diagram is **reduced** if no reduction moves can be done. Belk and Matucci prove that each closed abstract strand diagram has a unique reduced form [2].

3.4 Conjugacy in Thompson's Group V

The motivation for the solution to the conjugacy problem on V comes from the solution to the conjugacy problem on the free group.

Definition 3.4.1. An infinite group G is called **free** if it has generating set S with no relations.

Every element in G can be written uniquely as a reduced word over S.

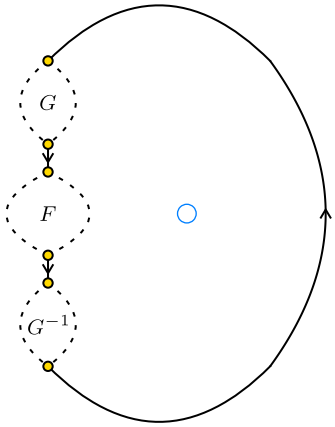


Figure 3.4.1. If F and G are closed strand diagrams, then if we reduce the closed strand diagram above we will obtain F

Definition 3.4.2. A word is **cyclically reduced** if it is reduced and the first and last elements are not inverses of each other.

Theorem 3.4.3. *Let α, β be words over generating set S for free group G. If we cyclically reduce α and obtain β then α and β are conjugate* [10].

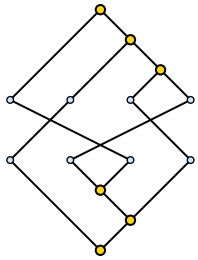
Example 3.4.4. Consider the following free group G with generating set $S = \{x, y, z\}$ and elements $\alpha = xyz^{-1}xy^{-1}$ and $\beta = x^{-1}zxyz^{-1}xy^{-1}z^{-1}x$. Observe that we can cyclically reduce β in the following manner:

$$x^{-1}(zxyz^{-1}xy^{-1}z^{-1})x \;\rightarrow\; z(xyz^{-1}xy^{-1})z^{-1} \;\rightarrow\; xyz^{-1}xy^{-1}$$

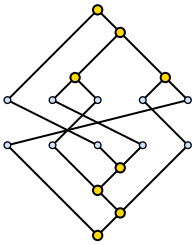
which is precisely α. We conclude that α and β are conjugate.

Theorem 3.4.5 (Belk, Matucci)**.** *Two elements of V are conjugate if and only if their reduced closed abstract strand diagrams are equivalent, i.e. they are isomorphic in a way that preserves their cohomology class* [2].

The idea of cyclically reducing manifests in closed abstract strand diagrams as shown in Figure 3.4.1. For a full proof of Theorem 3.4.5, the reader is referred to Belk and Matucci [2].



(a) Tree diagram X



(b) Tree diagram Y

Figure 3.4.2.

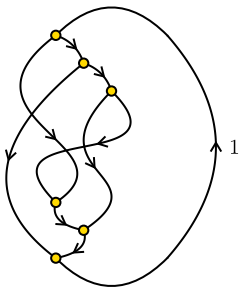
We will now work out a full example of determining the conjugacy of two elements of V from start to finish.

Example 3.4.6. Let $X \in V$ be the element of V defined by domain code $[0, 10, 110, 111]$, range code $[0, 100, 101, 11]$, and permutation code $[2, 0, 3, 1]$. Let $Y \in V$ be the element of V definied by domain code $[0, 100, 101, 110, 111]$, range code $[0, 100, 1010, 1011, 11]$, and permutation code $[2, 3, 1, 4, 0]$. The tree diagrams for X and Y are shown in Figure 3.4.2.

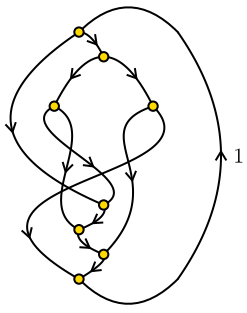
We can convert X and Y to strand diagrams and close them to obtain closed abstract strand diagrams. The result of this procedure is shown in Figure 3.4.3.

After reducing X and Y we obtain the reduced closed abstract strand diagrams shown in Figure 3.4.4.

We now have reduced closed abstract strand diagram $X = (V_X, E_X)$ such that $V_X = \{v_0, v_1\}$ and $E_X = \{e_0, e_1, e_2\}$; and $Y = (V_Y, E_Y)$ such that $V_Y = \{w_0, w_1\}$ and $E_Y = \{d_0, d_1, d_2\}$. We

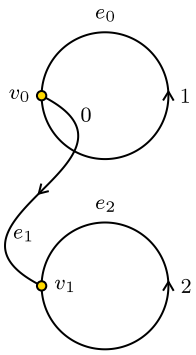


(a) Closed abstract strand diagram X

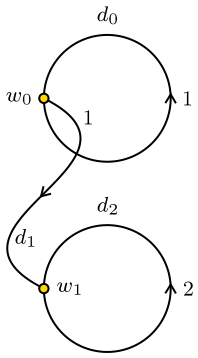


(b) Closed abstract strand diagram Y

Figure 3.4.3.



(a) Reduced closed abstract strand diagram X



(b) Reduced closed abstract strand diagram Y

Figure 3.4.4.

can define the following isomorphism $\phi_V : V_X \to V_Y$ and $\phi_E : E_X \to E_Y$:

$$\begin{array}{lcl} \phi_V(v_0) & = & w_0 \\ \phi_V(v_1) & = & w_1 \end{array} \qquad \begin{array}{lcl} \phi_E(e_0) & = & d_0 \\ \phi_E(e_1) & = & d_1 \\ \phi_E(e_2) & = & d_2 \end{array}$$

And we can construct the following coboundary matrix M, where for $i \in \{0,1,2\}$ row i corresponds to e_i and d_i, and for $j \in \{0,1\}$ column j corresponds to v_j and w_j:

$$M = \begin{bmatrix} 1-1 & 0 \\ 1 & -1 \\ 0 & 1-1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{bmatrix}$$

The rank of this matrix is 1.

We obtain a cocycle $\vec{a}$ from X by constructing a $3 \times n$ vector where for $i \in \{0,1,2\}$ the entry at $a[i,0]$ corresponds to the c-value of e_i. Similarly, we obtain a cocycle $\vec{b}$ from Y by constructing a $3 \times n$ vector where for $i \in \{0,1,2\}$ the entry at $b[i,0]$ corresponds to the c-value of d_i. Then we have

$$\vec{a} = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

and compute the difference

$$\vec{a} - \vec{b} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$$

The augmented matrix $\left[M | \vec{a} - \vec{b}\right]$ is the following:

$$\left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & -1 & -1 \\ 0 & 0 & 0 \end{array}\right]$$

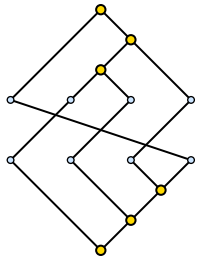


Figure 3.4.5. The conjugator for X and Y

which also has rank 1. We conclude that our original elements X and Y are conjugate.

Our algorithm determines whether two elements of V are conjugate; it does *not* produce the conjugator. However, we remark that the conjugator for these two elements is the element of V defined by domain code $[0, 100, 101, 11]$, range code $[0, 10, 110, 111]$, and permutation code $[3, 0, 1, 2]$; it is shown in Figure 3.4.5.

4
Implementing the Solution to the Conjugacy Problem

We have described our implementation of the `TreeDiagram` class and the process of converting them to `AbstractStrandDiagram`s. We must now close these strand diagrams, reduce the resulting closed abstract strand diagrams, and decide whether they are equivalent or not. Recall that two closed abstract strand diagrams are equivalent if they are isomorphic and their cocycles belong to the same cohomology class.

We will discusss how to close an abstract strand diagram and describe our implementation of the `ClosedAbstractStrandDiagram` class. Then we will describe how we check isomorphism between `ClosedAbstractStrandDiagram`s, obtain their (shared) coboundary matrix, calculate the difference in their cocycles, and determine whether the cocycle difference is in the column space of the coboundary matrix. If these checks pass then we conclude that our original elements of V are conjugate. The implementation of this process will be the subject of this chapter.

4.1 Implementing Closed Abstract Strand Diagrams

Our `ClosedAbsractStrandDiagram` class is based upon our regular `AbstractStrandDiagram` class, with additions and modifications as needed. We use the same `Vertex` and `Strand` classes, and store these objects in the same `DoublyLinkedList` structure.

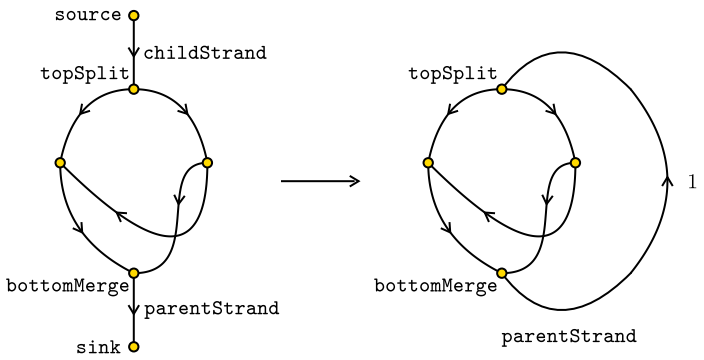


Figure 4.1.1. A visual explanation of the `close()` method on `AbstractStrandDiagram` objects

Recall that in addition to vertices and strands, closed abstract strand diagrams can also contain free loops that arise from deleted vertices during reductions. Additionally, the type III reduction move stipulates that if two free loops have the same c-value then they can be combined into one single loop. We therefore use a `Set` object of integer free loops to store the c-value for each free loop in a `ClosedAbstractStrandDiagram`. The `Set` does not allow for duplicate values; in other words, if we insert an integer x into the `Set` but the `Set` already contains x, then the `Set` will not change. Thus, the third reduction move is taken care of.

To construct a `ClosedAbstractStrandDiagram`, we can call the `close()` method on a regular abstract strand diagram, which esssentially updates the connections and c-values of the `AbstractStrandDiagram`. We outline this procedure in Algorithm 15, and include Figure 4.1.1 with corresponding variable names.

On lines 1 and 2 we access the `source` type `Vertex` and `sink` type `Vertex` of the `AbstractStrandDiagram` f. We need to access the topmost `split` type `Vertex` of f, which we call `topSplit` (lines 3-4). We attach the `Strand` playing the role `parent` at `sink` (aptly named `parentStrand` on line 5) to `topSplit` and delete the now unused `sink`, `source`, and `childStrand` (lines 6-7). Finally, we initialize an empty `Set` called `freeLoops` which we will use to store free loops when they arise (line 9).

Due to the absence of any repeat structures in Algorithm 15, we conclude that closing an `AbstractStrandDiagram` takes constant time.

Input: an `AbstractStrandDiagram` f
Output: the `ClosedAbstractStrandDiagram` for f

```
1  source = f.source
2  sink = f.sink
3  childStrand = source.getStrand(child)
4  topSplit = childStrand.getEndVertex()
5  parentStrand = sink.getStrand(parent)
6  topSplit.addStrand(parent, parentStrand)
7  delete(source, sink, childStrand)
8  parentStrand.incrementCValue()
9  Set<Integer> freeLoops = emptySet()
10 result = ClosedAbstractStrandDiagram(f.vertices, freeLoops)
11 return result
```

Algorithm 15: Algorithm to convert an `AbstractStrandDiagram` to a `ClosedAbstractStrandDiagram`

4.1.1 Implementing Reduction

The algorithm to reduce `ClosedAstractStrandDiagram`s is based that of of regular `AbstractStrandDiagram`s, with some added modifications to address free loops and maintaining the associated cohomology class. In particular, we must account for the special cases detailed in Section 3.3. We present and desribe the updated algorithms below.

We present our reduction method in Algorithm 16. The only difference between Algorithm 12 and Algorithm 16 (the reduction method for regular `AbstractStrandDiagram`s) is on line 11, which contains the additional check for a type I reduction that the c-value for the right child strand of `split` is equal to the c-value for the left child strand of `split`. This check corresponds to the check that $j == k$ in Figure 4.1.2.

The method for executing a type I reduction is shown in Algorithm 17. One difference between Algorithm 17 and Algorithm 13 (the type I reduction method for regular `AbstractStrandDiagram`s) occurs on lines 3-5. This is where we check for the special case that `splitParent` is equal to `mergeChild`. If so, we compute the c-value for the resulting free loop, which we call *loop*, and add it to our `Set` called `freeLoops`. The other difference occurs on lines 10-11. This is where we compute the c-value for the updated `Strand` connecting `splitParent` to `mergeChild`; see Figure 4.1.2 for details.

Input: one `ClosedAbstractStrandDiagram` f
Output: f `in reduced form`

```
1  splitsToCheck = getSplitsToCheck()
2  while splitsToCheck.isNotEmpty() do
3  |  split = splitsToCheck.pop()
4  |  splitParent = split.getStrand(parent).getBeginVertex()
5  |  lchildEnd = split.getStrand(lchild).getEndVertex()
6  |  rchildEnd = split.getStrand(rchild).getEndVertex()
7  |  lchildEndRole = split.getStrand(lchild).getEndRole()
8  |  rchildEndRole = split.getStrand(rchild).getEndRole()
9  |  j=split.getStrand(lchild).getCValue()
10 |  k=split.getStrand(rchild).getCValue()
11 |  if lchildEnd == rchildEnd && lchildEndRole == lparent && rchildEndRole ==
   |  rparent && j == k then
12 |  |  reductionI() // see Algorithm 17
13 |  |  continue
14 |  else if splitParent.type == merge then
15 |  |  reductionII() // see Algorithm 18
16 |  |  continue
17 end
```

Algorithm 16: Algorithm to reduce a `ClosedAbstractStrandDiagram`

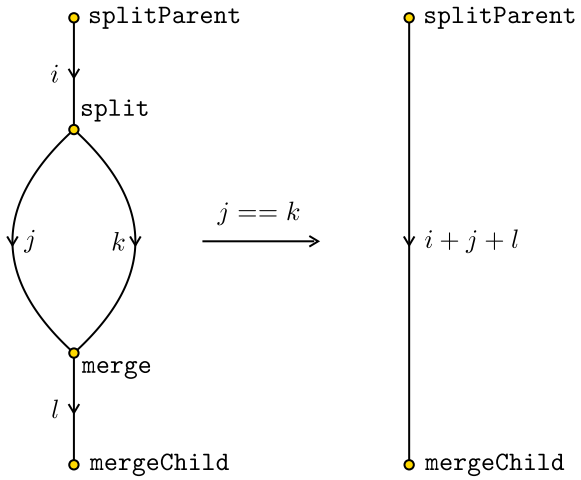


Figure 4.1.2. Requirements, procedure, and variable names for a type I reduction to occur

The only operation in Algorithm 17 which does not run in constant time is insertion into the `Set freeLoops`. This takes linear time, so Algorithm 17 runs in $\mathcal{O}(n)$ time, for number of vertices n.

Input: `one ClosedAbstractStrandDiagram` f
Output: f `after performing a type I reduction`

```
1  splitParentStrand = split.getStrand(parent)
2  mergeChildStrand = merge.getStrand(child)
3  if mergeChildStrand == splitParentStrand then
4  |  loop = i + j
5  |  freeLoops.add(loop)
6  else
7  |  mergeChild = mergeChildStrand.getEndVertex()
8  |  endRole = mergeChildStrand.getEndRole()
9  |  mergeChild.addStrand(endRole, splitParentStrand)
10 |  newC = i + j + l
11 |  splitParentStrand.updateCValue(newC)
12 delete(split,merge,lchildStrand, rchildStrand,mergeChildStrand)
13 if splitParent.type == split then
14 |  splitsToCheck.push(splitParent)
15 if mergeChild.type == split then
16 |  splitsToCheck.push(mergeChild)
```

Algorithm 17: Algorithm for `reductionI()` helper method in a `ClosedAbstractStrandDiagram`, called on line 12 of Algorithm 16

The method for executing a type II reduction is shown in Algorithm 18, which is significantly longer than Algorithm 12 (the type II reduction method for regular `AbstractStrandDiagrams`). This is due to the five special cases that can occur in a closed abstract strand diagram type II reduction move. Line 8 is the case that the right parent of `merge` is equal to the right child of `split` (see Figure 3.3.5a). Line 11 is the case that the left parent of `merge` is equal to the left child of `split` (see Figure 3.3.5b). Line 14 is the case that the right parent of `merge` is equal to the left child of `split` *and* the left parent of `merge` is equal to the right child of the split (see Figure 3.3.7a). Line 17 is the case that the right parent of `merge` is equal to the left child of `split` *only* (see Figure 3.3.6a). Line 21 is the case that the left parent of `merge` is equal to the right child of `split` *only* (see Figure 3.3.6b).

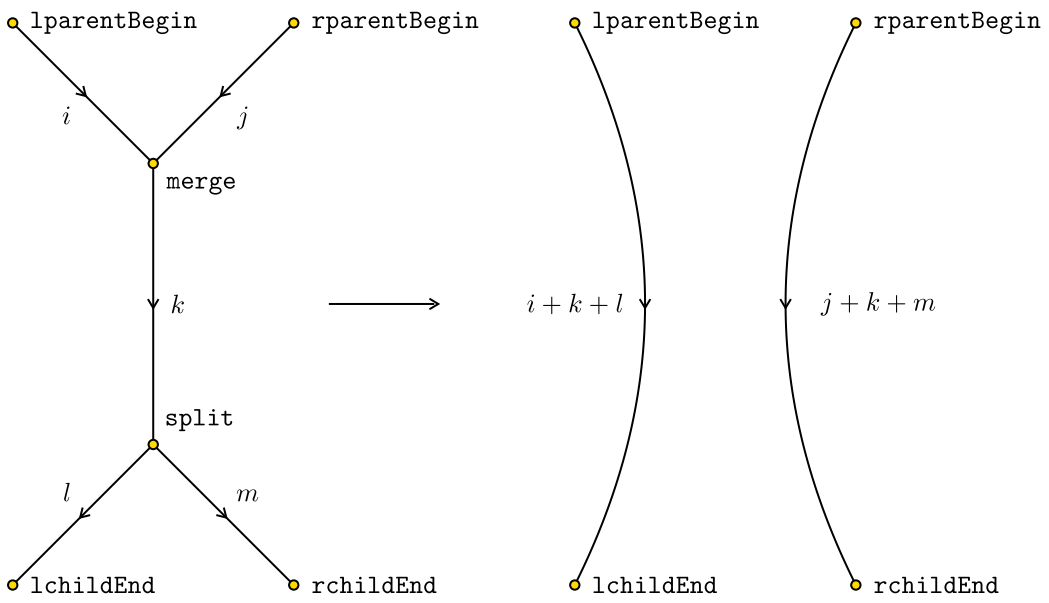


Figure 4.1.3. Requirements, procedure, and variable names for a type II reduction to occur

The last difference between Algorithm 16 and Algorithm 12 is on lines 26-29, which computes c-values for the updated `Strand` connecting `lparentBegin` and `lchildEnd` as well as the `Strand` connecting `rparentBegin` and `rchildEnd`; see Figure 4.1.3 for details.

Similarly to Algorithm 17, this algorithm runs in $\mathcal{O}(n)$ time.

Due to the single loop (while) on line 2 of Algorithm 16 and linear runtime of helper methods `reductionI()` and `reductionII()`, we conclude that our implementation of reduction of `ClosedAbstractStrandDiagrams` is quadratic on the number of vertices in the diagram.

We remark that it would be possible to implement reduction in $\mathcal{O}(n \log n)$ time if we stored free loops in a different way. Instead of using a `Set`, we could use an `Array`, since inserting an integer into an `Array` takes constant time. At the conclusion of the loop on line 2, we would sort this list and remove duplicates. This takes $\mathcal{O}(n \log n)$ time on the length of the list, which is bound by the nunmber of vertices in the diagram.

4.2 Preliminary Checks

Before makeing a direct comparison between the `Vertex` and `Strand` objects in two `ClosedAbstractStrandDiagrams`, we include a variety of constant, linear, and linearithmic time

Input: one `ClosedAbstractStrandDiagram` f
Output: f `after performing a type II reduction`

```
1  merge = splitParent
2  lparentStrand = merge.getStrand(lparent)
3  rparentStrand = merge.getStrand(rparent)
4  lparentBegin = lparentStrand.getBeginVertex()
5  rparentBegin = rparentStrand.getBeginVertex()
6  leftEndRole = lchildStrand.getEndRole()
7  rightEndRole = rchildStrand.getEndRole()
8  if rparentStrand == rchildStrand then
9  |  loop = j + k
10 |  freeLoops.add(loop)
11 if lparentStrand == lchildStrand then
12 |  loop = i + k
13 |  freeLoops.add(loop)
14 else if lparentStrand == rchildStrand && rparentStrand == lchildStrand then
15 |  loop = i + 2k + j
16 |  freeLoops.add(loop
17 else if rparentStrand == lchildStrand then
18 |  c = i + 2k + l + m
19 |  lparentStrand.updateCValue(c)
20 |  rchildEnd.addStrand(rightEndRole, lparentStrand)
21 else if lparentStrand == rchildStrand then
22 |  c = j + 2k + l + m
23 |  rparentStrand.updateCValue(c)
24 |  lchildEnd.addStrand(leftEndRole,rparentStrand
25 else
26 |  cLeft = i + k + l
27 |  cRight = j + k + m
28 |  lparentStrand.updateCValue(cLeft)
29 |  rparentStrand.updateCValue(cRight)
30 |  lchildEnd.addStrand(leftEndRole, lparentStrand)
31 |  rchildEnd.addStrand(rightEndRole, rparentStrand)
32 delete(merge,split,lchildStrand, rchildStrand,splitParentStrand)
33 if lparentBegin.type == split then
34 |  splitsToCheck.push(lparentBegin)
35 if rparentBegin.type == split then
36 |  splitsToCheck.push(rparentBegin)
37 if lchildEnd.type == split then
38 |  splitsToCheck.push(lchildEnd)
39 if rchildEnd.type == split then
40 |  splitsToCheck.push(rchildEnd)
```

Algorithm 18: Algorithm for `reductionII()` helper method in a `ClosedAbstractStrandDiagram`, called on line 15 of Algorithm 16

checks to quickly disqualify the possibility that they could be isomorphic. Since our final runtime is cubic, performing these checks greatly improves the average runtime of our program.

1. **Compare number of vertices**

   If two closed abstract strand diagrams A and B do not have the same number of vertices, then they cannot possibly be isomorphic. Each `ClosedAbstractStrandDiagram` stores its most recently updated size as member data so we can perform this check in constant time.

2. **Compare free loops sets**

   If A and B do not have identical free loop `Set`s then they cannot be isomorphic. As explained in Section 4.1, free loops are stored in `Set`s of integers, and the number of possible free loops is bound linearly by the number of vertices. Thus the equality comparison between these two `Set`s is $\mathcal{O}(n \log n)$.

3. **Compare number of components**

   If A and B do not have the same number of components then they cannot be isomorphic. The procedure to separate a `ClosedAbstractStrandDiagram` into its connected components is outlined Algorithm 19 and proceeds as follows:

   (a) Loop through each `Vertex` v in the `ClosedAbstractStrandDiagram`; this loop is declared on line 3.

   (b) If v is not "found," mark it "found" and add it to `VertexStack`, a `Stack` of vertices to process; this occurs on lines 4-6.

   (c) Crawl through the diagram outwards from v. Mark every encountered `Vertex` as "found" and add it to a `DoublyLinkedList` called `newComponent`; this happens on lines 8-15.

   (d) Once all vertices connected to v are "found," create a new `ClosedAbstractStrandDiagram` from `newComponent`, which now contains ever `Vertex` in the current component; this occurs on line 18.

Now we have $A = \{A_1, A_2, \ldots, A_k\}$ and $B = \{B_1, B_2, \ldots, B_l\}$, where A has k components and B has l components, and each A_i and B_j represents one such component. If $k == l$ then A and B have the same number of components and we can proceed to the next check.

We check to see whether each vertex is "found" twice. Then the time to separate a `ClosedAbstractStrandDiagram` with n vertices into components is linearly bound by n and proceeds in $\mathcal{O}(n)$ time. Subsequently, the time to compare the number of components each in A and B takes constant time.

Input: `Closed Abstract Strand Diagram` A
Output: `A list of the connected components of` A

```
1  List<ClosedAbstractStrandDiagram> components
2  Stack<Vertex> vertexStack
3  for v in A.vertices do
4      if v.isNotFound() then
5          v.markFound()
6          vertexStack.push(v)
7          DoublyLinkedList<Vertex> newComponent
8          while vertexStack.isNotEmpty() do
9              w0 = vertexStack.pop()
10             for s in w0.strands do
11                 w1 = w0.getOtherVertex(s)
12                 if w1.isNotFound() then
13                     w1.markFound()
14                     vertexStack.push(w1)
15                     newComponent.add(w1)
16             end
17         end
18         c = new ClosedAbstractStrandDiagram(newComponent)
19         components.add(c)
20 end
21 return components
```

Algorithm 19: Algorithm to separate a `ClosedAbstractStrandDiagram` A into its connected components

4. **Compare number of vertices in each component**

The problem of determining whether A and B are isomorphic includes determining whether we can construct a bijection $\phi : \{A_1, A_2, \ldots, A_k\} \to \{B_1, B_2, \ldots, B_k\}$ such that if

$\phi(A_i) = B_j$ then A_i and B_j are isomorphic. Before doing so, we can perform the cursory check that there exists a bijection $\phi_0 : \{A_1, A_2, \ldots, A_k\} \to \{B_1, B_2, \ldots, B_k\}$ such that if $\phi_0(A_i) = B_j$ then A_i and B_j have the same size. If no such bijection exists then there is no possible mapping between components which results in total isomorphism between A and B.

To do this we create a `Dictionary` with entries (A_i, m_i) where A_i is a component of A and m_i is the size of that component. We create a similar `Dictionary` for B. Next we sort each `Dictionary` by value, i.e. by size. Finally, we obtain an `Array` of the values from each `Dictionary`. If those arrrays are equal then we know that the aforementioned bijection ϕ_0 exists, and we can proceed with isomorphism checking.

`Array` sorting occurs in $\mathcal{O}(n \log n)$ time, and the length of our `Array` is the number of components per strand diagram. As the number of components is bound by the number of vertices, this check performs in $\mathcal{O}(n \log n)$ time where n is the number of vertices in the original reduced closed abstract strand diagram.

4.3 Implementing the Isomorphism Checker

Once all of these checks have passed, we are ready to construct the bijection $\phi : \{A_1, A_2, \ldots, A_k\} \to \{B_1, B_2, \ldots, B_k\}$ such that if $\phi(A_i) = B_j$ then A_i and B_j are isomorphic. For now, we will limit our discussion to the case in which A an B each consist of one single component. We will discuss the contribution of multiple components in Section 4.7. We will base our search for isomorphism between single component `ClosedAbstractStrandDiagram`s A and B on Definition 3.1.1 of isomorphism.

If A and B have the same size then we begin our isomorphism construction. We outline our isomorphism method in Algorithm 20. We initialize two dictionaries, `vertexBijection` and `strandBijection`, to represent our bijective functions $\phi : V_A \to V_B$ and $\psi : E_A \to E_B$, respectively (lines 1-2). The `vertexBijection` will contain entries of the form $\big(v, \phi(v)\big)$ for vertex $v \in V_A$ and `strandBijection` will contain entries of the form $\big(s, \psi(s)\big)$ for strand $s \in E_A$. Next

Input: `ClosedAbstractStrandDiagrams` A `and` B
Output: `The boolean value for whether` A `and` B `are isomorphic graphs or not`

```
1  Dictionary<vertex,vertex> vertexBijection
2  Dictionary<strand,strand> strandBijection
3  Stack<vertex> vertexStack
4  Stack<strand> strandStack
5  boolean keepGoing = true
6  vertex a = A.getFirstVertex()
7  for b in B.vertices do
8      if a.type == b.type then
9          vertexBijection.put(a,b)
10     else
11         continue
12     a.markFound()
13     b.markFound()
14     vertexStack.push(a)
15     while keepGoing && (vertexStack.isNotEmpty() || strandStack.isNotEmpty()) do
16         if vertexStack.isNotEmpty() then
17             processVertex()
18         if strandStack.isNotEmpty() then
19             processStrand()
20     end
21 end
22 return keepGoing
```

Algorithm 20: Algorithm to determine whether two closed abstract strand diagrams A and B are isomorphic

we fix a vertex a in A and enter a loop through all vertices in b.`vertices` (lines 6-7). In each iteration of this loop, we fix a vertex b, mark a and b as "found," and add (a, b) to our vertex bijection (lines 8-13).

Once we fix the mapping (a, b), we extend this partial isomorphism by adding new edge pairs and vertex pairs to ψ and ϕ (respectively). We do so by beginning at a and b and "crawling" through the strands and vertices in the rest of A and B. Recall that for any given closed abstract strand diagram with vertex set V and edge set E, each vertex $v \in V$ is connected to exactly three edges and each edge plays a distinct role at v. It follows that there are no choices to be made in the construction of our bijections, since ψ must preserve the role of each edge $e \in E_A$ at each vertex $v \in V_A$.

Beginning at v_0 and w_0, we incrementally crawl through A and B and compare each vertex pair and strand pair that we encounter. We use two helper methods and two stacks to accomplish this.

We call the `processVertex()` helper method in Algorithm 20 on line 17. We outline this method in Algorithm 21. We begin by popping `Vertex` v from the `vertexStack` and getting $w = \phi(v)$ from the `vertexBijection` (lines 1-2). Then we expand the partial isomorphism to each of the `Strand`s connected to v and w. For each `Strand` s which plays role r at v, we get the corresponding `Strand` t that plays the same role r at w (lines 4-5). In order for this isomorphism to work, our `strandBijection` must contain the entry (s, t). Four cases follow:

1. If `strandBijection` already contains the entry (s, t), then we move on to the next `Strand` pair, since this is the desired case (line 8)

2. If `strandBijection` contains the entry (s, x) for some `Strand` x other than t, then this partial isomorphism has failed and we must start over with new a and b (line 10)

3. If neither s nor t has been found yet, then we add them to the `strandBijection` and add them to the `strandStack` to process (line 13)

4. If only one of either s or t has been found, then this partial isomorphism has failed and we must start over with new a and b (line 18

We have now finished processing the `Vertex` v.

Returning to Algorithm 20, we call the `processStrand()` helper method on line 19. This method is described in Algorithm 22. We begin by popping `Strand` s from the `strandStack` and getting $t = \psi(s)$ from the `strandBijection` (lines 1-2). Then we expand the partial isomorphism to include the begin vertices at s and t, $sBegin$ and $tBegin$, as well as the end vertices at s and t, $sEnd$ and $tEnd$ (lines 3-6). In order for this isomorphism to work, our `vertexBijection` must contain the entries $(sBegin, tbegin)$ and $(sEnd, tEnd)$. We will first check for the former; four cases follow:

Input: `Vertex to process`
Output: `Updated` *keepGoing* `boolean, strandBijection, vertexBijection, and vertexStack`

```
1  v = vertexStack.pop()
2  w = vertexBijection.get(v)
3  for r in v.strandRoles do
4  |  s = v.getStrand(r)
5  |  t = w.getStrand(r)
6  |  if s.isFound() && t.isFound() then
7  |  |  if strandBijection.contains(s) then
8  |  |  |  if strandBijection.get(s) == t then
9  |  |  |  |  continue
10 |  |  |  else
11 |  |  |  |  keepGoing = false
12 |  |  |  |  return
13 |  |  else if s.isNotFound() && t.isNotFound() then
14 |  |  |  s.markFound()
15 |  |  |  t.markFound()
16 |  |  |  strandBijection.put(s,t)
17 |  |  |  strandStack.push(s)
18 |  |  else
19 |  |  |  keepGoing = false
20 |  |  |  return
21 end
```

Algorithm 21: Algorithm for processVertex() helper method in Algorithm 20

1. If `vertexBijection` already contains the entry $(sBegin, tBegin)$, then continue crawling since this is the desired case (line 9)

2. If `vertexBijection` contains the entry $(sBegin, x)$ for some `Vertex` x other than $tBegin$, then this partial isomorphism has failed an we must start over with new a and b (line 11)

3. If neither $sBegin$ nor $tBegin$ has been found yet, then we add them to the `vertexBijection` and add them to the `vertexStack` to proceses (line 14)

4. If only one of either $sBegin$ or $tBegin$ has been found, then this partial isomorphism has failed and we must start over with new a and b (line 19)

The case for $(sEnd, tEnd)$ follows simmilarly on lines 22-36, after which we have finished processing the `Strand` s.

Input: Strands to process
Output: Updated *keepGoing* boolean, strandBijection, vertexBijection, and strandStrack

```
1  s = strandStack.pop()
2  t = strandBijection.get(s)
3  sBegin = s.getBeginVertex()
4  sEnd = s.getEndVertex()
5  tBegin = t.getBeginVertex()
6  tEnd = t.getEndVertex()
7  if sBegin.isFound() && tBegin.isFound() then
8  |  if vertexBijection.contains(sBegin) then
9  |  |  if vertexBijection.get(sBegin) == tBegin then
10 |  |  |  continue
11 |  |  else
12 |  |  |  keepGoing = false
13 |  |  |  break
14 else if sBegin.isNotFound() && tBegin.isNotFound() then
15 |  sBegin.markFound()
16 |  tBegin.markFound()
17 |  vertexBijection.put(sBegin,tBegin)
18 |  vertexStack.push(sBegin)
19 else
20 |  keepGoing = false
21 |  break
22 if sEnd.isFound() && tEnd.isFound() then
23 |  if vertexBijection.contains(sEnd) then
24 |  |  if vertexBijection.get(sEnd) == tEnd then
25 |  |  |  continue
26 |  |  else
27 |  |  |  keepGoing = false
28 |  |  |  break
29 else if sEnd.isNotFound() && tEnd.isNotFound() then
30 |  sEnd.markFound()
31 |  tEnd.markFound()
32 |  vertexBijection.put(sEnd,tEnd)
33 |  vertexStack.push(sEnd)
34 else
35 |  keepGoing = false
36 |  break
```

Algorithm 22: Algorithm for helper method processStrand() in Algorithm 20

Algorithm 20 occurs in quadratic time $\mathcal{O}(n^2)$ for n vertices. This is due to the outer loop (for) on line 7 and the inner loop (while) on line 15. The `processVertex()` and `processStrand()` method calls on lines 17 and 19 (respectively) run in constant time.

Now that we have determined that our two reduced `ClosedAbstractStrandDiagram`s are isomorphic, it is time to determine whether their cocycles belong to the same cohomology class or not.

4.4 Obtaining the Coboundary Matrix

We outline our method to construct the coboundary matrix M in Algorithm 23. Since A and B are isomorphic, they have the same coboundary matrix, so we will arbitrarily consruct the matrix using A. Before filling in the entries of M we must assign column IDs to each vertex $v \in V_A$ and row IDs to each strand $s \in E_A$ (lines 2-3 and 7-8).

We initialize M as a $\frac{3n}{2}$ by n matrix of 0's where n is the number of vertices in A and $\frac{3n}{2}$ is the number of strands (line 11). Next, for each `Vertex` v with column ID j we iterate through v`.strandRoles`. If the role r of a `Strand` s with row ID i at v is parent, right parent, or left parent, then s is incoming at v (line 18). Thus we must subtract 1 from the current entry $M[i, j]$ (line 19). If the role of s at v is child, right child, or left child, then s is outgoing at v (line 20). Thus we must add 1 to the current entry $M[i, j]$ (line 21). The reason we iterate through roles instead of strands themselves is that some strands play multiple roles at a single vertex and may be both incoming and outgoing. However, each role is distinct in all cases.

Assignments of row IDs and column IDs both occur in linear time $\mathcal{O}(n)$ where n is the number of vertices. Since we alter 3 matrix entries per per vertex and there are n vertices, the construction of the coboundary matrix takes $\mathcal{O}(n)$ time as well. Thus Algorithm 23 runs in linear time.

We must now compute the rank of the coboundary matrix, which can be done in $\mathcal{O}(n^\omega)$ time. The exponent ω is the matrix multiplication complexity coefficient, which is the minimum value of ω such that matrices can be multiplied in n^ω time. This value of ω is not known but

```
Input: ClosedAbstractStrandDiagram A
Output: The coboundary matrix M for A
1  i = 0
2  for v in A.vertices do
3  |  v.setColumnID(i)
4  |  i++
5  end
6  i=0
7  for s in A.strands do
8  |  s.setRowID(i)
9  |  i++
10 end
11 Matrix M = Matrix(n, 3n/2)
12 for v in A.vertices do
13 |  colID = v.columnID
14 |  for r in v.strandRoles do
15 |  |  s=v.getStrand(r)
16 |  |  rowID = s.rowID
17 |  |  curEntry = M.get(rowID,colID)
18 |  |  if r.equals(parent) || r.equals(lparent) || r.equals(rparent) then
19 |  |  |  M.set(rowID, colID, curEntry-1)
20 |  |  else if r.equals(child) || r.equals(lchild) || r.equals(rchild) then
21 |  |  |  M.set(rowID, colID, curEntry+1)
22 |  end
23 end
24 return M
```

Algorithm 23: Algorithm to generate the coboundary matrix for a `ClosedAbstactStrandDiagram` A

we do know that it has an upper bound of 2.373 [9]. Our implementation uses singular value decomposition to compute matrix rank, which runs in cubic time [13].

We remark that we only must compute the rank of this coboundary matrix once for a given closed abstract strand diagram, even if we find multiple isomorphisms. This will become relevant in Section 4.6.

4.5 Obtaining the Difference in Cocycles

To obtain a cocycle for each closed abstract strand diagram A and B, we construct a $\frac{3n}{2}$ by 1 vector C where entry $C[i, 1]$ is the difference in c-values for the `Strand` s (with row ID i) and $\psi(s)$,

as indicated by the `strandBijection` from the isomorphism construction. We initialize C on line 2 of Algorithm 24 and enter a loop over all `Strand`s in A on line 3. For each pair (s, t) in `strandBijection`, we store the difference in their c-values in the appropriate entry of C (lines 5-7).

Input: `Strand bijection for two ClosedAbstractStrandDiagrams` A `and` B
Output: `The difference in cocycles for` A `and` B

```
1 n = A.vertices.size()
2 C = Matrix[3*n/2,1]
3 for s in A.strands do
4 |   t = strandBijection.get(s)
5 |   c_s = s.cValue
6 |   c_s = t.cValue
7 |   C.set(s.rowID,0,c_s − c_s)
8 end
9 return C
```

Algorithm 24: Algorithm to compute the difference in cocycles for two `ClosedAbstractStrandDiagram`s A and B

The loop (for) over all of the strands in A on line 3 requires linear time for this algorithm.

4.6 Computing the Rank of the Augmented Matrix

Our final step is to compute the rank of the matrix $\left[M|\vec{a}-\vec{b}\right]$ for known coboundary matrix M and cocycle difference $\vec{a}-\vec{b}$. Recall that we have already computed the rank of the matrix M. If this rank is equal to the rank of the augmented matrix, then we conclude that A and B are conjugate; otherwise they are not.

We can compute the rank of the augmented matrix in slightly more efficient time than it took to compute the rank of the original matrix M.

Theorem 4.6.1 (Frandsen, Frandsen)**.** *Dynamic matrix rank over an arbitrary field can be solved using $\mathcal{O}(n^2)$ arithmetic operations per element change (worst case). This bound is valid when a change alters arbitrarily many entries in a single column. Given an initial matrix the data structure for the dynamic algorithm can be built using $\mathcal{O}(n^\omega)$ arithmetic operations* [6].

We obtain the augmented matrix through $\frac{3n}{2}$ element changes in the final column of the matrix. Therefore, the rank of this new matrix can be computed in $\mathcal{O}(n^2)$ time [6].

In the worst case, we must compute the rank for this augmented matrix one time per vertex. It *can* happen that the number of isomorphisms between two closed abstract strand diagrams grows linearly with the number of vertices, even though we believe this case to be rare. Thus the loop (for) on line 7 of Algorithm 20 requires linear time and the rank computation requires quadratic time. It is at this point, of course, that we can decide the conjugacy of our orginal elements and the algorithm is complete. Thus our final runtime is $\mathcal{O}(n^3)$.

We remark that a randomized algorithm found that the runtime for computing the rank of the updated matrix is $\mathcal{O}(n^{1.495})$ [15], which gives our algorithm an average case runtime of $\mathcal{O}(n^{2.495})$. Additionally, in the event that two elements of V are *not* conjugate, we believe that this will most often be determined after separating the closed reduced strand diagrams into components and comparing their sizes, which should have an average runtime of $\mathcal{O}(n \log n)$ due to list sorting.

4.7 The Role of Multiple Components

Up until now we have discussed only the case in which our closed abstract strand diagrams consist of a single component each. We will now discuss the multi-component case.

We have two closed abstract strand diagrams A and B, both with n vertices, which we have already broken up into components $\{A_1, A_2, \ldots, A_k\}$ and $\{B_1, B_2, \ldots, B_k\}$. Since the number of components in A is the same as the number of components in B, we can proceed to search for isomorphisms between A and B. We will first consider the case for which each component has the same size m. Since the original diagrams A and B have size n total, we know that $m = \frac{n}{k}$.

We know that the time to compare the equivalence of two components of size n takes $f(n) = \mathcal{O}(n^3)$ time. In the worst case, we must compare A_1 to all k components in B, which takes $kf(m)$ time. Since A_1 has now been matched to a component in B, we only need compare A_2 to $k-1$ components of B. Thus the time to match A_2 with a component in B is $(k-1)f(m)$, etc. So to

create a mapping between all components in A and B takes

$$kf(m) + (k-1)f(m) + \cdots + f(m) = \frac{k(k+1)}{2} f(m)$$

We know that $f(n) = Cn^3$ for some constant C. Thus we have

$$f(n) = \frac{k(k+1)}{2} \cdot Cm^3 = \frac{k(k+1)}{2} \cdot C\left(\frac{n}{k}\right)^3 = \frac{k+1}{2k^{3-1}} \cdot Cn^3 = \frac{k+1}{2k^2} \cdot Cn^3$$

.

We can see that $\lim_{k\to\infty} \frac{k+1}{2k^2} = 0$. Thus the worst case runtime for isomorphism checking for equally sized components is when there is only one component in each closed abstract strand diagram.

Now suppose that the components $\{A_1, A_2, \ldots, A_k\}$ of A have potentially different sizes $\{m_1, m_2, \ldots, m_k\}$ respectively. Again, we know that the time to compare the equlity of two components of size n is $f(n) = Cn^3$ for some constant C. But as we search for mappings between components of A and components of B, if the components do not have the same size then we need not compare them since they can't possibly be isomorphic. Thus the maximum number of comparisons for the first component A_1 of A with size m_1 is equal to the number of components in B which have the same size as A_1, which could be in the best case 1 or in the worst case k. Then the time to find an isomorphism between A_1 and a component in B takes time $kf(m_1)$. Once A_1 is mapped to some B_j, one fewer comparisons is required for A_2. We can see that this case has reduced to the previous case.

We conclude that the worst case for isomorphism checking is when there is only one component in each closed abstract strand diagram. For this reason, we need not analyze the multi-component case any further, and we conclude that the runtime of our final conjugacy checking program is indeed $\mathcal{O}(n^3)$.

5
Conclusion and Future Work

We have presented a cubic time algorithm to solve the conjugacy problem on Thompson's group V using strand diagrams. We have also presented data structures to store elements of V and perform the necessary operations on them.

Due to the use of rank computation, our proposed algorithm uses the least upper bound on the matrix multiplication coefficient as it is currently known, which is about 2.373 [9]. We also use Frandsen and Frandsen's method for rank one updates which runs in quadratic time [6].

Our actual implementation uses the Jama matrix library for storing and manipulating matrices [13], which calculates matrix rank in cubic time. Thus the worst case runtime for our implementation is quartic, but we believe the average case runtime to be significantly more efficient.

We release our implementation in the form of a Java applet and graphical interface programmed in Java that can be freely downloaded and run offline. We also release our source code in the following GitHub repository:https://github.com/rnales/ConjugacyV. To the best of our knowledge, this is the first implementation of an algorithm to solve the conjugacy problem in V using strand diagrams. We hope that this software will be useful to the research community in Thompson's groups.

For future work, we would like to bound the number of free loops and the sum of c-values of free loops using number of vertices. We would also like to better understand the typical number of isomorphisms between two reduced closed abstract strand diagrams, since it currently contributes a factor of n to the runtime of our algorithm.

Bibliography

[1] S. Adyan, *Finitely presented groups and algorithms*, Dokl. Akad. Nauk SSSR, 1957, pp. 9–12.

[2] J. Belk and F. Matucci, *Conjugacy and Dynamics on Thompson's Groups*, Geometriae Dedicata **169.1** (2014), 239–261.

[3] J. Cannon, W. Floyd, and W. Parry, *Introductory Notes on Richard Thompson's Groups*, L'Enseignement Mathematique **42** (1996), 215–256.

[4] H.Y. Cheung, T. Kwok, and L. Lau, *Fast Matrix Rank Algorithms and Applications*, Journal of the ACM **60** (2013), no. 5, Article 31.

[5] M. Dehn, *Über unendliche diskontinuierliche Gruppen*, Mathematische Annalen **71** (1911), no. 1, 116–144.

[6] G. Frandsen and P. Frandsen, *Dynamic Matrix Rank*, Theoretical Computer Science (2009), 4085–4093.

[7] G. Higman, *Finitely presented infinite simple groups*, Department of Pure Mathematics, Department of Mathematics, I.A.S. Australian National University, Canberra, 1974, Notes on Pure Mathematics, No. 8 (1974).

[8] N. Hossain, R. McGrail, J. Belk, and F. Matucci, *Deciding Conjugacy in Thompson's Group F in Linear Time*, Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 15th International Symposium on. IEEE (2013).

[9] F. Le Gall, *Powers of tensors and fast matrix multiplication*, International Symposium on Symbolic and Algebraic Computation (2014), 296–303.

[10] R. Lyndon and P. Schupp, *Combinatorial Group Theory*, Springer, 2001.

[11] P. Novikov, *Unsolvability of the conjugacy problem in the theory of groups.(Russian)*, Izv. Akad. Nauk SSSR. Ser. Mat **18** (1954), 485–524.

[12] M. Rabin, *Recursive unsolvability of group theoretic problems*, Ann. of Math **67** (1958), no. 2, 172–194.

[13] C. Reuden, J. Shindelin, M. Hiner, and K. Eliceiri, *Jama*, `https://github.com/fiji/Jama`.

[14] O. Salazar-Diaz, *Thompson's group V from the dynamical viewpoint*, PhD thesis, State University of New York at Binghamton, 2006.

[15] P. Sankowski, *Faster Dynamic Matchings and Vertex Connectivity*, SODA '07 Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, 118–126.