2013

Implementation of the Solution to the Conjugacy Problem in Thompson's Groups

Nabil Tarique Hossain
Bard College

Implementation of the Solution to the Conjugacy Problem in Thompson's Groups

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

by
Nabil Hossain

Annandale-on-Hudson, New York
May, 2013

Abstract

In this project, we present an efficient implementation of the solution to the conjugacy problem in Thompson's group F, a certain infinite group whose elements are piecewise-linear homeomorphisms of the unit interval $[0, 1]$. Our algorithm checks for conjugacy by constructing and comparing directed graphs called strand diagrams. We provide a comprehensive description of our solution algorithm, including the data structure we used to hold strand diagrams and the additional subroutines for manipulations of strand diagrams. We prove that our algorithm theoretically achieves an $O(n)$ bound in the size of the input, and we present a $O(n^2)$ working solution as an executable JAR file, a web application, and Java source code.

Contents

Abstract **1**

Dedication **6**

Acknowledgments **7**

Introduction **9**

1 Background **12**
- 1.1 Conjugacy 12
- 1.2 Directed Graphs Embedded on Surfaces 15
- 1.3 Thompson's Group F 22
  - 1.3.1 Dyadic Rearrangements 23
  - 1.3.2 Tree Diagrams 24
- 1.4 Strand Diagrams 26
  - 1.4.1 Strand Diagram Manipulations 28

2 Annular Strand Diagrams **31**
- 2.1 Closing Strand Diagrams 31
- 2.2 Reductions 32
- 2.3 Concentric Components 35
- 2.4 The Cutting Path 40
- 2.5 Isotopy of Reduced Annular Strand Diagrams 43

3 Algorithm for the Conjugacy Problem in F **47**
- 3.1 Algorithm Overview 48
- 3.2 The Data Structure 50
  - 3.2.1 Background: Doubly Linked Lists 50

3.2.2 Class: Edge 51
3.2.3 Class: Vertex 53
3.2.4 Class: Graph 54
3.2.5 Class: Strand 55
3.2.6 Class: Annular 55
3.3 Strand Diagram Generation 56
3.4 Reducing 58
3.4.1 Keeping Track of Potential Future Reductions 60
3.4.2 Cutting Path Update and Free Loop Generation 61
3.5 Connected Component Labeling 66
3.6 Encoding Annular Strand Diagrams into Planar Graphs 67
3.6.1 The Encoding Algorithm 72
3.7 Retrieving Annular Strand Diagrams from Planar Graph Encodings 72
3.8 Isomorphism Checking 74
3.9 Our Actual Implementation 75
3.9.1 Isotopy Detector Given two Corresponding Vertices 76
3.9.2 Isotopy Detector Given two Connected Reduced Annular Strand Diagrams 78
3.10 Results 79
3.10.1 Verification of Results 80
3.10.2 Shortest Cyclically Reduced Word for the Identity 82
3.11 The Software 82
3.11.1 The Variants and the Implementation Details 83
3.11.2 Using the Application 84

4 Conclusion and Future Work **90**

Appendix: Algorithm Descriptions **92**

Bibliography **99**

List of Figures

1.2.1 Homotopy of curves on the plane. 18
1.2.2 Isotopy of graphs on the plane. 19
1.2.3 A, B, and C are different embeddings of the same planar graph. None of the embeddings are isotopic on the plane, but B and C are isotopic on the sphere. 19
1.3.1 Generators for Thompson's Group F. The right half of x_1 is the same as x_0. 24
1.3.2 The infinite binary tree of standard dyadic intervals in $[0, 1]$ (image taken from [2]). 25
1.4.1 A strand diagram, a merge, and a split (image taken from [3]). 26
1.4.2 Construction of the strand diagram for x_0 from its tree diagram (image taken from [3]). 27
1.4.3 Generators for F and their inverses. 28
1.4.4 The two reduction rules (image taken from [3]). 28
1.4.5 Composing elements of F by concatenating their strand diagrams. Note that the concatenation makes a type II reduction move possible around the vertex v_s (image taken from [3]). 29

2.1.1 Annular strand diagram for x_1 obtained by closing its strand diagram . . . 32
2.2.1 The reduction moves for annular strand diagrams. A type I move is allowed when the shaded region is a topological disk. A type III move is allowed when the space between the two free loops is a topological annulus containing no vertices (image taken from [3]). 32
2.2.2 A type I move reduces the central annular strand diagram to a free loop, but a reduction II creates two concentric free loops. A reduction III on the diagram in the right merges the two free loops, preserving unique normal forms (image taken from [3]). 33

2.2.3 Reducing an annular strand diagram. In the pictures, the green regions are subject to type I moves, and the red and blue regions are each subject to type II moves. The type II move on the red region in (a) breaks the connected annular strand diagram into two connected components. 34
2.3.1 A reduction II move causing a connected component to split into two (each shaded region represents a component). 36
2.3.2 Two reduced annular strand diagrams ($\mathcal{A}_1$ has been taken from [3]). 37
2.4.1 For a cutting path, the edge crossing in (a) is allowed and (b) is not allowed 40
2.4.2 Update of the cutting path for each reduction move. 41
2.4.3 Status of a cutting path (a) after closing a strand diagram (crosses edge e_c), (b) after performing a type II reduction, and (c) after the annular strand diagram is reduced. The numbers denote the order of the edges in the cutting path 42
2.5.1 Three reduced annular strand diagrams in which $\mathcal{A}_1$ and $\mathcal{A}_2$ are isotopic. . 45

3.1.1 Overview of the solution algorithm for the conjugacy problem in F 49
3.2.1 The Java model of the data structure used for the solution to the conjugacy problem in F. Note that all the linked lists are doubly linked. 52
3.4.1 (a) Reduction I and (b) reduction II, with labeled edges and the split involved in the reduction labeled v. The green vertices can be splits or merges, they may not be distinct from each other or from the yellow vertices. 58
3.4.2 Special cases for updating `cuttingPath` during a type II move. Refer to (b) in Figure 3.4.1 for edge and vertex labels. 66
3.6.1 The encoding of $s \in X$ when s is a free loop. $\phi(s) \in G$ is the planar graph with a single vertex having a loop. 68
3.6.2 Different input and output types for edges. The vertex on the left is a merge, and the one on the right is a split 69
3.6.3 (a) shows the highlighted edge e_k which is the lone output of a merge and the lone input to a split. The encoding of e_k produces the planar graph g_k in (b). 69
3.6.4 Encoding of x_0x_0 to its corresponding planar graph. 70
3.10.1 A general structure for reduced annular strand diagrams with two vertices. The red dashed circles show free loops. 80
3.11.1 The user interface of the application for the conjugacy problem in F. 83

Dedication

To my father, mother, brother, and sister, who all make my world magical

Acknowledgments

Most of the credits for this project's success goes to my adviser James Belk, an excellent researcher, a wonderful teacher, and a very good friend. I will be always surprised at how he managed to advise four long senior projects simultaneously with ease. I also thank him for the skeleton program in Mathematica that supports basic strand diagram drawing.

Special thanks to my other adviser Robert McGrail, whose expertise in algorithms, and feedback on this writeup have been invaluable, and whose Friday barbecues were equally phenomenal. Plus he has been a wonderful career adviser, helping me to identify my own interests in science.

I would like to take this opportunity to thank all my teachers who have prepared me for future endeavors. Specifically I mention the name of Keith O'Hara whose Intro to Object Oriented Programming course got me to switch my major from Physics to Computer Science. Sven Anderson, I greatly benefited from consulting with you about the Java implementation issues regarding this project. And of course, Becky Thomas, whose classes have cemented my foundations on computer science. Her Theory of Computation course was fantastic, specially in the amount of material covered.

I thank all the friends who have supported me the last four years, making Bard feel homely. Anis Zaman, you have been an incredible friend, and more than that, like a brother. I will always remember the wonderful times we shared, and the way we stuck together during times of frustrations. Azfar Khan, you are a great roommate, and the top chef in the house without a question. Blagoy Kaloferov, I will not forget the late night FIFA games during stressful times, and also the free car rides to my apartment. Weiying Liu, Nazmus Saquib, and Prabarna Ganguly, you guys leave deep impressions sketched in my mind that I will never forget.

I am grateful to my guardians in the USA - Anowar Hasan, Fauzia Parvin, Mustafizur Rahman, and Mostofa Mohammad. Whenever I sought your help or advise, you offered them without any hesitation.

By thinking about two wonderful siblings who are willing to give away everything for me, I stay away from the wrong path and learn to be more responsible. I feel very lucky to be their big brother, and I am equally proud of them.

Lastly, I can never repay the enormous debt I owe to my parents, who have made unimaginable sacrifices towards my success. You show me hope in my distress, you teach me to stand up and fight, you have faith in me in my darkest moments, and you are the reason I am here today.

Introduction

Thompson's groups are certain infinite groups that are considered interesting in the fields of geometric group theory and homotopy theory. There are three such groups, called F, T, and V, defined by Richard J. Thompson in the 1960s. The elements of F are piecewise linear homeomorphisms of the interval $[0, 1]$ with finitely many breakpoints satisfying certain conditions, with function composition as the group operation. T and V are similar to F except T consists of homeomorphisms of the circle, and V the homeomorphisms of the Cantor set. For a comprehensive introduction to these groups, the reader is referred to Canon, Floyd and Perry [4].

In a group, the conjugacy problem is the problem of determining whether any two elements are conjugate. It was introduced by the mathematician Max Dehn in 1911 as one of three fundamental algorithmic problems in the study of infinite groups [5]. The conjugacy problem is not solvable in general [15], but solutions to the conjugacy problem are known for many important classes of groups such as free groups, surface groups, braid groups and so forth.

Guba and Sapir [8, 9] provided a solution to the conjugacy problem in F using graphs called *diagrams*. Building upon this solution, Belk and Matucci [3] introduced certain directed graphs called **strand diagrams**, and showed the existence of a solution to the conjugacy problem for all Thompson's Groups using these strand diagrams.

In this project, based on the methods in [3], we make the solution to the conjugacy problem in F precise, show that our solution executes in linear time, and present an efficient implementation of this solution as an application.

Given two input elements of F, our solution algorithm efficiently constructs the corresponding strand diagrams, modifies them using certain operations such as closing, concatenation, and reduction, and eventually compares the resulting strand diagrams to see if they are the same. We also present an efficient data structure to hold and manipulate strand diagrams in such a way that is geared towards achieving the fastest possible running time of the algorithm.

We prove that the best algorithm solving the conjugacy problem in Thompson's Group F is of the order $O(n)$, where n is the sum of the length of the two input elements compared for conjugacy. However, note that this proof uses the linear time algorithm proposed by Hopcroft and Wong [10] to determine whether two planar graphs are isomorphic, which has not been implemented to date because of its complicated design. As a result, our implementation replaces the isomorphism check between two planar graphs with an alternative method that directly compares two strand diagrams to determine whether they are the same. Due to this change, our implementation takes quadratic time.

We release a Java implementation of our solution algorithm as a web application, an executable JAR file, and the source code. As far as we know, this is the first implementation of the solution to the conjugacy problem in F. We hope that it will be helpful to researchers in studying Thompson's Groups.

The rest of this paper is organized as follows:

Chapter 1 provides all the relevant background information and the definitions which will be used throughout the paper; Chapter 2 introduces and discusses the structure of **annular strand diagrams**, which are obtained from strand diagrams, and used in our solution algorithm; Chapter 3 provides a comprehensive description of the algorithm for the conjugacy problem in F along with details on how to use our software; Chapter 4 concludes the paper, drawing attention to future work; and finally in the Appendix some of the important sub-algorithms in our implementation are detailed.

1
Background

1.1 Conjugacy

Definition 1.1.1. In a group G, elements $g_1, g_2 \in G$ are **conjugate** if there exists an element $h \in G$ such that $g_1 = hg_2h^{-1}$. In this case, we say that g_1 is the conjugate of g_2 by h, or h conjugates g_1 to g_2. △

Conjugacy is an equivalence relation, which partitions G into equivalence classes, known as **conjugacy classes**. Every element of a group belongs to one conjugacy class only, and two elements g_1 and g_2 are conjugate if and only if they belong to the same conjugacy class.

Example 1.1.2. Two permutations in the symmetric group S_n are conjugate if and only if the permutations have the same cycle structure (See [6], Proposition 4.3.11). For instance, the group S_3 has $3! = 6$ permutations of the set $P = \{1, 2, 3\}$. In cycle notation, $S_3 = \{(1),\ (1\ 2),\ (1\ 3),\ (2\ 3),\ (1\ 2\ 3),\ (1\ 3\ 2)\}$. There are three conjugacy classes of S_3:

1. The identity (1) is conjugate only to itself: $\{(1)\}$
    - $(1) = (1)(1)(1)^{-1}$

2. The 2-cycles: $\{(1\ 2),\ (2\ 3),\ (1\ 3)\}$

   - $(1\ 2) = (1\ 3)(2\ 3)(1\ 3)^{-1}$
   - $(1\ 2) = (2\ 3)(1\ 3)(2\ 3)^{-1}$

3. The 3-cycles: $\{(1\ 2\ 3),\ (1\ 3\ 2)\}$

   - $(1\ 2\ 3) = (1\ 2)(1\ 3\ 2)(1\ 2)^{-1}$ △

For the following definition, we assume that the reader is familiar with the term generating set. Recall the definition of a word.

Definition 1.1.3. In a group G with a generating set S, a **word** in S is an arbitrary product of elements of S and their inverses, and a **reduced word** in S is a word that does not have any adjacent pair xx^{-1} or $x^{-1}x$, for all $x \in S$. △

Notice that a word represents an element of G. Since S is a generating set in G, every element of G can be represented by a word in S.

In 1911, the German American mathematician Max Dehn formulated the following three fundamental problems for groups [5]:

Definition 1.1.4. In a group G with a given generating set S, the **word problem** is the decision problem of determining whether two given words w_1 and w_2 in S represent the same element of G. △

Definition 1.1.5. In a group G with a given generating set S, the **conjugacy problem** is the decision problem of determining whether two given words w_1 and w_2 in S are conjugate. △

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

Dehn was hoping to obtain general solutions to word and conjugacy problems for any finitely presented group as well as a general solution to the isomorphism problem. It has since been shown that the word and conjugacy problems are undecidable for some finitely presented groups [15], and that the isomorphism problem is undecidable in general [1, 16].

Note that solving the conjugacy problem is trivial on finite groups because given a finite group G and two elements $g_1, g_2 \in G$, we can try out all the other $h \in G$ to determine whether there exists any relationship $g_1 = hg_2h^{-1}$. However, this algorithm will not terminate on infinite groups in the case where two elements are not conjugate. The following example examines an infinite group called a **free group** in which the conjugacy problem is decidable.

Example 1.1.7. A group G is called **free** if it has a generating set S with no relations between the generators. Note that every element in G can be written uniquely as a reduced word in S. For example, if $S = \{x, y\}$, then the elements of G are all of the reduced words involving x, x^{-1}, y, and y^{-1}.

A word is **cyclically reduced** if it is reduced and its first and last elements are not inverses of each other. We can cyclically reduce any reduced word by canceling all inverse pairs from the beginning and the end. For example:

$$x^{-1}yxy^{-1}xxy^{-1}x \qquad \to \qquad yxy^{-1}xxy^{-1} \qquad \to \qquad xy^{-1}xx$$

Note that the resulting cyclically reduced word is conjugate to the original reduced word:

$$x^{-1}yxy^{-1}xxy^{-1}x \qquad = \qquad (x^{-1}y)(xy^{-1}xx)(x^{-1}y)^{-1}$$

If w_1 and w_2 are cyclically reduced words, it is not hard to show that w_1 and w_2 are conjugate if and only if w_2 is a cyclic permutation of w_1. For instance, $xy^{-1}xx$ is conjugate to $xxxy^{-1}$ because

$$xxxy^{-1} = (xx)(xy^{-1}xx)(xx)^{-1}.$$

Therefore, we can determine whether any two words are conjugate by cyclically reducing them and checking whether the results are cyclic permutations of each other. It follows that the conjugacy problem in free groups is decidable.

Given two strings s and t over an alphabet Σ, the **string matching problem** is the problem of determining whether s is a substring of t. It has been proven by Knuth,Morris and Pratt [12] that this problem is decidable in linear time in the length of the input strings. By reducing the problem of checking whether two cyclically reduced words are the same to the string matching problem, Madlener and Avenhaus [13] have shown that the solution to the conjugacy problem in free groups is solvable in $O(n)$, where n is the sum of the lengths of the two input words in the free group. $\Diamond$

As we will see later in this paper, the solution to the conjugacy problem in Thompson's Group F proposed by [3] is similar to the solution shown above for free groups. Instead of using reduced words, their solution begins by cyclically reducing strand diagrams. Next, instead of checking whether the results are cyclic permutations of each other, they check whether the resulting annular strand diagrams are **isotopic** (see Definition 1.2.11).

1.2 Directed Graphs Embedded on Surfaces

In this section, we discuss graphs on a surface S. For our purposes in this project, we restrict our discussions to the following surfaces: the Euclidean plane $\mathbb{R}^2$, the sphere, the unit square, and the annulus. While the sphere can be thought of as $\mathbb{R}^2 \cup \{\infty\}$, both the unit square and the annulus are subspaces of the Euclidean plane.

Definition 1.2.1. A **directed graph** is a 4-tuple $G = (V, E, s, t)$ where:

1. V is the set of vertices,
2. E is the set of directed edges,
3. the function $s : E \to V$ assigns a **source** vertex to each edge, and

4. the function $t : E \to V$ assigns a **target** vertex to each edge. △

Notice that Definition 1.2.1 allows a directed graph to have one or more loops from a vertex to itself as well as multiple distinct directed edges between the same source and the same sink.

Definition 1.2.2. Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be undirected graphs. Then, an **isomorphism** of G and H is a pair (ϕ_V, ϕ_E) where:

1. $\phi_V : V_G \to V_H$ is a bijection, and
2. $\phi_E : E_G \to E_H$ is a bijection

such that an edge $e \in E_G$ connects vertices v_1 and v_2 in V_G if and only if $\phi_E(e) \in E_H$ connects $\phi(v_1)$ and $\phi(v_2)$ in V_H.

An **isomorphism of two directed graphs** G and H is an isomorphism (ϕ_V, ϕ_E) such that the source of each edge e in G corresponds to the source of $\phi_E(e)$ in H, and similarly the target of e corresponds to the target of $\phi_E(e)$. △

Two graphs G and H are said to be **isomorphic** if there exists an isomorphism between them. Intuitively, two isomorphic graphs can be thought of as similar graphs because they have certain common properties. In particular, any property that is true for an object in G and is preserved by the isomorphism is also true for its corresponding object in H.

For our purposes, we consider embeddings of graphs on surfaces. For the following definition, a **curve** on a surface S is any continuous, one-to-one function $\gamma : [0, 1] \to S$, where $\gamma(0)$ is the **begin point** and $\gamma(1)$ is the **end point** of the curve.

Definition 1.2.3. An **embedding** of a directed graph $G = (V, E, s, t)$ on a surface S is an ordered pair (f, c) where:

1. $f : V \to S$ is one-to-one,

2. $c = \{c_e\}_{e \in E}$ is a collection of curves on S, one for each directed edge, that do not intersect except at their begin points and end points, and

3. if $e \in E$, then c_e is a curve with begin point $f(s(e))$ and end point $f(t(e))$.

An **embedded graph** is a graph together with its embedding on some surface S. $\triangle$

In essence, an embedding of G on S is a visual representation or a drawing of G on S. In such a drawing, vertices are represented by points on S, each directed edge is a curve on S, and the directed edges are allowed to intersect only at common end vertices. Notice that it is possible for a directed graph to have several different embeddings on a surface.

Example 1.2.4. A **planar graph** is a graph that can be embedded on the plane. Figure 1.2.3 shows three different embeddings of the same planar graph. $\diamondsuit$

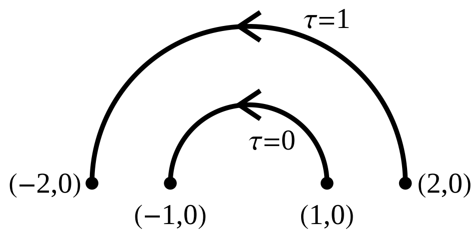
We need a formal notion of what it means to move a graph around on a surface. In order to formulate this notion, we need to understand what it means to move a curve around a surface.

Definition 1.2.5. A **homotopy** of curves on a surface S is a family of curves $\{\gamma_\tau\}_{\tau \in [0,1]}$ on S such that the function $f : [0,1] \times [0,1] \to S$ defined by $f(\sigma, \tau) = \gamma_\tau(\sigma)$ is continuous. $\triangle$

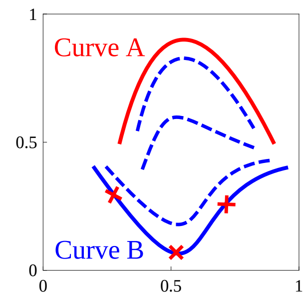
Intuitively, a homotopy is a way of "continuously deforming" a curve on a surface. We can think of τ as the time during the deformation, and for each value of τ corresponding to a curve C, σ represents the spatial coordinates along the length of C (i.e., where a point is on C). Observe that we are allowing the end points of a curve to move during the homotopy. Thus, any two curves on the plane are homotopic.

Example 1.2.6. In Figure 1.2.1.(a), at $\tau = 0$ the curve is

$$\gamma_0(\sigma) = (\cos(\pi\sigma), \sin(\pi\sigma)),$$



(a) a semicircular curve centered at (0,0)



(b) Deformation of curve B into A

Figure 1.2.1. Homotopy of curves on the plane.

and at $\tau = 1$ the curve is

$$\gamma_1(\sigma) = (2\cos(\pi\sigma), 2\sin(\pi\sigma)).$$

A homotopy for this semicircular curve is

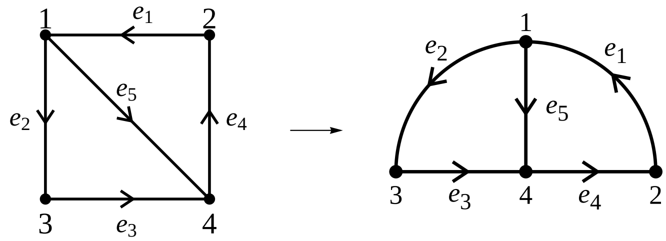
$$\{\gamma_\tau(\sigma) = ((\tau + 1)\cos(\pi\sigma), (\tau + 1)\sin(\pi\sigma))\}_{\tau\in[0,1]}.$$

In (b), each of the curves represents a unique value of τ with σ continuously increasing from 0 to 1. The blue, dashed curves show the deforming of curve B to curve A at discrete values of τ, each leading to a unique γ_τ. The red crosses represent the pair $(\sigma, 0)$ where $\sigma \in [0, 1]$ is unique for each red cross. The solid curve B corresponds to the range of γ_0, and the range of γ_1 is the curve A. Thus, B is continuously deformed into A as τ is increased from 0 to 1. $\Diamond$

Definition 1.2.7. An **isotopy** of directed graphs on a surface S is a family of embeddings $\{(f_\tau, c_\tau)\}_{\tau\in[0,1]}$ of a directed graph $G = (V, E, s, t)$ on S such that:

1. for each $v \in V$, $\tau \mapsto f_\tau(v)$ is continuous, and
2. for each $e \in E$, $\{(c_\tau)_e\}_{\tau\in[0,1]}$ is a homotopy of directed curves, where $f_\tau(s(e)) = (c_\tau(0))_e$ and $f_\tau(t(e)) = (c_\tau(1))_e$. $\triangle$

Isotopy can be visualized as the continuous deformation of an embedding on a surface. Notice that isotopy is a consequence of the continuous deformation of the points on the



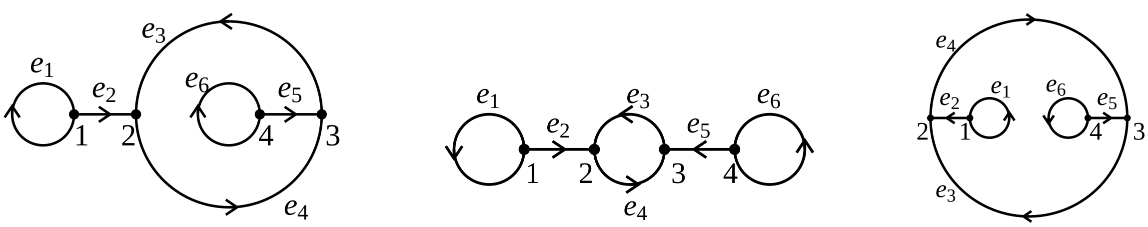
Embedding $\mathcal{E}_0$ Embedding $\mathcal{E}_1$

Figure 1.2.2. Isotopy of graphs on the plane.

surface representing the vertices, and the curves on the surface representing the edges, of the embedded graph. Figure 1.2.2 shows an isotopy of a directed graph on the plane.

Definition 1.2.8. Let $\mathcal{E}_0$ and $\mathcal{E}_1$ be two embeddings of a directed graph G on a surface S. Then $\mathcal{E}_0$ and $\mathcal{E}_1$ are **isotopic** if there exists an isotopy $(f_\tau, c_\tau)_{\tau\in[0,1]}$ on S such that at $\tau = 0$ the embedding is $\mathcal{E}_0$, and at $\tau = 1$ the embedding is $\mathcal{E}_1$. △

Example 1.2.9. Figure 1.2.3 shows three embeddings A, B, and C of the same directed graph on the plane. There is no way to move the vertices 1 and 4 in B into the circular topological space between edges e_3 and e_4. This shows that B is not isotopic to A and C. Because vertex 1 in A cannot be moved into the region enclosed by edges e_3, e_4, e_5, and e_6, we can safely claim that A is not isotopic to C. Thus, none of these embeddings are isotopic on the plane. ♢



Embedding A Embedding B Embedding C

Figure 1.2.3. A, B, and C are different embeddings of the same planar graph. None of the embeddings are isotopic on the plane, but B and C are isotopic on the sphere.

If two directed graphs are isomorphic, and one of the graphs has an embedding $\mathcal{E}$ on a surface, obviously this gives an embedding of the other graph on the surface. Stated in a different way, we can "compose" the embedding $\mathcal{E}$ with the isomorphism to obtain an embedding of the other graph. The following definition formalizes this notion.

Definition 1.2.10. Let G and G' be directed graphs. Let $\mathcal{E} = (f, c)$ be an embedding of G on a surface S, and let $\phi = (\phi_V, \phi_E)$ be an isomorphism from G' to G. Then the **induced embedding** of G' on S is $\mathcal{E}' = (f', c')$, where $f' = f \circ \phi_V$ and $c' = c \circ \phi_E$. We say that $\mathcal{E}'$ is the embedding of G' on S **induced** by ϕ. △

We can use induced embeddings to define isotopy between two different graphs on a surface.

Definition 1.2.11. Let G and H be directed graphs with embeddings $\mathcal{E}_G$ and $\mathcal{E}_H$ respectively on a surface S. Then G and H are **isotopic** on S if there exists an isomorphism $\phi : G \to H$ such that the induced embedding $(\mathcal{E}_H \circ \phi)$ is isotopic to $\mathcal{E}_G$ on S. △

Intuitively, we can think of two directed graphs G and H as isotopic on a surface if H is the image of G under some continuous deformation on the surface.

We will now discuss a simple algorithm that determines whether two embedded graphs are isotopic. This involves the notion of the rotation system.

Definition 1.2.12. Let G be a directed graph with vertex set V and with embedding $\mathcal{E}$ on a surface S. For each vertex $v \in V$, let ρ_v denote the counterclockwise order of the edges connected to v. Then the set $\{\rho_v \mid v \in V\}$ is called the **rotation system** of the embedding $\mathcal{E}$ on S. △

We need to clarify two major points about this definition:

1. The term "counterclockwise order" means the cyclic ordering of the directed edges connected to a vertex v. This is a cyclic ordering, i.e., linear ordering up to cyclic

permutation. Note that the word "counterclockwise" assumes some standard orientation of the surface S. In particular, we cannot define a rotation system for an embedding on a non-orientable surface.

2. Because our directed graphs can be multigraphs, they can have loops, which are edges from a vertex to itself. We need to distinguish the two appearances of the edge that forms a loop, that is, whether the edge comes into the vertex before it goes out, or the opposite. In this case, we need to mark the two occurrences of the edge in the counterclockwise order so that there is no ambiguity in the ordering.

In the example below, we use e^{in} to mark the incoming edge in the loop, and e^{out} to mark the outgoing edge in the loop.

Example 1.2.13. In Figure 1.2.3, observe that on the sphere B can be deformed into C by translating B to the northern hemisphere (think of it as the rear of the sphere) and then "pulling" the loops e_1 and e_6 towards the southern hemisphere (the front of the sphere). Then this deformed embedding of B will appear the same as the embedding C. Therefore, B and C are isotopic to each other on the sphere.

The rotation systems of the three directed graphs are:

- A $\rightarrow \{(e_1^{in}, e_1^{out}, e_2), (e_4, e_3, e_2), (e_3, e_5, e_4), (e_6^{out}, e_6^{in}, e_5)\}$
- • B $\rightarrow \{(e_1^{out}, e_1^{in}, e_2), (e_4, e_3, e_2), (e_3, e_4, e_5), (e_6^{out}, e_6^{in}, e_5)\}$
- C $\rightarrow \{(e_1^{out}, e_1^{in}, e_2), (e_4, e_3, e_2), (e_3, e_4, e_5), (e_6, e_6, e_5)\}$

Notice that the rotation systems of B and C are exactly the same, but that of A is different because in the counterclockwise order of the edges around vertex 1 for A starting at edge e_2, the input edge into the loop precedes the output edge in the loop. The opposite happens for vertex 1 of both B and C where the loop is directed counterclockwise and thus the output edge precedes the input edge in the loop. Furthermore, the counterclockwise order of the edges around vertex 3 in A is not an element of the rotation systems of B and C. $\Diamond$

Theorem 1.2.14. *Two embeddings of a connected directed graph on a sphere are isotopic if and only if both embeddings induce the same rotation system.*

The proof of this theorem follows from Theorem 3.2.4 and Corollary 3.2.5 in Section 3.3 in [14].

This theorem allows us to use rotation systems to check for isotopy of directed graphs on the sphere. Thus, in Example 1.2.13 we can immediately conclude that embeddings B and C are isotopic on the sphere because their rotation systems are the same. However, they are not isotopic on the plane. This is because the outer region in B corresponds to the inside region in C enclosed by edges e_3 and e_4. Furthermore, neither B nor C is isotopic to A on the sphere because the rotation system of A is different from those of B and C.

Corollary 1.2.15. *Let G and H be connected directed graphs embedded on the sphere. Then G and H are isotopic if and only if there exists an isomorphism $\phi : G \to H$ that preserves the counterclockwise order of the edges connected to corresponding vertices between G and H.*

Note that the correspondence between the vertices are defined by the isomorphism. Preserving the order means that the rotation systems for the corresponding vertex sets are the same.

1.3 Thompson's Group F

Most of the definitions in this section can be found in [2] and [4]. Thompson's Group F is one of the three Thompson's Groups, and it is a certain group of piecewise-linear homeomorphisms of the interval $[0, 1]$ under the operation function composition. We now describe this group, provide a generating set for it, and show how its elements can be represented in graphs called tree diagrams. For a thorough introduction to F, the reader is encouraged to look into [4].

1.3.1 Dyadic Rearrangements

Definition 1.3.1. A **dyadic subdivision** is any subdivision of the interval $[0, 1]$ obtained by:

1. choosing whether to divide the interval in half,
2. if chosen, then dividing the interval in half, and for each of the resulting intervals, looping back to step (1). $\triangle$

For example, $\{[0, \frac{1}{2}], [\frac{1}{2}, 1]\}$ and $\{[0, \frac{1}{8}], [\frac{1}{8}, \frac{1}{4}], [\frac{1}{4}, \frac{1}{2}], [\frac{1}{2}, 1]\}$ are dyadic subdivisions. Note that $\{[0, 1]\}$ is also a dyadic subdivision.

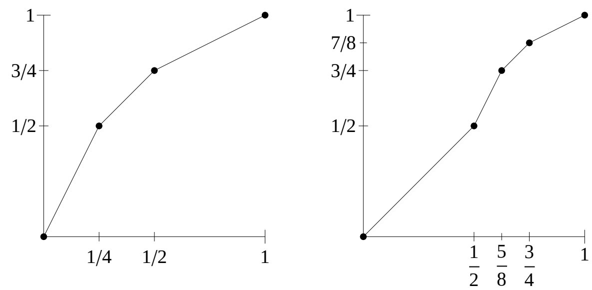
A **standard dyadic interval** is an interval of the form: $[\frac{k}{2^n}, \frac{k+1}{2^n}]$, where $k, n \in \mathbb{N}$ and $k \leq 2^n - 1$. Therefore, all the intervals in a dyadic subdivision are standard dyadic intervals. Moreover, a dyadic subdivision is the division of $[0, 1]$ into standard dyadic intervals.

If d_1 and d_2 are two dyadic subdivisions having the same number of standard dyadic intervals, then we can create a piecewise-linear homeomorphism $f : [0, 1] \to [0, 1]$ which linearly maps each interval of d_1 onto a corresponding interval of d_2. Then f is called a **dyadic rearrangement** of $[0, 1]$. For instance, Figure 1.3.1 shows two dyadic rearrangements.

Theorem 1.3.2. *Let $f : [0, 1] \to [0, 1]$ be a piecewise-linear homeomorphism. Then f is a dyadic rearrangement if and only if:*

1. *Each slope of f is a power of 2, and*
2. *Each breakpoint of f has dyadic rational coordinates*

For the proof of this theorem, the reader is referred to [2].



(a) x_0 (b) x_1

Figure 1.3.1. Generators for Thompson's Group F. The right half of x_1 is the same as x_0.

Corollary 1.3.3. *The set F of all dyadic rearrangements forms a group under function composition.*

The resulting group is called **Thompson's Group F**.

Proposition 1.3.4. *(also Theorem 1.3.9 in* [2]*) The two dyadic rearrangements x_0 and x_1 generate the group F with presentation*

$$\langle x_0, x_1 \mid x_1x_2 = x_3x_1, x_1x_3 = x_4x_1 \rangle$$

where $x_2 = x_0x_1x_0^{-1}, x_3 = x_0^2x_1x_0^{-2}$, and $x_4 = x_0^3x_1x_0^{-3}$.

As mentioned in [2] and [4], this is a common presentation for F. Note that our algorithm for the conjugacy problem in F only accepts input words in the generating set $\langle x_0, x_1 \rangle$.

1.3.2 Tree Diagrams

We now show how certain graphs called tree diagrams can be used to describe elements of F.

Proposition 1.3.5. *Any dyadic subdivision d can be encoded into a* **finite rooted binary tree** *T where:*

1. *the root of T represents the interval* $[0, 1]$*,*
2. *each vertex of T is a standard dyadic interval,*

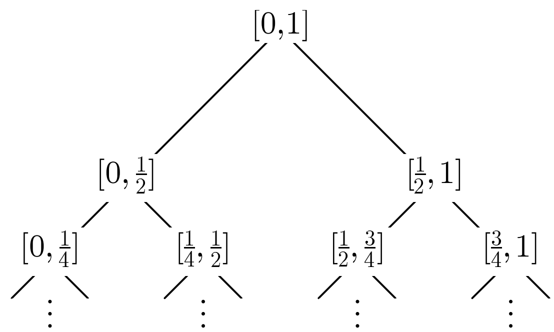


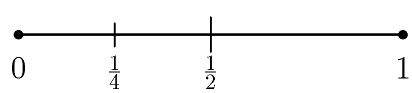
Figure 1.3.2. The infinite binary tree of standard dyadic intervals in $[0,1]$ (image taken from [2]).

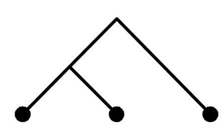
3. an edge of T is a pair of standard dyadic intervals (C, P) such that C is either the left half of P and we get a left edge, or the right half of P in which case we get a right edge, and

4. each leaf of T is an interval in d.

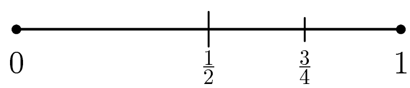
Another way of stating this proposition is that all dyadic subdivisions correspond to finite **subtrees** of the infinite binary tree shown in Figure 1.3.2. Using this encoding, we can identify any element of F as a pair of finite subtrees corresponding to the subdivisions in the domain and the range respectively. Such a pair of subtrees makes up the **tree diagram** for the element.

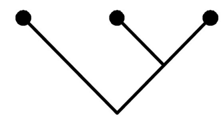
Example 1.3.6. The element x_0 is a piecewise linear function that maps intervals of the subdivision D which has breakpoints $\{0, \frac{1}{4}, \frac{1}{2}, 1\}$ onto the intervals of the subdivision R which has breakpoints $\{0, \frac{1}{2}, \frac{3}{4}, 1\}$. The subdivision D and its corresponding subtree are:



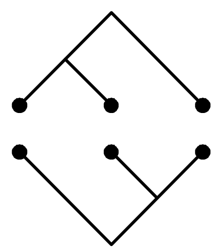


Similarly, the subdivision R and its corresponding subtree are:





Therefore, the *tree diagram* for x_0 can be obtained by pairing these two trees as shown below:



Note that the two trees are arranged in a way such that the corresponding leaves are vertically aligned. By convention, we place the subtree representing the domain above the subtree representing the range (all images in this example have been taken from [2]). $\Diamond$

1.4 Strand Diagrams

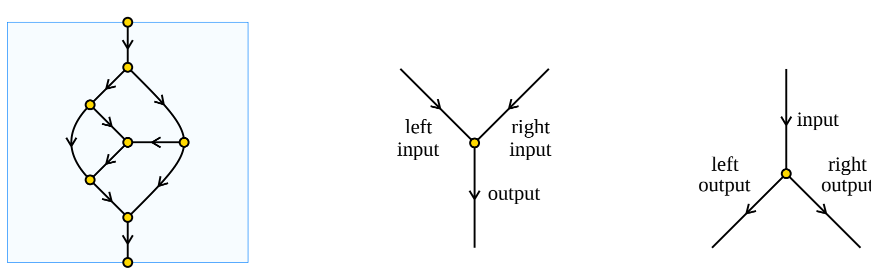


Figure 1.4.1. A strand diagram, a merge, and a split (image taken from [3]).

In this section, we introduce a certain type of planar, directed graphs called **strand diagrams**, which are derived from the tree diagrams that we described in Section 1.3.2. Moreover, previous research by Belk and Matucci [3] shows that in each Thompson's

Group there exists a solution to the conjugacy problem that involves strand diagrams, and our algorithm for the conjugacy problem in F also uses strand diagrams.

Definition 1.4.1. A **strand diagram** is a finite acyclic digraph embedded on the unit square with the following properties:

1. The graph has a **source** along the top edge of the square having an outgoing edge, and a **sink** along the bottom edge with an incoming edge.
2. Any other vertex is:

   (a) either a **merge** that has two incoming edges and one outgoing edge, or

   (b) a **split** that has one incoming edge and two outgoing edges (see Figure 1.4). △

Two isotopic strand diagrams are considered equal by convention.

A strand diagram corresponding to an element of F can be produced by gluing the domain and the range subtrees of the element at their corresponding leaves and making the edges directed from the domain subtree towards the range subtree. For instance, Figure 1.4.2 shows how the strand diagram corresponding to x_0 can be obtained.

Our implementation uses the generating set $\langle x_0, x_1 \rangle$ to create words for elements of F. Figure 1.4.3 shows the strand diagrams for the generators. Notice that the strand diagram for x_0^{-1} can be produced by reversing the edge directions in the strand diagram for x_0 and then flipping the resulting strand diagram vertically. In general, given the strand diagram

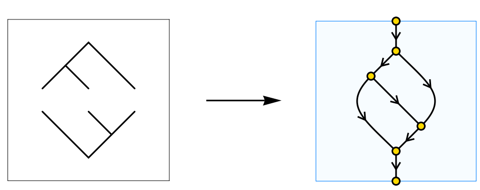
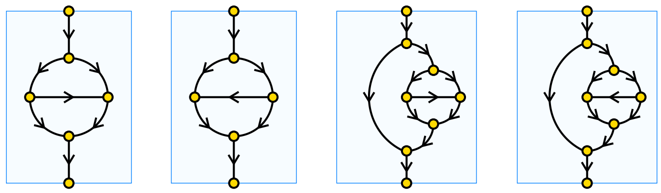


Figure 1.4.2. Construction of the strand diagram for x_0 from its tree diagram (image taken from [3]).



(a) x_0 (b) x_0^{-1} (c) x_1 (d) x_1^{-1}

Figure 1.4.3. Generators for F and their inverses.

S for an element $f \in F$, the strand diagram for f^{-1} can be obtained by reversing all the edge directions in S, and then flipping S vertically.

1.4.1 Strand Diagram Manipulations

We now describe certain operations that are used to modify strand diagrams in our algorithm for the conjugacy problem in F.

Definition 1.4.2. A **reduction** of a strand diagram is a simplification of the strand diagram using one of the two moves shown in Figure 1.4.4. We say that a strand diagram has been **reduced** if no reductions can be performed on it. $\triangle$

Notice that a type II reduction move is possible whenever there exists an edge from a merge to a split.

Definition 1.4.3. The **concatenation** of two strand diagrams s_1 and s_2 is the strand diagram created by by gluing the sink of s_1 to the source of s_2 and then removing the resulting vertex of degree 2. In this case, we say that s_1 has been concatenated to s_2. $\triangle$

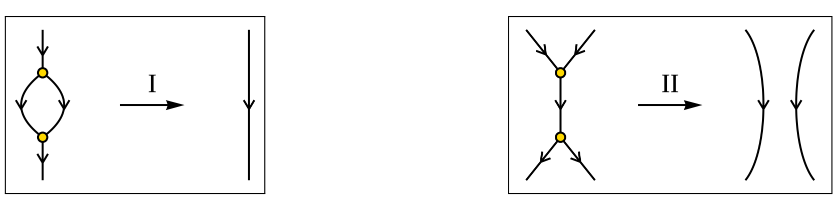
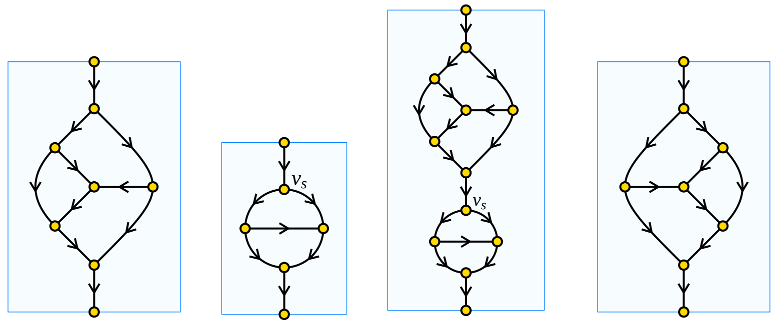


Figure 1.4.4. The two reduction rules (image taken from [3]).



f $\quad$ x_0 $\quad$ $x_0 \circ f$ $\quad$ $x_0 \circ f$ (reduced)

Figure 1.4.5. Composing elements of F by concatenating their strand diagrams. Note that the concatenation makes a type II reduction move possible around the vertex v_s (image taken from [3]).

Concatenation of strand diagrams is equivalent to composition of corresponding elements in F.

Example 1.4.4. Figure 1.4.5 shows the concatenation of two elements f and x_0, which produces the strand diagram for the element $x_0 \circ f$, which has word $x_0 f$. Notice that a concatenation immediately gives rise to a type II reduction move at the point of concatenation. $\Diamond$

The following theorem shows that the construction of a strand diagram is linear in the length of the corresponding word, and it plays a significant role in proving that our algorithm for the conjugacy problem in F is linear.

Theorem 1.4.5. *Let n be the length of the input word in the generating set $\langle x_0, x_1 \rangle$ for a strand diagram S. Let V_S and E_S be the number of vertices and edges respectively in S. Then $V_S + E_S \leq 15n + 3$.*

Proof. Let the input word for S have length n. We will prove the theorem using induction on n.

Base Case: Let $n = 0$. Then S is the strand diagram for the identity, which has a source, a sink, and an edge connecting them. Thus $V_S + E_S = 2 + 1 = 3 \leq 3$.

Inductive Case: Assume that for a strand diagram S corresponding to a word of length n, we have $V_S + E_S \leq 15n + 3$.

Let S' be a strand diagram for an input word w, of length $n+1$. By the inductive hypothesis, for the word with the first n characters in w, there exists a strand diagram S such that $V_S + E_S \leq 15n + 3$. Let g be the strand diagram for the $(n+1)$th element in w. Observe that $g \in \{x_0, x_0^{-1}, x_1, x_1^{-1}\}$. We can create S' by concatenating g to S. We have two cases to consider.

Case 1: $g = x_0$ or $g = x_0^{-1}$. Then $V_g = 6$ and $E_g = 7$. Since concatenation removes 2 vertices and 1 edge, we have

$$V_{S'} + E_{S'} = V_S + E_S + V_g + E_g - 2 - 1 = 15n + 3 + 6 + 7 - 2 - 1 = 15n + 13 \leq 15(n+1) + 3.$$

Case 2: $g = x_1$ or $g = x_1^{-1}$. Then $V_g = 8$ and $E_g = 10$. Again, since concatenation removes 2 vertices and 1 edge, we have

$$V_{S'} + E_{S'} = V_S + E_S + V_g + E_g - 2 - 1 = 15n + 3 + 8 + 10 - 2 - 1 = 15n + 18 \leq 15(n+1) + 3.$$

It follows that $V_{S'} + E_{S'} \leq 15(n+1) + 3$. □

2
Annular Strand Diagrams

Recall that our algorithm for the conjugacy problem in F involves the use of strand diagrams. To be more precise, the solution proposed by Belk and Matucci [3] is directly based on the manipulation of certain directed graphs called **annular strand diagrams**, derived from strand diagrams.

In this chapter, we discuss annular strand diagrams, provide the solution to the conjugacy problem in F [3] that involves modifying and comparing these graphs, and describe the strategies we use to dynamically monitor the structure of mutating annular strand diagrams, with particular emphasis on the decomposition into multiple connected components.

2.1 Closing Strand Diagrams

We can **close** any strand diagram embedded on the unit square by making the output from the parent of its sink the input to the child of its source, and then removing the source and the sink. This turns the strand diagram to a graph embedded in an annulus, called an **annular strand diagram**. One such example is shown in Figure 2.1.1.

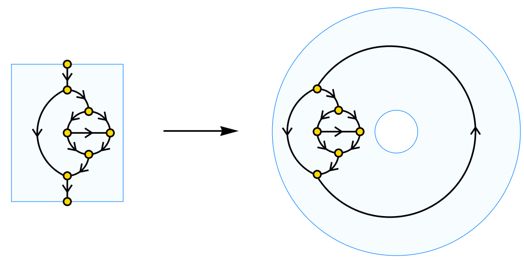


Figure 2.1.1. Annular strand diagram for x_1 obtained by closing its strand diagram

Definition 2.1.1. An **annular strand diagram** is a finite directed graph embedded in the annulus, with the following properties:

1. Each vertex is either a merge or a split
2. Every directed cycle winds counterclockwise around the central hole. △

Although we have defined an annular strand diagram as a directed graph, it can have **free loops**, which are directed cycles without any vertices. From property (2) in Definition 2.1.1, it follows that every free loop winds counterclockwise around the central hole of the annulus. Because free loops do not have end vertices, they are not allowed to be present in directed graphs, however, they can exist in annular strand diagrams.

2.2 Reductions

Annular strand diagrams can be reduced using the three reduction moves shown in Figure 2.2.1. The third move merges two consecutive free loops with no vertices in the region

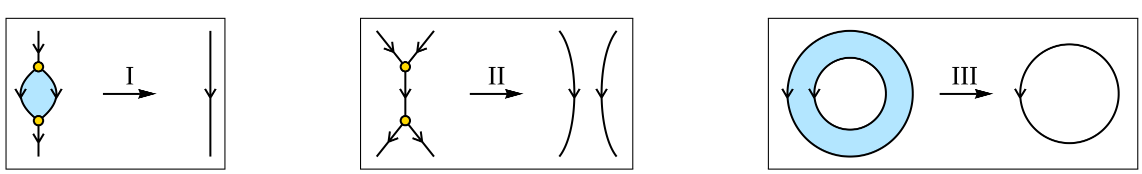


Figure 2.2.1. The reduction moves for annular strand diagrams. A type I move is allowed when the shaded region is a topological disk. A type III move is allowed when the space between the two free loops is a topological annulus containing no vertices (image taken from [3]).

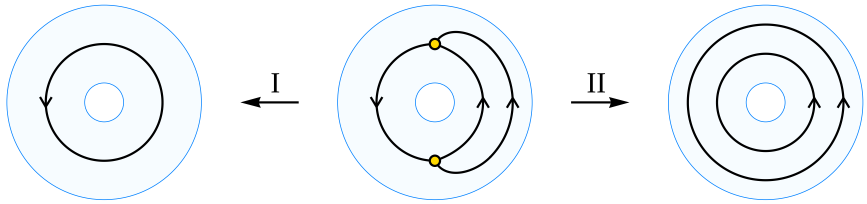


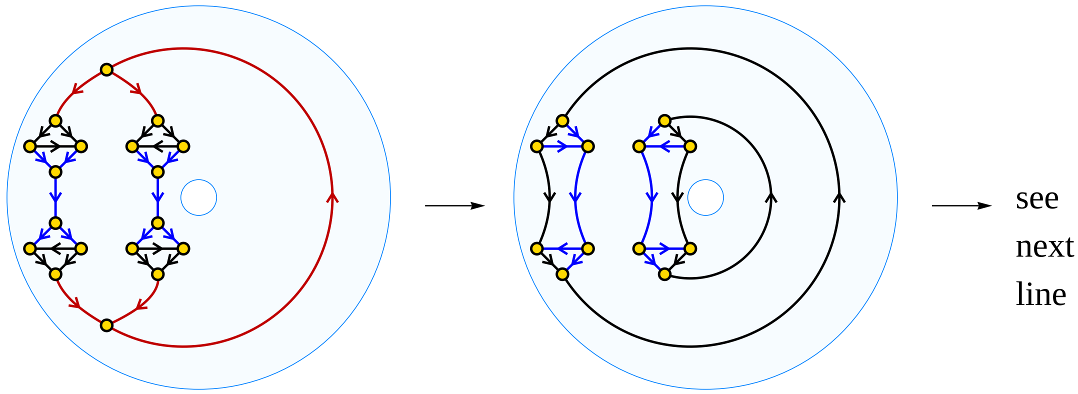
Figure 2.2.2. A type I move reduces the central annular strand diagram to a free loop, but a reduction II creates two concentric free loops. A reduction III on the diagram in the right merges the two free loops, preserving unique normal forms (image taken from [3]).

between them into one free loop. This move is required to make reductions of annular strand diagrams confluent so that every annular strand diagram reduces to a unique reduced annular strand diagram, as shown in the following example.

Example 2.2.1. The central annular strand diagram in Figure 2.2.2 is subject to both type I and type II moves, but the two moves produce different annular strand diagrams. A type III move reconciles the results, ensuring that the central annular strand diagram reduces to a unique normal form. ◊

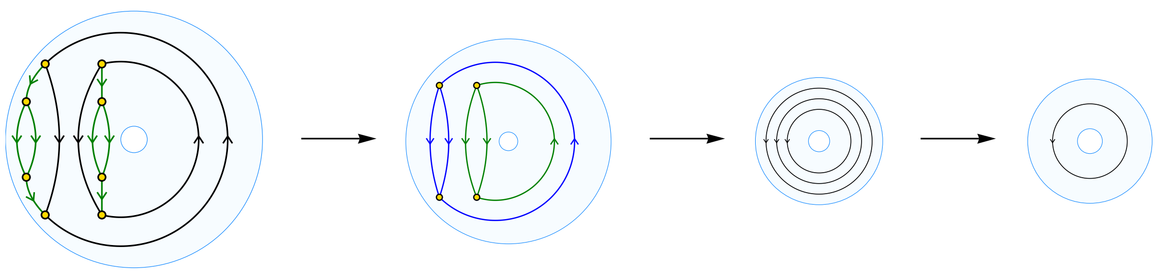
Example 2.2.2. Figure 2.2.3 shows an annular strand diagram and the reductions applied to it until it reduces to a free loop. Notice that a type II move on (a) splits the connected annular strand diagram into two connected components. The reductions applied to (d) demonstrate that both type I and type II moves in annular strand diagram can result in the formation of free loops. ◊

At this point we are prepared to comprehend the solution to the conjugacy problem in F proposed by [3]. The solution, summarized in Theorem 2.2.3, involves reducing annular strand diagrams and checking for isotopy.



(a) an annular strand diagram,

(b) splits into two components after some type II moves,



(c)

(d)

(e) free loops

(f) reduced

Figure 2.2.3. Reducing an annular strand diagram. In the pictures, the green regions are subject to type I moves, and the red and blue regions are each subject to type II moves. The type II move on the red region in (a) breaks the connected annular strand diagram into two connected components.

Theorem 2.2.3. (Belk and Matucci). *Let g and h be elements of Thompson's group F. Let G and H be strand diagrams for g and h, and let G' and H' be the reduced annular strand diagrams obtained by closing G and H and then reducing. Then g and h are conjugate if and only if G' and H' are isotopic.*

For the proof of this theorem, the reader is referred to [3]. As will be shown in the next chapter, our algorithm for the conjugacy problem in F fundamentally builds on this theorem. The rest of this chapter provides insights into the structure of annular strand diagrams and explains the strategies we employ to keep track of the configuration of an annular strand diagram before, during, and after it is reduced.

2.3 Concentric Components

Reduced annular strand diagrams can have multiple connected components. In this section, we discuss how annular strand diagrams can decompose into two or more connected components during reductions, showing that directed cycles and type II reduction moves are directly responsible, and we also provide further insights into the structure of these connected components.

Proposition 2.3.1. *In any strand diagram, there is a directed path from every vertex to the sink.*

Proof. Let S be a strand diagram. Assume that there exists a vertex $v \in S$ such that there is no directed path from v to the sink. Construct a directed path P by arbitrarily following an outgoing edge from v to a vertex $v_1 \in S$ which cannot be the sink. Expand P by following an arbitrary outgoing edge of v_1 to another vertex $v_2 \in S$ which is again not the sink. Create an infinite path by repeatedly keep expanding P in this manner. Because S has finitely many vertices, the infinite path P must have a cycle. But S is acyclic, which contradicts the assumption that P has a cycle. Hence, P must be a directed path from v to the sink. $\square$

Proposition 2.3.1 leads immediately to the following corollary.

Corollary 2.3.2. *All strand diagrams are connected.*

Since closing any strand diagram only "glues" the source and the sink, it follows that closing does not split a strand diagram into two components. Thus, any annular strand diagram produced by closing a strand diagram is connected. Note that closing the strand diagram for the identity produces a free loop.

However, a sequence of reductions can sometimes change the number of connected components in an annular strand diagram. Observe that closing any strand diagram creates a directed cycle, and an edge which is an output of a merge and an input to a split. Therefore any annular strand diagram produced by closing a strand diagram is immediately subject to a type II reduction move. As we have seen in Figure 2.2.3, reductions on annular strand diagrams can result in the formation of multiple components as well as the merging of multiple connected components. In particular, a type II move can split a connected annular strand diagram into two components if the edge from the merge to the split involved in the reduction is the intersection of two directed cycles, as shown in Figure 2.3.1. Such a reduction disconnects the strand diagram, making the cycles disjoint, and creating two components. Furthermore, a reduction III move can merge two connected components into one. Because reducing an annular strand diagram can modify the number of connected components, we must keep track of the order of these components in order to correctly construct the reduced annular strand diagram.

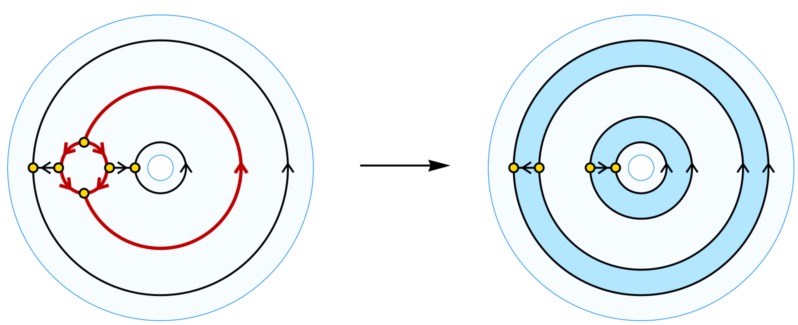
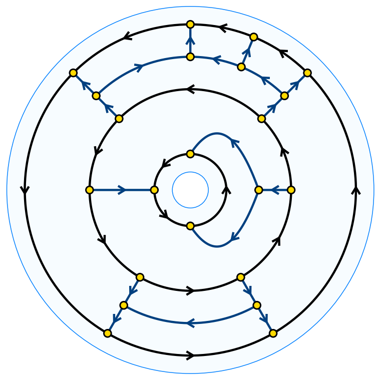
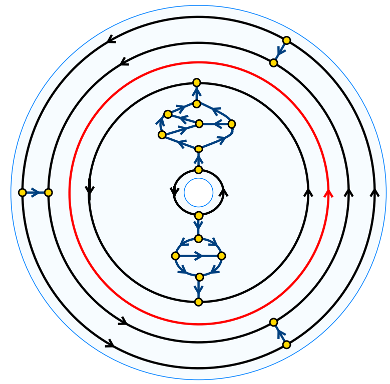


Figure 2.3.1. A reduction II move causing a connected component to split into two (each shaded region represents a component).





$\mathcal{A}_1$ (connected) $\mathcal{A}_2$ (disconnected)

Figure 2.3.2. Two reduced annular strand diagrams ($\mathcal{A}_1$ has been taken from [3]).

We will use the following terminologies to describe certain cycles in annular strand diagrams:

- **split loop** - a directed cycle which has only splits
- **merge loop** - a directed cycle which has only merges

Example 2.3.3. Figure 2.3.2 shows two reduced annular strand diagrams $\mathcal{A}_1$ and $\mathcal{A}_2$. $\mathcal{A}_1$ is a connected graph having a merge loop as the inner and the outermost directed cycles, and a split loop between them. Consecutive directed cycles in $\mathcal{A}_1$ are connected by trees (colored blue). $\mathcal{A}_2$ has three components, where the second component is a free loop. The two directed cycles in the first component in $\mathcal{A}_2$ are connected by reduced strand diagrams (colored blue). Excluding the free loop, the directed cycles (colored black) in both $\mathcal{A}_1$ and $\mathcal{A}_2$ are either merge loops or split loops. $\Diamond$

Proposition 2.3.4. *In any reduced annular strand diagram $\mathcal{R}$:*

1. *Each component $\mathcal{C}$ has at least one directed cycle.*
2. *Each directed cycle is either a split loop, a merge loop, or a free loop.*
3. *No directed cycle intersects another directed cycle or itself.*
4. *Every component surrounds the inside hole of the annulus.*

Proof.

1. Observe that if $\mathcal{C}$ is a free loop, then it has a directed cycle. Now assume that $\mathcal{C}$ is not a free loop. Observe that each vertex in $\mathcal{C}$ has at least one output edge. In $\mathcal{C}$, construct an infinite path P using the same infinite path construction method discussed in Proposition 2.3.1. Because C has finitely many vertices, eventually in P a vertex v_n will be repeated. Then the path P has a directed cycle starting at v_n.

2. Assume that there exists a directed cycle with merges and splits. Then there must be at least one merge with an outgoing edge to a split. It follows that $\mathcal{R}$ can still undergo a reduction II move, which is a contradiction since $\mathcal{R}$ is already reduced.

3. Assume that two directed cycles intersect. Then the cycles must intersect at a merge and then split apart, leading to a reduction II move, which is a contradiction since $\mathcal{R}$ is already reduced. By using the same argument, we can show that a directed cycle cannot intersect itself.

4. Because every component has at least one directed cycle D, and since every directed cycle annular strand diagram winds counterclockwise around the inside hole (see part (2) of Definition 2.1.1), it follows that D must go around the inside hole of the annulus. Therefore, each component surrounds the inside hole. □

Therefore, any reduced annular strand diagram has a finite number of disjoint directed cycles winding counterclockwise around the central hole. Furthermore, consecutive directed cycles in a connected component are held together by acyclic directed planar subgraphs, consisting of splits and merges.

Proposition 2.3.5. *In each component $\mathcal{C}$ with two or more directed cycles in a reduced annular strand diagram, these cycles alternate concentrically between split loops and merge loops.*

Proof. Let L and L' be two concentric directed cycles in C. Assume without loss of generality that L is a split loop in $\mathcal{C}$. Observe that for each split v in L, there exists an outgoing edge that is not part of L. Since L and L' are concentric and since they are present in the same component C, there must be at least one edge e emanating from L into the annular region between L and L'. Using the split $v \in L$ which has the edge e as an output, construct an infinite path P starting at e using the same infinite path construction method discussed in Proposition 2.3.1. Because $\mathcal{C}$ has finitely many vertices, eventually in P a vertex v_n will be repeated. It follows that the path P has a directed cycle $D \neq L$ starting at v_n. Either $D = L'$ or L' is between L and D, and since L and L' are concentric, in both cases it follows that P must have hit at least one vertex $v' \in L'$. But then v' has another edge that is part of the directed cycle L'. It follows that v' is a merge, and therefore L' must be a merge loop.

The case when L is a merge loop can be proved using a similar reasoning as above, and we omit its details. □

We have obtained deeper insights into the structure of connected components in annular strand diagrams. In an annular strand diagram, the connected components are in concentric order in the annulus, and each component is itself an annular strand diagram. A component with exactly one directed cycle must be a free loop. For each component excluding the free loop:

1. there exist at least two directed cycles,
2. the entire component lies in the annular region between its innermost and outermost directed cycles.

This means that given an annular strand diagram $\mathcal{A}$ in the annulus, any straight line L drawn from the inside of the annulus to the outside first intersects an edge that is part of the innermost directed cycle in $\mathcal{A}$, and L crosses the remaining directed cycles in concentric order until it reaches the outside of the annulus. The last edge crossed by L is an edge in the outermost directed cycle in $\mathcal{A}$.

2.4 The Cutting Path

In this section, we describe our strategy to keep track of the ordering of components in an annular strand diagram. Our approach involves having a dynamic ordered list of some of the edges in the annular strand diagram, and for this purpose we use the **cutting path**.

Definition 2.4.1. A **cutting path**, in an annular strand diagram A, is a directed path from the inside hole of the annulus all the way to the outside such that it crosses at least one edge of A, and the edge crossing rules in Figure 2.4.1 hold. $\triangle$

Theorem 2.4.2. *Each annular strand diagram obtained by closing a strand diagram has a cutting path, and its corresponding reduced annular strand diagram also has a cutting path.*

Proof. Let S be a strand diagram and let $\mathcal{A}$ be the annular strand diagram obtained by closing S. It follows that $\mathcal{A}$ has a cutting path P which crosses the edge created during closing (see Figure 2.4.3). Perform all possible reductions on $\mathcal{A}$, and during each

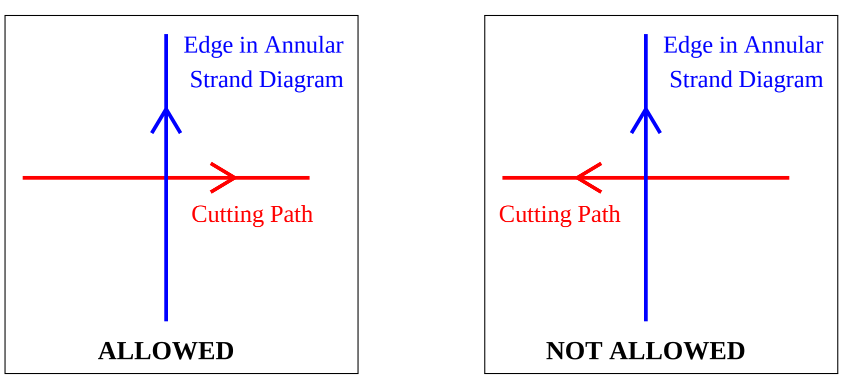


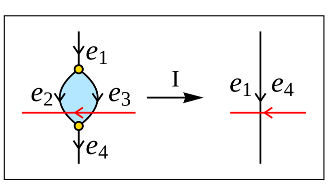
Figure 2.4.1. For a cutting path, the edge crossing in (a) is allowed and (b) is not allowed

reduction, if P goes through a region of reduction as shown in Figure 2.4.2, then update P as described below.

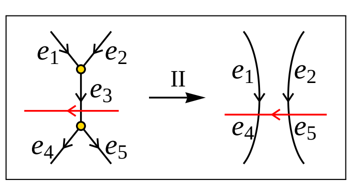
- **Reduction I:** P does not need to be updated if it goes through e_1 or e_4 since the reduction will merge these edges accordingly. If P goes through the shaded disk, it must enter the disk by crossing e_3 and leave by crossing e_2. In this case, replace e_3 and e_2 in P with e_1 (same as e_4).
- **Reduction II:** Again P does not require an update if it crosses all the other edges except e_3 since these edges will be modified by the reduction itself. If P goes through e_3, then after performing the reduction, it must go through the edge e_1 (same as e_5) before the edge e_2 (same as e_4).
- **Reduction III:** Replace the two free loops in P with the new free loop.

Note that it is possible that P crosses more than once an edge which is involved in a reduction, but this is not a problem if we use the same rule to update edges in P for all such crossings. Observe that updating P during a reduction move does not violate the edge crossing rules for a cutting path. Therefore, when $\mathcal{A}$ is reduced, it has the cutting path P. □

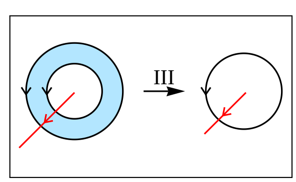
Example 2.4.3. Figure 2.4.3 shows the edges in an annular strand diagram that the cutting path intersects before and after the annular strand diagram is reduced. ◊



(a) reduction I

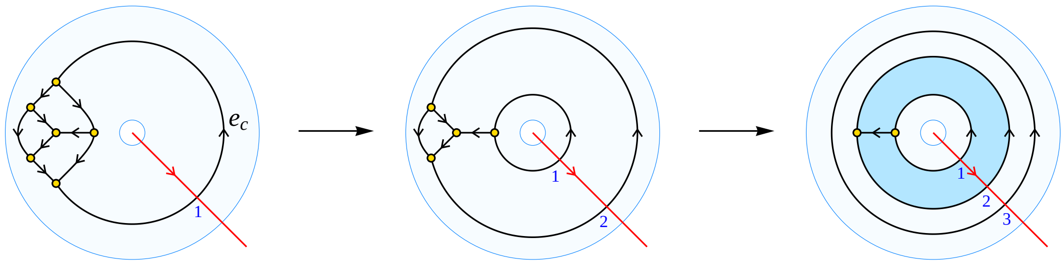


(b) reduction II



(c) reduction III

Figure 2.4.2. Update of the cutting path for each reduction move.



(a) (b) (c)

Figure 2.4.3. Status of a cutting path (a) after closing a strand diagram (crosses edge e_c), (b) after performing a type II reduction, and (c) after the annular strand diagram is reduced. The numbers denote the order of the edges in the cutting path

The following proposition gives an important insight into how a cutting path can identify the concentric order of components in a reduced annular strand diagram.

Proposition 2.4.4. *Let $C_1, ..., C_M$ be the components of a reduced annular strand diagram $\mathcal{A}$ in concentric order. Let $e_1, ..., e_n$ be the sequence of edges crossed by a cutting path P. Then $e_1, ..., e_n$ consists of one or more edges from C_1 followed by one or more edges from C_2 and so forth, ending with one or more edges from C_M.*

Proof. Observe that the first edge that P intersects must be part of the innermost directed cycle in $\mathcal{A}$. It follows that $e_1 \in C_1$.

Because all the directed cycles in $\mathcal{A}$ are directed counterclockwise, if P has already crossed an edge in such a directed cycle d, then it cannot cross an edge in d in the opposite direction due to the edge crossing rules in Figure 2.4.1. As a result, P must go through every directed cycle exactly once, and P must cross the directed cycles in $\mathcal{A}$ in concentric order from the central hole to the outside of the annulus.

Every component has an innermost cycle and an outermost cycle (which are the same for a free loop). Therefore, for each component C_i, the first edge P crosses is part of the innermost cycle, and P keeps crossing edges in C_i until it crosses the outermost cycle. Once P leaves the outermost cycle of C_i, it cannot get back into C_i because of the edge crossing rules for a cutting path. Therefore, P is not allowed to re-enter a connected component which it has already entered once, and this forces P to continue to enter the concentric components in order until it exits the annulus. □

It follows that by keeping track of:

- the order of the edges the cutting path meets, and
- the connected components to which these edges belong,

the cutting path allows us to identify the concentric ordering of components in any annular strand diagram, with very little computation and storage. Knowledge of the sequence of connected components is essential for checking whether two reduced annular strand diagrams are isotopic.

2.5 Isotopy of Reduced Annular Strand Diagrams

As proven by Belk and Matucci [3], two elements of F are conjugate if and only if their corresponding reduced annular strand diagrams are isotopic (see Theorem 2.2.3). Hence, our solution algorithm for the conjugacy problem in F must be able to deduce whether two reduced annular strand diagrams are isotopic. In this section, we describe the term isotopy in the context of annular strand diagrams.

Recall from Definition 1.2.12, in an embedding of a directed graph, the rotation system is the family of the counterclockwise order of the edges connected to each vertex. For an annular strand diagram, knowing the counterclockwise order of the edges connected to a merge is the same as knowing the left and right inputs, and similarly the left and right outputs in the case of a split. This constitutes a rotation system for annular strand diagrams.

The following theorem has a significant role in the design of our algorithm for the conjugacy problem in F. However, its proof is beyond the scope of this project, so we provide an outline of the proof and leave it to the reader to figure out the missing details.

Theorem 2.5.1. *Two connected annular strand diagrams $\mathcal{A}_1$ and $\mathcal{A}_2$ are isotopic in the annulus if and only if there exists a directed graph isomorphism between them that preserves the rotation system.*

Sketch of Proof. Observe that the annulus is a sphere with two holes in it:

1. the inner hole at (0,0), and
2. the outer hole at ∞.

It is not difficult to see that two directed graphs G and H are isotopic in the annulus if and only if:

1. G and H are isotopic on the sphere, and
2. the inner and outer holes respectively lie in corresponding faces in G and H.

When G and H are connected annular strand diagrams, observe that their faces containing inner and outer holes are the only faces whose boundaries are directed cycles. Moreover, the directed cycle surrounding the inner hole goes counterclockwise and the one surrounding the outer hole goes clockwise (since it appears counterclockwise on the plane).

Therefore, $\mathcal{A}_1$ and $\mathcal{A}_2$ are isotopic in the annulus if and only if they are isotopic on the sphere. But by Corollary 1.2.15, we know that two connected annular strand diagrams are isotopic on the sphere if and only if there exists a directed graph isomorphism that preserves the rotation system. $\square$

Corollary 2.5.2. *If two connected annular strand diagrams $\mathcal{A}_1$ and $\mathcal{A}_2$ represent different embeddings of the same directed graph, and they agree for:*

1. *every merge on the left and right inputs, and*

2. every split on the left and right outputs,

then $\mathcal{A}_1$ and $\mathcal{A}_2$ are isotopic.

In other words, isotopy preserves the direction of the edges in addition to isomorphism. For instance, in the case of two isotopic annular strand diagrams G and H, the left output of a vertex in G corresponds to the left output of the corresponding vertex in H mapped by the isomorphism. Furthermore, the counterclockwise order of input(s) and output(s) of a vertex in G is the same as the counterclockwise order of input(s) and output(s) of the corresponding vertex in H.

Example 2.5.3. Figure 2.5.1 shows three reduced annular strand diagrams $\mathcal{A}_1, \mathcal{A}_2$, and $\mathcal{A}_3$. Observe that $\mathcal{A}_2$ can be obtained by applying a 180° rotation to $\mathcal{A}_1$ on the annulus. Vertex $v_1 \in \mathcal{A}_1$ would fall over vertex $v_2\mathcal{A}_2$, and a depth first search from these corresponding vertices will confirm the presence of an isotopy. On the other hand, $\mathcal{A}_3$ is not isotopic to the $\mathcal{A}_1$ or $\mathcal{A}_2$. Observe that the left output from the left split in the tree in $\mathcal{A}_3$ has target v_3, which has an outgoing edge to the outermost directed cycle. However, the left output from the left split in the tree in $\mathcal{A}_1$ has a target vertex, which has an outgoing edge to the innermost directed cycle. Similarly, $\mathcal{A}_3$ is not isotopic to $\mathcal{A}_2$. ◊

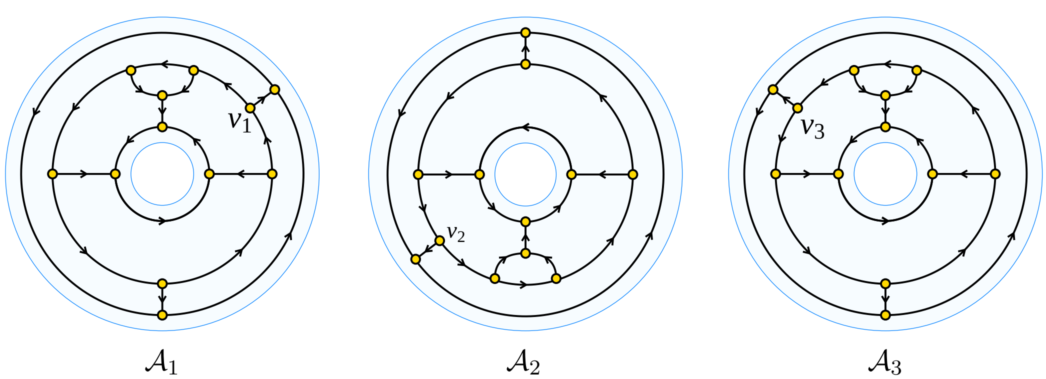


Figure 2.5.1. Three reduced annular strand diagrams in which $\mathcal{A}_1$ and $\mathcal{A}_2$ are isotopic.

Corollary 2.5.4. *Given two reduced annular strand diagrams G and H, where $C_G = \{g_1, g_2, ..., g_n\}$ and $C_H = \{h_1, h_2, ..., h_m\}$ are the sets of connected components in G and H respectively in concentric order from the inside of the annulus to the outside, then G and H are isotopic if:*

1. *$n = m$, and*
2. *$g_i \in C_G$ and $h_i \in C_H$ are isotopic for each $i \in \{1, 2, ..., n\}$.*

3

Algorithm for the Conjugacy Problem in F

In this chapter, we provide a comprehensive description of our algorithm for the conjugacy problem in F, the biggest contribution in our research. We discuss our implementation called **ConjugacyF**, with particular emphasis on key notes in each major step of the algorithm, and analyze our solution algorithm to show that theoretically it executes in $O(n)$, where n is the sum of the lengths of the input words. We provide a tested, bug-free Java implementation of ConjugacyF, showing evidence of correctness using our results, and describe the user interface towards the end of this chapter.

The theoretical implementation reduces the problem of checking whether two reduced annular strand diagrams are isotopic to the problem of determining whether two planar graphs are isomorphic. It uses the $O(|V|)$ algorithm proposed by [10] for the isomorphism problem in planar graphs, where $|V|$ is the number of vertices in the input planar graphs. However, as stated by the authors in [10], their algorithm is mainly theoretical and too complicated to be implemented. Due to this reason, ConjugacyF directly determines whether two reduced annular strand diagrams are isotopic, and as a result, it executes in $O(n^2)$.

Theorem 3.0.5. *Given two input words w_1 and w_2 in $\langle x_0, x_1\rangle$ representing elements of F, the proposed algorithm for the conjugacy problem decides whether w_1 and w_2 are conjugate in $O(n)$, where $n = |w_1| + |w_2|$.*

The rest of this chapter proves this theorem. We emphasize that this theorem uses the $O(|V|)$ algorithm proposed by [10] for the isomorphism problem in planar graphs.

3.1 Algorithm Overview

In this section, we describe the major steps in the flow of the algorithm. A flowchart for the algorithm is shown in Figure 3.1.1, and the steps are highlighted below:

1. The algorithm takes as input two words $w1$ and $w2$ in the generating set $\langle x_0, x_1\rangle$. Hence, an input word is a string with a sequence of characters from the set $\{x_0, x_1, x_0^{-1}, x_1^{-1}\}$.
2. The strand diagrams for these words are constructed, and they are closed to obtain the corresponding annular strand diagrams (Section 3.3).
3. The annular strand diagrams are reduced (Section 3.4).
4. The connected components in each reduced annular strand diagram are labeled and stored in a list in concentric order from the inside of the annulus to the outside (Section 3.5).
5. For each reduced annular strand diagram, the components are encoded to planar graphs, which are stored in lists in the same concentric order as that of the components (Section 3.6).
6. Then the following procedure determines whether $w1$ and $w2$ are conjugate:

   Let $P_{w1} = \{g_1, g_2, ..., g_n\}$ and $P_{w2} = \{g'_1, g'_2, ..., g'_m\}$ be the lists of planar graphs for the elements $w1$ and $w2$ respectively. If $n \neq m$, then $w1$ and $w2$ are not conjugate.

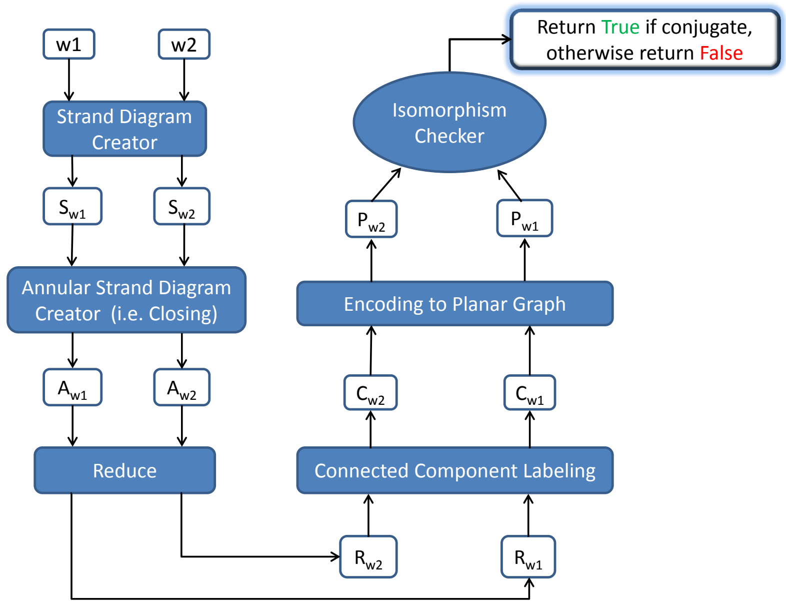


Figure 3.1.1. Overview of the solution algorithm for the conjugacy problem in F

Otherwise, for each $i \in \{1, 2, ..., n\}$, the algorithm checks whether g_i and g'_i are isomorphic. If all such g_i and g'_i are isomorphic, then $w1$ and $w2$ are conjugate, and not conjugate otherwise (Section 3.8).

In accordance with the sections mentioned above, the other major topics in this chapter are organized as follows:

Section 3.2 describes the data structure; Sections 3.6 - 3.7 prove an important theorem relating isotopy of strand diagrams to isomorphism of planar graphs; Section 3.9 describes the subroutine in our Java program that compares the rotation systems of annular strand diagrams to determine whether they are isotopic, proving that this subroutine causes the running time of ConjugacyF to be quadratic; Section 3.10 analyzes the results we obtained using ConjugacyF and provides evidence to show that the implementation is bug-free; and finally Section 3.11 describes the software we created for the conjugacy problem in F using Java, with details on how to use the interface.

Furthermore, any section where part of the solution algorithm is discussed also presents an analysis of that part of the algorithm towards proving that the overall algorithm is linear in the length of the input words.

3.2 The Data Structure

3.2.1 Background: Doubly Linked Lists

To minimize the running time of the solution algorithm, on several occasions we use the doubly linked list data structure. For a broader description of doubly linked lists, the reader is referred to [17].

Definition 3.2.1. A **doubly linked list** is a data structure containing sequentially linked nodes, each of which has:

- a data field holding a value,
- a previous field that refers to the previous node, and
- a next field that refers to the next node

in the sequential ordering of nodes in the data structure. △

Conventionally, the previous field of the beginning node and the next field of the ending node are set to null. Discussed below are certain methods that our solution algorithm will invoke on a customized doubly linked list D. Note that all these methods take constant time.

- add(<Type> a): creates a new node N containing a, and attaches N at the end of D.
- addAfter (<Type> a, <Type> b): creates a new node N containing b, inserts N between the node p holding a and the node p+1, and creates new links between p,N, and p+1.
- remove(Node N): removes N from D and and links its previous node to its next node.

- `remove(<Type> a)`: inquires the `node` field of `a` to find the node in which `a` belongs in `D`. Then removes this node, and links its previous node to its next node.
- `replace(<Type> a, <Type> b)`: creates a new node `N` containing `b`, substitutes the node holding `a` with `N`, thus replacing the data `a` with the data `b` in `D`.

Note that it is important that the above operations each take constant time for ConjugacyF to execute in linear time. However, our Java implementation of ConjugacyF uses the in-built LinkedList data structure for Java, which may not allow these operations in constant time.

We now describe the data structure (as a Java model) for representation and manipulation of strand diagrams and annular strand diagrams. Recall that strand diagrams for elements of F have four kinds of vertices:

- a *merge*, which has two parents and one child
- a *split*, which has one parent and two children
- a *source* of degree 1, with an outgoing edge to a split, and
- a *sink* of degree 1, with an incoming edge from a merge

The source and the sink are involved only in concatenations (see Section 1.4.1) and closing (see Section 2.1) . Annular strand diagrams have only merges and splits as vertices.

Note 3.2.2. All the linked lists in our data structure are doubly linked lists with constant time access. ◊

3.2.2 Class: Edge

1. Each `Edge` object is directed from a `source` vertex to a `target` vertex.
2. Each `Edge` object has a unique `ID`.
3. The field `class` is an array of two integers that records the **class** to which the edge belongs (see Section 3.6 for discussion of "class"). In this array, the first integer

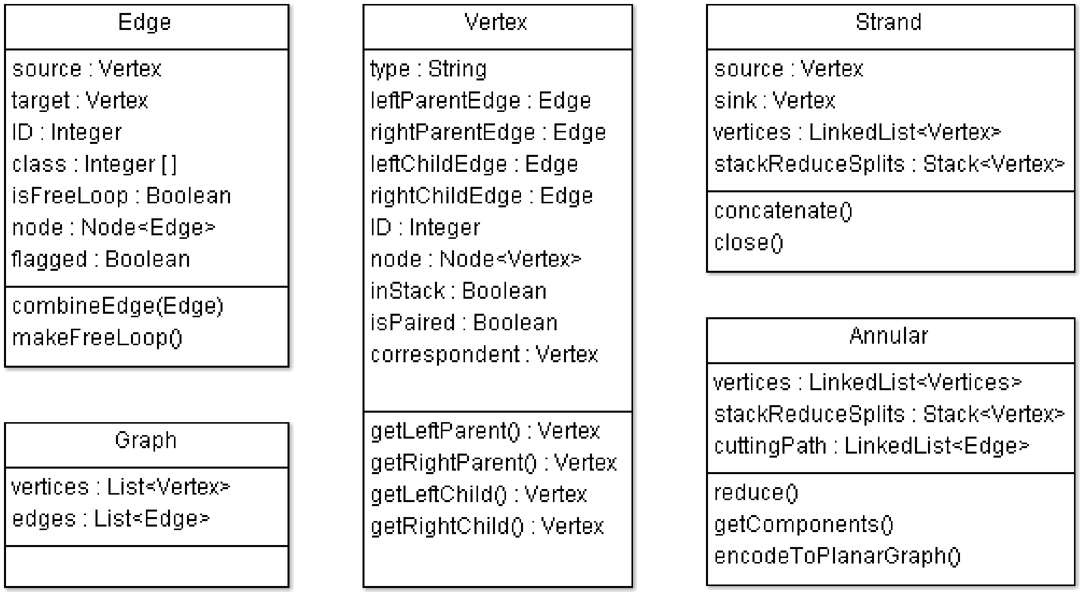


Figure 3.2.1. The Java model of the data structure used for the solution to the conjugacy problem in F. Note that all the linked lists are doubly linked.

denotes the input type and the second integer denotes the output type for the edge. These integers can be the following:

- $0 \to$ free loop
- $1 \to$ left input or left output
- $2 \to$ right input or right output

4. ConjugacyF involves insertion of edges into a linked list called `cuttingPath`, which stores a dynamic subsequence of edges in the order in which they meet a particular cutting path (see Section 3.2.5). The field `node` for an `Edge` object stores the container node that holds the edge, if present, in `cuttingPath`, otherwise `node` is set to `null`.

5. The field `flagged` is required to mark edges during connected component labeling of reduced annular strand diagrams (discussed in Section 3.5).

6. Invoking the method `makeFreeLoop()` turns the edge into a free loop by setting `isFreeLoop` to `true`, both elements of `class` to 0, and making the `source` and the `target` vertices `null`.

7. Given an edge e_1 with source vertex s, the `combineEdge()` method (see Algorithm 7 in the Appendix takes another edge e_2 with target vertex t as input, and then merges the two edges. As a result, both e_1 and e_2 are the same edge with source vertex s, target vertex t, `class[0]` from old e_1, `class[1]` from old e_2, and:

   - if both their `node` fields are `null`, then a *fake* `Node` having an empty `data` field is constructed for e_1, and the `node` for e_2 is also assigned this `Node`.
   - if neither of their `node` fields are `null`, then the `node` for e_2 is destroyed first, and then assigned the `node` for e_1.
   - if exactly one of the edges has a `node` that is not `null`, then the other edge has its `node` set to the former's `node`.

Note 3.2.3. In the description of the solution algorithm, for any `Edge` object e, if e.`node` is `null`, then e.`node` is an unassigned or it is a fake `Node`. $\Diamond$

3.2.3 Class: Vertex

1. We use the field `type` to denote the vertex type, which can be any string in the set {source, sink, merge, split}. Note that each `Vertex` object has four `Edge` objects associated with it. Using the `type` field, we can safely decide which of these `Edge` objects are allowed for a vertex, as shown in Table 3.2.1.

   By convention, a vertex having a lone input edge has the edge stored in `leftParentEdge`, and a vertex having a lone output edge has the edge stored in `leftChildEdge`. By inquiring the associated `Edge` objects, the children and parents of any vertex can be found, which are provided by the getter methods.

2. Each vertex has a unique `ID`, required for creating and copying strand diagrams.

3. In (annular) strand diagrams, vertices will be stored in a linked list. In order to remove vertices from this linked list in constant time during reductions, the container node in which a vertex belongs is stored in the `node` for `Vertex` objects.

4. As will be shown later, during reductions, split vertices may be put into a stack called `stackReduceSplits` in the `Annular` class, and the field `inStack` is used to tell whether a vertex is currently on this stack.

5. During the isotopy check discussed in 3.9, each vertex in an annular strand diagram will have its `isPaired` field set to `true` when its corresponding vertex in the other annular strand diagram has been assigned, and the `correspondent` field is used to hold this corresponding vertex.

Note 3.2.4. The `Vertex` data structure preserves the counterclockwise order of the edges since it keeps track of the left and right parents of a merge, and similarly the left and the right children of a split. ◊

3.2.4 Class: Graph

The `Graph` data structure is used to hold planar graphs that are generated from reduced annular strand diagram components using the encoding algorithm (discussed in Section 3.6). A list of the vertices and a list of the undirected edges are sufficient to represent planar graphs. In the solution algorithm, `Graph` objects will be compared to determine whether

	leftParentEdge	rightParentEdge	leftChildEdge	rightChildEdge
source	✗	✗	✓	✗
target	✓	✗	✗	✗
merge	✓	✓	✓	✗
split	✓	✗	✓	✓

Table 3.2.1. The `Edge` objects associated with certain vertex types.

they are isomorphic in order to decide whether two reduced annular strand diagrams are isotopic (see Theorem 3.7.1).

3.2.5 Class: Strand

This data structure represents elements of F in strand diagram forms.

1. We construct a `Strand` object, representing a strand diagram, from its input *word*, which is a string in the generating set $\langle x_0, x_1 \rangle$.
2. We store the `source` and the `sink` vertices so that concatenations and closing can be performed in constant time.
3. As each `Vertex` object stores the counterclockwise order of edges connected to it, the field `vertices`, holding all the vertices in a strand diagram, has sufficient information to correctly construct the strand diagram.
4. `stackReduceSplits` is a stack that stores a list of split vertices during concatenations (discussed in Section 3.3).
5. The method `concatenate()` performs concatenation of two `Strand` objects (see Algorithm 8).
6. Invoking the method `close()` on a `Strand` object turns it into an `Annular` object that represents the corresponding annular strand diagram.

3.2.6 Class: Annular

This data structure is used to hold annular strand diagrams, which do not have a source or a sink vertex. In addition:

1. An `Annular` object is created by invoking the method `close()` on a `Strand` object. During the construction, the fields `vertices` (excluding the source and the sink) and `stackReduceSplits` are copied from the corresponding `Strand` object.

2. The field `cuttingPath` is a linked list that stores a subsequence of edges in a particular cutting path in the annular strand diagram.

3. The `reduce()` method performs all the possible reduction moves on an annular strand diagram, thereby reducing it.

4. The `getComponents()` method returns a concentrically ordered list of the connected components in the annular strand diagram. These connected components are also `Annular` objects.

5. The method `encodeToPlanarGraph()` (see Algorithm 10) encodes connected components to planar graphs, which are `Graph` objects.

Now that the data structure has been described, we present the algorithm for the conjugacy problem in Thompson's Group F in Algorithm 1. The proof of this algorithm's correctness comes from Theorem 2.2.3 and Theorem 3.7.1. We now begin a thorough discussion and analysis of this algorithm.

3.3 Strand Diagram Generation

This section covers Lines 1-3 of Algorithm 1. Recall that in the string denoting an input word, each character is an element from $\{x_0, x_1, x_0^{-1}, x_1^{-1}\}$. Given an input word as a string of size n, the `Strand` object for the word is constructed by:

1. going through each character in the input string from left to right,
2. creating a `Strand` object for each character encountered, and
3. concatenating these `Strand` objects.

The `Strand` object corresponding to each character in the input word is constructed by creating at most 18 vertices and edges (see Theorem 1.4.5, note that each character has a unit length). As shown in Algorithm 8, the method `concatenate()` performs the

Input: `String` w_1`,` `String` w_2
Output: Whether w_1 and w_2 are conjugate: `true` or `false`

```
for w in {w1, w2} do
    // generate strand diagram from word w
    Strand sd = new Strand(w)
    Annular asd = sd.close() // obtain annular strand diagram
    asd.reduce() // discussed in Algorithm 2
    List<Annular> components = asd.getComponents(asd.cuttingPath)
    P_w = new List<Graph>()
    for c in components do
        P_w.add(c.encodeToPlanarGraph()) // see Algorithm 10
    end
end
if P_w1.size() ≠ P_w2.size() then
    return false
for i = 0 → P_w1.size() − 1 do
    Graph p1 = P_w1.get(i)
    Graph p2 = P_w2.get(i)
    if !(isIsomorphic(p1, p2)) then
        // the linear algorithm proposed in [10]
        return false

end
return true
```

Algorithm 1: Algorithm for the solution to the conjugacy problem in F.

concatenation of two strand diagrams in constant time. Because step (1) runs n times, it follows that construction of the strand diagram takes $O(n)$.

The corresponding `Annular` object is then produced by running the method `close()`, which merges the source and the sink, and therefore closing happens in constant time. This concludes that up to annular strand diagram generation, Algorithm 1 takes $O(n)$.

Note 3.3.1. The edge e_c, created by closing, is immediately added to the linked list called `cuttingPath` that now represents a cutting path for the annular strand diagram (see (a) in Figure 2.4.3 for visualization), and e_c`.node` is assigned the `Node` that contains e_c in `cuttingPath`. Notice that e_c is the first edge added to `cuttingPath`. $\Diamond$

Figure 3.4.1. (a) Reduction I and (b) reduction II, with labeled edges and the split involved in the reduction labeled v. The green vertices can be splits or merges, they may not be distinct from each other or from the yellow vertices.

Note 3.3.2. During the creation and closing of a `Strand` object, the stack `stackReduceSplits` stores the list of split vertices at the point of concatenations (see Algorithm 8 in the Appendix, such as the vertex v_s in Figure 1.4.5. The reason for keeping this list is that, right after the concatenation, these split vertices are the only split vertices where a reduction (reduction II to be exact) is possible. Thus, `stackReduceSplits` allows us to know all possible locations of reductions upon creation of an annular strand diagram. $\Diamond$

3.4 Reducing

In this section we discuss Line 4, of Algorithm 1, that performs all possible reductions. This section assumes that the input word for the annular strand diagram has length n. Our algorithm to reduce annular strand diagrams is shown in Algorithm 2. Reduction I and reduction II are further described in Algorithm 3 and Algorithm 4 respectively. We also emphasize the necessity of efficiently check for reductions and updating the cutting path.

The major steps in reducing an annular strand diagram are discussed below:

1. We perform a stack based reduction to speed up the reduction step. Recall that the stack `stackReduceSplits` initially stores all the split vertices in the annular strand diagram that take part in a reduction, which is a reduction II (see Note 3.3.2).

2. Until this stack is empty, we continue to pop a split from the top of the stack, set its `inStack` field to false, and if the split is involved in a reduction I (Line 6,

```
   // Algorithm to reduce annular strand diagrams:
1  while !stackReduceSplits.isEmpty() do
2      Vertex split = s.pop()
3      split.inStack = false
4      Vertex lchild = split.getLeftChild()
5      Vertex parent = split.getLeftParent()
6      if lchild.type == "merge" and split.leftChildEdge == lchild.leftParentEdge and
       split.rightChildEdge == lchild.rightParentEdge then
7          reductionI(split) // See Algorithm 3
8      else if parent.type. == "merge" then
9          reductionII(split) // See Algorithm 4
10
11 end
   // Perform Reduction III: Merge consecutive free loops
12 Node current = cuttingPath.getFirst()
13 while current.next ≠ null do
14     if current.data.isFreeLoop and current.next.data.isFreeLoop then
15         current = current.next
16         cuttingPath.remove(current.previous)
17     else
18         current = current.next
19
20 end
```

Algorithm 2: The `reduce()` method for `Strand` objects. Note that `reductionI()` or `reductionII()` may push splits into `stackReduceSplits`.

Algorithm 2) or a reduction II (Line 8, Algorithm 2), we perform the appropriate reduction.

3. Notice that both reductions I and II happen around a split that is removed after the reduction, that is, the vertex v in Figure 3.4.1. After a reduction I or a reduction II is performed, nearby split vertices may be put into the stack (discussed in Section 3.4.1).

4. When the stack `stackReduceSplits` becomes empty, all the possible reductions I and II have been performed.

5. Since each of reduction I and reduction II removes two vertices, and since the total number of vertices in the original annular strand diagram is $O(n)$, it follows that the total number of these reductions is also $O(n)$.

6. Finally, we loop through all the edges in `cuttingPath` and merge concentrically adjacent free loops using reduction III. Since the number of edges is $O(n)$, it follows that the number of reduction III moves in the worst case is bounded by $O(n)$.

3.4.1 Keeping Track of Potential Future Reductions

We need to make sure that the number of checks for possible reduction I and reduction II moves is bounded by $O(n)$. An obvious algorithm to carry out all possible reductions is to:

1. go through all the vertices in the annular strand diagram, checking for reductions, and

2. whenever a reduction is possible, carrying it out, and

3. going to step (1) until no more reductions can be performed on the annular strand diagram.

In the worst case, this algorithm runs $O(n)$ times, and since there are $O(n)$ vertices initially that need to be checked for a possible reduction, the overall running time of this algorithm is $O(n^2)$.

This is why we perform a stack based reduction which ensures that the overall number of inquiries for possible reduction I and reduction II moves is bounded by $O(n)$. Notice that whenever a reduction I or a reduction II is performed, this can give rise to a new reduction at the split vertices around the current region of reduction. We efficiently keep track of these possible reductions by:

1. pushing into `stackReduceSplits` all the neighboring split vertices, which can be all of the green vertices in Figure 3.4.1.

2. setting the `inStack` field of any split to `true` when the split is put into the stack, and to `false` when it is removed from the stack. This is an optimization to decide in constant time whether a split that should be put on the stack is already in the stack, and to avoid adding duplicates to the stack. Note that this step only affects the overheads in the running time.

According to Figure 3.4.1, at most four vertices can be put on the stack due to a reduction I or a reduction II. Because the total number of reduction I and reduction II moves is $O(n)$, it follows that the total number of items added to the stack until the annular strand diagram is reduced is also $O(n)$. Hence, the number of checks for possible reduction I and reduction II moves until the strand diagram is reduced is $O(n)$.

3.4.2 Cutting Path Update and Free Loop Generation

Here we show that updating the cutting path until the annular strand diagram is reduced takes at most $O(n)$, and discuss how our algorithm correctly detects all free loops arising from reduction moves.

Note 3.4.1. `cuttingPath` does not store a cutting path, but stores an instance of a cutting path. This instance has only one occurrence of each edge that the cutting path crosses, and after the annular strand diagram is reduced, the sequence of edges in `cuttingPath` is sufficient to correctly determine the concentric order of components. Furthermore, the size of `cuttingPath` is bounded by $O(n)$. ◊

A cutting path can cross an edge multiple times, but for our purposes, it suffices to record only one crossing of each edge. Given a component C, if an edge $e_c \in C$ crosses the cutting path multiple times, all these crossings will happen:

```
// Perform Reduction I (see (a) in Figure 3.4.1 for visualization):
Input: Vertex v, the split vertex that will be removed after the reduction
Vertex p = v.getLeftParent()
Vertex gc = w.getLeftChild() // grand child of v
if p == w then
    e1.makeFreeLoop() // the condition for a free loop
else
    e4 = e1.combineEdge(e4)
    for nbr in {p, gc} do
        if nbr.type == "split" and !(nbr.inStack) then
            stackReduceSplits.push(nbr); nbr.inStack = true

    end
// Cutting path update:
if e3.node ≠ null then
    if e1.node ≠ null then
        cuttingPath.remove(e3)
    else
        cuttingPath.replace(e3, e1)
    cuttingPath.remove(e2)
vertices.remove(w); vertices.remove(v)
```

Algorithm 3: The `reductionI()` subroutine. Note that the `replace()` and `remove()` methods take constant time (see Section 3.2.1).

- after the crossing of an edge from the innermost directed cycle in C, and
- before the crossing of an edge from the outermost directed cycle in C.

Therefore, we can arbitrarily record only one such occurrence of e_c in `cuttingPath` and still have a sequence of edges in `cuttingPath` that identify components in order. Because the number of edges in the reduced annular strand diagram is bounded by $O(n)$, it follows that the maximum possible size of `cuttingPath` is also $O(n)$.

Moreover, this is necessary to perform reductions in constant time. As will be shown in this section, reductions modify edges, and if an edge is in `cuttingPath` multiple times, all its occurrences in `cuttingPath` need to be updated, which is undesirable.

Note 3.4.2. `cuttingPath` only needs to be updated if it contains an edge that is removed in a reduction, that is, any black edge crossed by the red edge in Figure 2.4.2. $\diamondsuit$

```
// Perform Reduction II (see (b) in Figure 3.4.1 for visualization):
if v.rightChildEdge == w.rightParentEdge then
    e2.makeFreeLoop() // a free loop is created on the right edges
else
    e5 = e2.combineEdge(e5)
if v.leftChildEdge == w.leftParentEdge then
    e1.makeFreeLoop() // a free loop is created on the left edges
else
    e4 = e1.combineEdge(e4)
for nbr in {e2.source,e2.target,e1.source,e1.target} do
    if nbr ≠ null and nbr.type == "split" and nbr.inStack == false then
        stackReduceSplits.push(nbr); nbr.inStack = true

end
if e3.node ≠ null then
    red2CuttingUpdate() // Cutting path update (see Algorithm 5)
vertices.remove(v); vertices.remove(w)
```

Algorithm 4: The `reductionII()` subroutine.

Recall that the other edges in Figure 2.4.2 that may be contained in `cuttingPath` prior to the reduction are modified during the reduction using the `combineEdge()` method, which updates `cuttingPath` accordingly.

In the case of a reduction I (see (a) in Figure 3.4.1 for visualization):

- a free loop is generated only if e_1 and e_4 are the same edge prior to the reduction (see Figure 2.2.2).
- `cuttingPath` does not need to be updated if it contains e_1 or e_4 because `combineEdge()` ensures that there is at most one container node in `cuttingPath` holding the edge created by merging e_1 and e_4 after the reduction.
- if `cuttingPath` contains e_3, then it must also have a node holding e_2 (see Figure 2.4.2). In this case, if e_1`.node` is `null`, then we replace the adjacent nodes containing e_2 and e_3 in `cuttingPath` with e_1. Otherwise, we remove e_3 and e_2 from `cuttingPath`.

Recall that the `node` field for an `Edge` object stores the node that contains the edge in `cuttingPath` (see Section 3.2.2). Therefore, given an edge e, the node in `cuttingPath` containing e can be accessed in constant time, and consequently, updating `cuttingPath` in reduction I (Lines 13 - 18 in Algorithm 3) takes a constant number of operations. It follows that reduction I takes constant time.

```
// see (b) in Figure 3.4.1 for visualization
boolean replaced = false
if e2.node == null then
    cuttingPath.replace(e3, e2)
    replaced = true
if e1.node == null then
    if replaced then
        cuttingPath.addAfter(e2, e1)
    else
        cuttingPath.replace(e3, e1)

else if !replaced then
    cuttingPath.remove(e3)
```

Algorithm 5: The `red2CuttingUpdate()` method, which updates `cuttingPath` during a reduction II. See Section 3.2.1 for description of the methods invoked on `cuttingPath`.

In the case of a reduction II (see (b) in Figure 3.4.1 for visualization):

- at most two free loops are created (see Figure 2.2.2) as shown in the cases below:

    1. $e_2 = e_5$, or
    2. $e_1 = e_4$ prior to the reduction.

- we do not need to update `cuttingPath` if it contains the edges e_1, e_2, e_4, or e_5 since `combineEdge()` will update it accordingly.

- However, `cuttingPath` requires an update if it contains the edge e_3. In this case, after the reduction the corresponding cutting path is expected to cross e_2 before it crosses e_1. We have the following possible cases when updating `cuttingPath`:

  1. If both e_1 and e_2 are not in `cuttingPath`, then we replace the node for e_3 in `cuttingPath` first with a node holding e_2 followed by a node holding e_1.
  2. If both e_1 or e_2 are already in `cuttingPath`, then we simply remove the node for e_3.
  3. If exactly one of e_1 or e_2 is not in `cuttingPath`, then we create a new node holding this edge and replace the node for e_3 with this node.

- Below we list three special cases where precaution need to be taken in updating `cuttingPath`. The cases correspond to Figure 3.4.2.

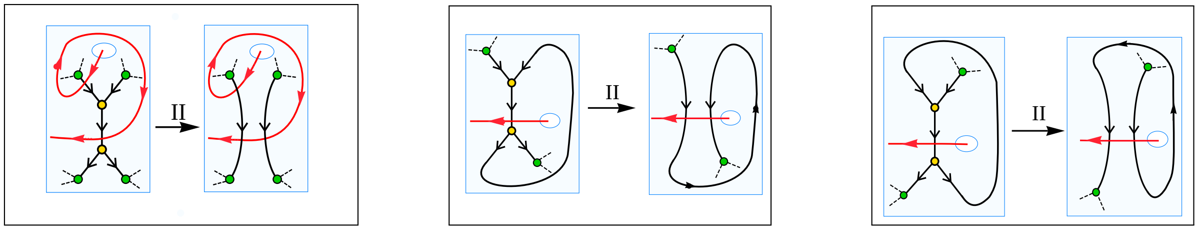
  (a) Prior to the reduction, the cutting path crosses e_1 before it crosses e_3: after the reduction, the node for e_1 must be contained in `cuttingPath` before the node for e_2.

  (b) `v.leftChildEdge = w.rightParentEdge`: Make sure that the same edge is not added twice in `cuttingPath`.

  (c) `v.rightChildEdge = w.leftParentEdge`: Again, do not add the same edge twice in `cuttingPath`.

Recall that the methods invoked on the doubly linked list `cuttingPath` each perform a constant number of operations. It follows that the method `red2CuttingUpdate()`, which updates the cutting path during a reduction II move, takes a constant number of operations. Thus, we can conclude that a single reduction II move executes in constant time.

At this point, we have shown that carrying out all the possible reductions I and II, that is, the `while` loop starting in Line 1 of Algorithm 2, take $O(n)$ in the worst case. Earlier in



(a) (b) (c)

Figure 3.4.2. Special cases for updating `cuttingPath` during a type II move. Refer to (b) in Figure 3.4.1 for edge and vertex labels.

this section, we showed that all the reduction III moves (the `while` loop starting in Line 13 of Algorithm 2) are bounded by $O(n)$. Therefore, the method `reduce()`, which is the algorithm to reduce an annular strand diagram, is bounded by $O(n)$.

3.5 Connected Component Labeling

This section describes how connected components are labeled in reduced annular strand diagrams, which happens in Line 5 of Algorithm 1. Because each connected component is a reduced annular strand diagram, it is represented using an `Annular` object. To construct a connected component, it suffices to start with an edge belonging to the component and performing a depth first search along its source and target to find all of the vertices in the component. Below we discuss the major points regarding the method `getComponents()`, which is comprehensively described in Algorithm 9 in the Appendix:

1. When the `Annular` object representing a connected component has no vertices but has a single edge, it means that the component is a free loop.

2. `getComponents()` takes as input `cuttingPath` after the annular strand diagram has been reduced. Recall that at this point `cuttingPath` is an ordered subsequence of a cutting path for the reduced annular strand diagram, containing at least one edge from each connected component.

3. Given an edge e in `cuttingPath`, the algorithm proceeds by obtaining the `source` vertex v of e, and then performing a depth first search on all the edges connected to v to discover the whole component that contains e. Any edge e' discovered during this search has its `flagged` field set to true so that if e' is in `cuttingPath`, a depth first search is not performed on e'.

4. Similar to the `reduce()` method, we again use a stack of vertices and the `inStack` field for `Vertex` objects to ensure that each vertex is queried only once during the depth first search.

Analysis of Connected Component Labeling: We will analyze all the connected component labeling in a reduced annular strand diagram R collectively. Let the original strand diagram corresponding to R be S, which is produced from the input word of length n. Let V_S and E_S be the number of vertices and edges respectively in S. Recall from Theorem 1.4.5 that $V_S + E_S \leq 15n + 3$. It is easy to see that the number of vertices and edges in R is also bounded by $15n+3$. Therefore, from statement (4) above, it follows that the `while` loop in Line 13 of Algorithm 9 executes $O(n)$ times. Because each vertex in R has a constant number of connected edges, it follows that one iteration of this `while` loop performs a constant number of operations. Hence all the connected component labeling are collectively bounded by $O(n)$.

3.6 Encoding Annular Strand Diagrams into Planar Graphs

As mentioned in the beginning of this chapter, we reduce the problem of determining whether two reduced annular strand diagrams are isotopic to the problem of determining whether two planar graphs are isomorphic, and then use the linear time planar graph isomorphism checker algorithm proposed in [10].

In this section, we cover Lines 6-9 of Algorithm 1, that is, the conversion of each connected component in a reduced annular strand diagram to a corresponding planar graph. We provide a function that encodes connected components to planar graphs, and we create an algorithm that implements this encoding. Then we show that this algorithm is linear in the number of vertices and edges in the reduced annular strand diagram, and therefore linear in the length of the input word used to construct the annular strand diagram.

Let X be the set of all connected, reduced annular strand diagrams. Let G be the set of planar graphs to which elements of X will be encoded. Let $\phi : X \to G$ be the encoding function. If $s \in X$ is a free loop, then let ϕ encode s to the planar graph $\phi(s)$ shown in Figure 3.6.1.

Consider the case when s is not a free loop. Observe that an edge $e \in s$ is both an output from a vertex, and an input to a vertex. As shown in Figure 3.6.2, e can be any of the following input and output types:

Input Types for e	Output Types for e
1. the lone input to a split	1. the lone output of a merge
2. the left input to a merge	2. the left output of a split
3. the right input to a merge	3. the right output of a split

Thus, there are nine unique output-input combinations, each of which defines an edge **class**, which is a pair of integers (out,in) such that out represents the output type and in represents the input type of the edge.

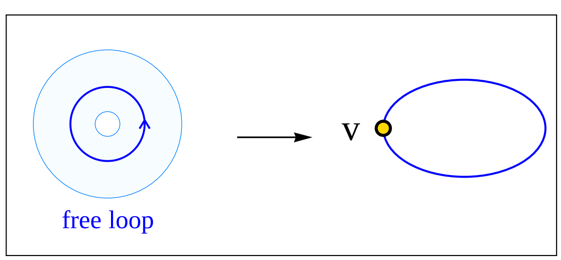


Figure 3.6.1. The encoding of $s \in X$ when s is a free loop. $\phi(s) \in G$ is the planar graph with a single vertex having a loop.

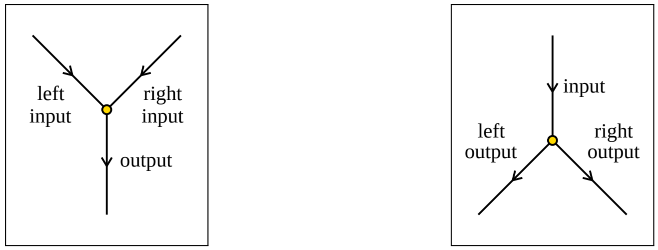


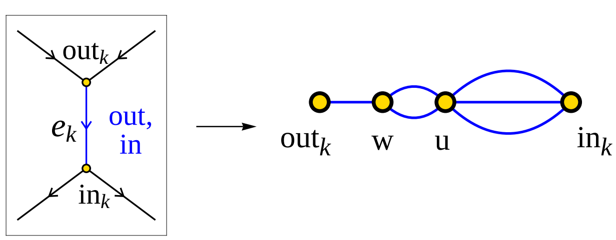
Figure 3.6.2. Different input and output types for edges. The vertex on the left is a merge, and the one on the right is a split

In order to achieve unique encoding of each $s \in X$, we will generate unique planar graphs for each possible class of edge in s. Below we discuss how we generate these planar graphs, and how the class to which an edge e belongs, uniquely identifies the corresponding planar graph for e.

Class (1,1): Edge $e_k = (out_k, in_k) \in s$ is the output of a merge out_k and the input of a split in_k. Produce the corresponding planar graph g_k using the following steps:

1. Create a null graph g_k having only the vertices out_k and in_k.
2. Add new vertices w_k and u_k to g_k.
3. In g_k, create an edge (out_k, w_k), two edges between w_k and u_k, and three edges between u_k and in_k.

Then g_k is the planar graph encoding of the edge e_k. Figure 3.6.3 shows the encoding of e_k in graphical form. The other eight edge classes have encodings very similar to edge



(a) e_k (b) the corresponding planar graph g_k

Figure 3.6.3. (a) shows the highlighted edge e_k which is the lone output of a merge and the lone input to a split. The encoding of e_k produces the planar graph g_k in (b).

class (1,1), except that they each differ in the number n_e of edges created between u_k and in_k in the corresponding planar graph. As the value of n_e is unique for each class of edge in the annular strand diagram, it follows that the encoding of each class of edge is unique. The encodings of all the edge classes are described in Table 3.6.

We are ready to encode the reduced, connected, non-free loop annular strand diagram s. Assume that $E = \{e_1, e_2, ...e_n\}$ is the edge set of s. To obtain $\phi(s)$, follow the steps below:

1. Create a null graph g.
2. Copy all the vertices from s to g.
3. For each edge $e_k = (v_k^{out}, v_k^{in}) \in E$, create vertices w_k and u_k in g. Identify the edge class in Table 3.6 to which e_k belongs, and let this be the class C, where $C \in \{1,2,3\} \times \{1,2,3\}$. Then create an edge (v_k^{out}, w_k), two edges (w_k, e_k), and p edges (u_k, v_k^{in}) where p is the number of edges (u, v_2) in the planar graph corresponding to class C in Table 3.6.

Then, $g = \phi(s)$.

Example 3.6.1. Let $s \in X$ be the reduced annular strand diagram for the word x_0x_0. The encoding creates the corresponding planar graph $\phi(s)$ shown in Figure 3.6.4. $\Diamond$

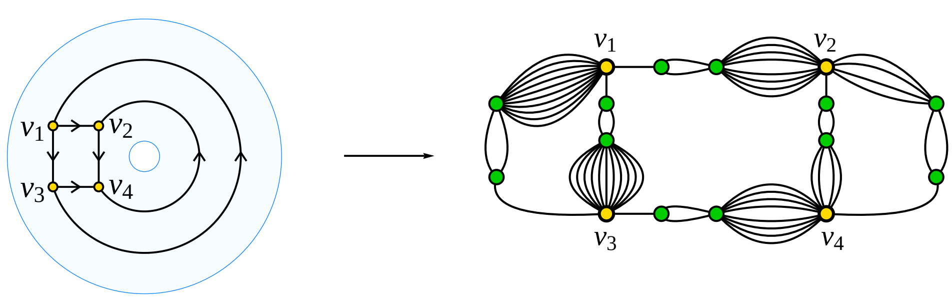


Figure 3.6.4. Encoding of x_0x_0 to its corresponding planar graph.

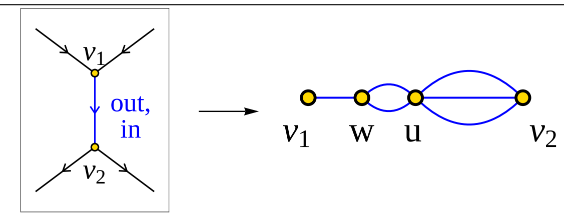
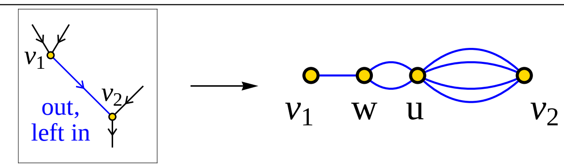
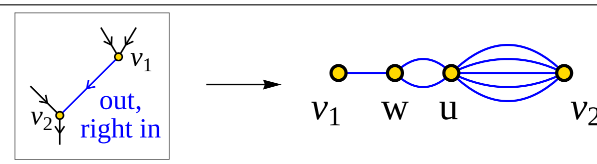
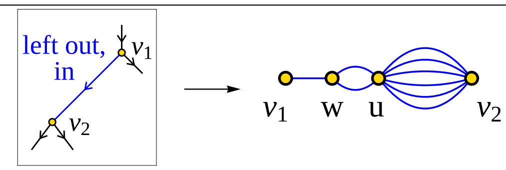
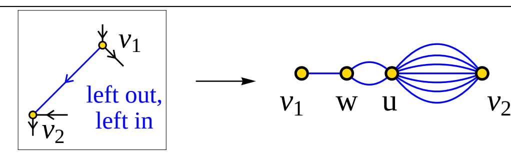
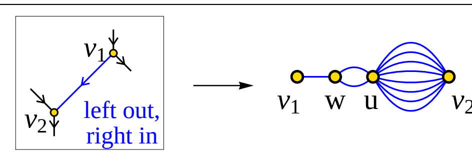
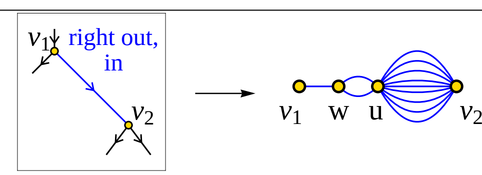
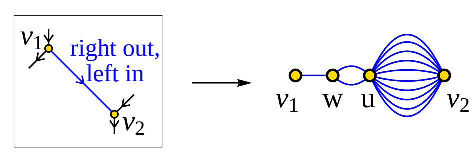
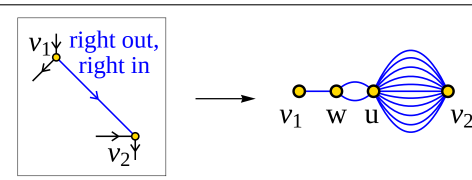
Class (1,1): an edge is the output of a merge v_1 and the input of a split v_2. **Solution:** create 3 edges between u and v_2	
Class (1,2): an edge is the output of a merge v_1 and the right input of a merge v_2. **Solution:** create 4 edges between u and v_2	
Class (1,3): an edge is the output of a merge v_1 and the right input of a merge v_2. **Solution:** create 5 edges between u and v_2	
Class (2,1): an edge is the left output of a split v_1 and the input of a split v_2. **Solution:** create 6 edges between u and v_2	
Class (2,2): an edge is the left output of a split v_1 and the left input of a merge v_2. **Solution:** create 7 edges between u and v_2	
Class (2,3): an edge is the left output of a split v_1 and the right input of a merge v_2. **Solution:** create 8 edges between u and v_2	
Class (3,1): an edge is the right output of a split v_1 and the input of a split v_2. **Solution:** create 9 edges between u and v_2	
Class (3,2): an edge is the right output of a split v_1 and the left input of a merge v_2. **Solution:** create 10 edges between u and v_2	
Class (3,3): an edge is the right output of a split v_1 and the right input of a merge v_2. **Solution:** create 11 edges between u and v_2	

Table 3.6.1. Different types of edges in the annular strand diagram, and their encodings to planar graphs

3.6.1 The Encoding Algorithm

We now describe our implementation of an algorithm to encode components of reduced annular strand diagrams into planar graphs. Let X be the set of reduced connected annular strand diagrams. As shown in Algorithm 10 (in the Appendix), the method `encodeToPlanarGraph()` takes as input $s \in X$, applies the encoding function ϕ to s, and returns the corresponding planar graph $\phi(s)$.

Analysis of the Encoding Algorithm: Note that this analysis covers Lines 7-9 in Algorithm 1. We will analyze the encodings of all connected components collectively using the same approach we used to analyze the running time of connected component labeling in Section 3.5.

Recall that given an input word of length n, all the connected components together have sum of vertices and edges bounded by $O(n)$. Therefore, the `for` loops in Line 7 and in Line 12 execute $O(n)$ times and each iteration of both of these loops performs a constant number of operations. It follows that the execution efficiency of Algorithm 10 is $O(n)$.

3.7 Retrieving Annular Strand Diagrams from Planar Graph Encodings

Now we will show that given $p_s \in G$, the planar graph encoding of $s \in X$, we can decode p_s to derive s. Observe that the vertices in p_s that also belong to s share at least three edges with at least one vertex, and they cannot share exactly two edges with any other vertices. Using these properties, we can construct all the vertices in s from the set of vertices in p_s. Since each class of edge in s has a unique corresponding planar graph encoding, conversely, each of these planar graphs correspond to a unique class of edge in s. Let $P = \phi(X)$ be the image of ϕ. Let $\psi : P \to X$ be the decoding function. Let $p_s \in P$ and $p_s = (V, E)$, where V and E are the vertex and edge sets of g respectively. If p_s is a graph with only a single vertex having a loop, then $\psi(p_s)$ is the free loop. Otherwise we compute $\psi(g)$ using the steps below:

1. Create a null annular strand diagram s.
2. For each $v_k \in V$, if there exists $u, w \in V$ such that there exist exactly one edge (v_k, w) and exactly two edges (w, u) in E, then add v_k to s.
3. For each v_k in the vertex set of s, find all $v'_k \in s$ such that $u, w \in p_s$, and there exist exactly one edge (v_k, w), exactly two edges (w, u), and at least three edges (u, v'_k) in E. For each v'_k, if the total number of edges (u, v'_k) is c_n, then use c_n to find the corresponding edge class C in Table 3.6, and create in s a directed edge (v_k, v'_k) whose output and input type are the same as that of the highlighted directed edge in class C.

Then, $s = \psi(p_s)$. Algorithm 11 shows the decoding algorithm that produces s given p_s. Notice that it is the inverse of the encoding algorithm. We do not need to worry about the execution efficiency of the decoding algorithm as this algorithm is not used in our proposed solution to the conjugacy problem in F. We skip the analysis of the decoding algorithm.

Theorem 3.7.1. *Any two connected, reduced, and non-free loop Annular Strand Diagrams s_1 and s_2 can be encoded into two planar graphs g_1 and g_2 respectively such that s_1 and s_2 are isotopic if and only if g_1 and g_2 are isomorphic.*

Proof. Let s_1 and s_2 be two connected, reduced, and non-free loop annular strand diagrams. Let the encoding be the function ϕ described in Section 3.6. In order to prove the theorem, it suffices to show that the encoding is one-to-one. Let $g_1 = \phi(s_1)$ and $g_2 = \phi(s_2)$. Assume that $g_1 = g_2$. Because of the nature of ϕ, it immediately follows that $\phi(s_1) = \phi(s_2)$. Also, by applying the decoding function ψ, described in Section 3.7, on g_1 and g_2, we get the following:

$$s_1 = \psi(g_1) = \psi(g_2) = s_2$$

It follows that ϕ is one-to-one. □

3.8 Isomorphism Checking

This section covers Lines 11-20 in Algorithm 1 and completes the analysis of the solution algorithm. The relevant lines of code are very straightforward:

1. If the total number of planar graph encodings is not the same for both reduced annular strand diagrams, then it immediately follows that the reduced annular strand diagrams are not isotopic.
2. Otherwise, we take each planar graph encoding p_1 and p_2 corresponding to words w_1 and w_2 respectively in order of increasing distance from the cutting path, and then check whether p_1 and p_2 are isomorphic.
   (a) If any such pair is not isomorphic, then the corresponding reduced annular strand diagrams are not isotopic, and hence the words w_1 and w_2 are not conjugate.
   (b) If all pairs are isomorphic, then w_1 and w_2 are conjugate.

Analysis of Check for Isomorphism between Planar Graphs: Hopcroft and Wong [10] have shown an algorithm that determines whether two planar graphs are isomorphic in $O(|V|)$, where $|V|$ is the sum of the number of vertices in the input planar graphs. Their algorithm can also be applied on planar graphs which have loops and multiple edges between vertices such as our encoded planar graphs.

Because the total number of vertices from all our planar graph encodings are collectively bounded by $O(n)$, it follows that the worst case running time of the `for` loop starting in

Line 13 of Algorithm 1 (i.e., all the checks for isomorphism between planar graphs) is $O(n)$.

This proves Theorem 3.0.5 and confirms that the solution we presented to the conjugacy problem in F is linear in the sum of the length of the input words.

3.9 Our Actual Implementation

To the best of our knowledge, the linear time algorithm that checks for an isomorphism between two planar graphs proposed in [10] used in Line 16 in Algorithm 1 has not been implemented yet. We believe that this is due to the complicated design of the algorithm. Moreover, the authors of [10] stated that this algorithm is not practical. As a result, we have not attempted an implementation of this algorithm, and instead programmed a direct isotopy search as a substitute for the following steps:

1. planar graph encoding of each component in the two annular strand diagrams, and
2. checking for isomorphism between all corresponding pairs of planar graph encodings.

Because of this change, ConjugacyF takes $O(n^2)$. We implemented Algorithm 6, which modifies Algorithm 1 by substituting the two steps above with a detector that checks the rotation systems of two reduced annular strand diagrams to determine whether they are isotopic. The rest of this section describes the isotopy detector, and analyzes its running time.

Recall that our data structure provides the counterclockwise ordering of edges connected to each vertex in an annular strand diagram. Using this information, we can create the rotation system for two reduced annular strand diagrams, and then determine whether one rotation system is a permutation of the other.

Input: String w_1, String w_2
Output: Whether w_1 and w_2 are conjugate: `true` or `false`

```
for w in {w1, w2} do
    // generate strand diagram from word w
    Strand sd = new Strand(w)
    Annular asd = sd.close() // obtain annular strand diagram
    asd.reduce() // see Algorithm 2
    List<Annular> C_w = asd.getComponents()
end
if C_w1.size() ≠ C_w2.size() then
    return false
for i = 0 → C_w1.size() − 1 do
    Annular c1 = C_w1.get(i)
    Annular c2 = C_w2.get(i)
    if !isIsotopic(c1, c2) then
        // see Algorithm 13 in the Appendix
        return false

end
return true
```

Algorithm 6: Our working implementation ConjugacyF

By convention, we obtain the edges for a merge starting with the left parent edge and going counterclockwise. In the case of splits, we start with the left child edge and move counterclockwise.

3.9.1 Isotopy Detector Given two Corresponding Vertices

First we discuss the method `isotopyHelper()`, our implementation to check whether two two connected, reduced annular strand diagrams c_1 and c_2 are isotopic given two corresponding vertices `ref` $\in c_1$ and `corr` $\in c_2$. A brief outline of the flow of this algorithm is shown below, and a comprehensive description is provided in Algorithm 12 (in the Appendix). Note that the term **edge cycle** refers to a counterclockwise order of edges connected to a vertex.

1. This algorithm proceeds by checking whether the edge cycles of the vertices `ref` and `corr` are the same, which happens when these vertices are of the same type (i.e., both merges or both splits) and the vertex at the other end of an edge $e \in$ `ref` is of the same type as the vertex at the other end of its corresponding edge $e' \in$ `corr`, for all edges in the edge cycle of `ref`.

2. If step (1) successfully checks out, then for each edge e in the edge cycle of `ref`, the vertices v connected to e and v' connected to its correspondent e' in the edge cycle of `corr` are marked using the field `isPaired`, v' is set as the corresponding vertex of v using the field `correspondent`, and v is added to a stack. The function of `isPaired` is to avoid putting a vertex on the stack more than once, which guarantees that each edge cycle in c_1 is assigned exactly one corresponding edge cycle in c_2, and `correspondent` records the correspondence between an edge cycle in c_1 and an edge cycle in c_2.

3. While the stack is not empty, a vertex v_1 is popped from the stack, and the algorithm loops back to step (1) with v_1 as the reference and v_1.`correspondent` as its correspondent.

4. If each vertex $v_1 \in c_1$ is successfully assigned a unique corresponding vertex $v_2 \in c_2$ then the algorithm declares that c_1 and c_2 are isotopic.

Analysis of isotopyHelper(): Given that the sum of the vertices in c_1 and c_2 is N, since:

1. the total number of vertices added to the stack is less than N, and

2. there is a constant number of edges connected to each vertex

it follows that the execution of `isotopyHelper()` (see Algorithm 12) takes $O(N)$.

3.9.2 Isotopy Detector Given two Connected Reduced Annular Strand Diagrams

We now describe the method `isIsotopic()`, which determines whether two connected, reduced annular strand diagrams c_1 and c_2 are isotopic. As shown in Algorithm 13 (in the Appendix), this method proceeds by fixing a vertex `reference` $\in c_1$ and then looping through all vertices `v` $\in c_2$ to check whether c_1 and c_2 are isotopic with `reference` corresponding to any such `v`, using `isotopyHelper()`.

Analysis of isIsotopic(): Assuming that the sum of the vertices in c_1 and c_2 is N, in the worst case, the `for` loop in Line 6 runs $O(N)$ times. Each execution of Line 10 inside this `for` loop takes $O(N)$ in the worst case. It follows that the method `isIsotopic()`, summarized in Algorithm 13 takes $O(N^2)$.

Analysis of the algorithm that determines whether two Reduced Annular Strand Diagrams are Isotopic: We will analyze this step in the same way we analyzed connected component labeling earlier, that is, we collectively analyze whether all corresponding connected components of two reduced annular strand diagrams are isotopic.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two reduced annular strand diagrams each corresponding to an input word of length $O(n)$. Recall that the number of vertices in $\mathcal{A}_1$ and $\mathcal{A}_2$ is bounded by $O(n)$. The worst case happens when all corresponding components are isotopic except the last ones, which are "almost" isotopic. For this case, let $C_1, ..., C_M$ be the components of $\mathcal{A}_1$ in concentric order with such that for each C_i, the number of vertices is denoted by V_i. Then, for each C_i, the call to `isIsotopic()` in Line 12 of Algorithm 6 takes $O((V_i)^2)$. Then all the calls to `isIsotopic()` collectively take

$$O((V_1)^2) + O((V_2)^2) + ... + O((V_M)^2) \approx O((V_1)^2 + (V_2)^2 + ... + (V_M)^2).$$

However, observe that $(V_1)^2 + (V_2)^2 + ... + (V_M)^2 < n^2$.

It follows that all the calls to `isIsotopic()` in the `if` statement in Line 12 are collectively bounded by $O(n^2)$.

Therefore, our working implementation ConjugacyF for the conjugacy problem in Thompson's Group F has running time $O(n^2)$.

3.10 Results

n	p, the No. of vertices in reduced annular diagrams													
	0	2	4	6	8	10	12	14	16	18	20	22	24	26
1	0	4	0	0	0	0	0	0	0	0	0	0	0	0
2	0	6	6	0	0	0	0	0	0	0	0	0	0	0
3	0	6	10	8	0	0	0	0	0	0	0	0	0	0
4	0	8	16	14	12	0	0	0	0	0	0	0	0	0
5	0	8	26	40	22	16	0	0	0	0	0	0	0	0
6	0	8	34	58	56	34	28	0	0	0	0	0	0	0
7	0	8	44	98	124	98	62	40	0	0	0	0	0	0
8	0	8	56	156	240	234	194	106	72	0	0	0	0	0
9	0	8	60	228	452	536	476	368	202	120	0	0	0	0
10	1	8	68	314	756	1108	1148	954	742	378	216	0	0	0
11	1	8	68	386	1204	2136	2638	2434	1990	1474	730	376	0	0
12	1	8	72	480	1806	3790	5436	5794	5324	4160	2988	1472	704	0

Table 3.10.1. Each entry in the table shows the number of unique reduced annular strand diagrams having p vertices obtained using words of length up to n.

In this section, we verify that the Java implementation of ConjugacyF is correct and bug-free by using ConjugacyF on a large number of inputs and showing that the outputs are free of error.

Notice that each conjugacy class in F corresponds to a unique reduced annular strand diagram. In our experiment, first we computed all the cyclically reduced words starting from length $n = 1$ up to $n = 12$, and sorted them into conjugacy classes using ConjugacyF. Then, for each value of n. we tabulated the total number of unique reduced annular strand diagrams with number of vertices in the set $\{0, 2, 4, ..., 26\}$. The results are summarized in Table 3.10.1. Note that there exists no reduced annular strand diagram with an odd number of vertices because the number of merge and splits have to be the same (i.e., both odd or both even).

3.10.1 Verification of Results

We will now analyze the total number of conjugacy classes with two vertices and with four vertices obtained using our algorithm and verify that they are correct.

Conjugacy classes with two vertices: Observe that all reduced annular strand diagrams with two vertices can be generalized into the same basic structure shown in Figure 3.10.1. In this structure,

1. the edge connecting the two vertices can have two possible directions,
2. there can be a free loop in the innermost region, and
3. a free loop in the outermost region,

giving $2 \times 2 \times 2 = 8$ unique reduced annular strand diagrams. This is in accordance with the column $p = 2$ in Table 3.10.1, where the entries reach their maximum value 8 as n is increased to 4. Furthermore, each of the following 8 words was the first input for which ConjugacyF found a unique reduced annular strand diagram with 2 vertices:

x_0 x_1 x_0^{-1} x_1^{-1} $x_0x_1^{-1}$ $x_1x_0^{-1}$ $x_0x_1x_0^{-1}x_1^{-1}$ $x_0x_1^{-1}x_0^{-1}x_1$

We verified these results by manually drawing and reducing the annular strand diagrams corresponding to these words.

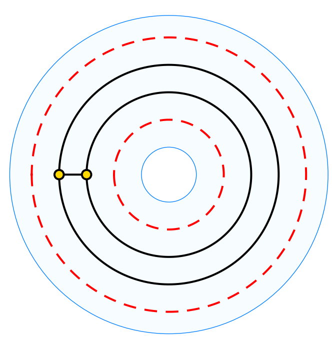


Figure 3.10.1. A general structure for reduced annular strand diagrams with two vertices. The red dashed circles show free loops.

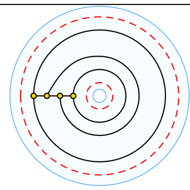
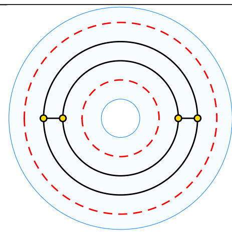
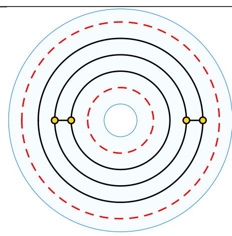
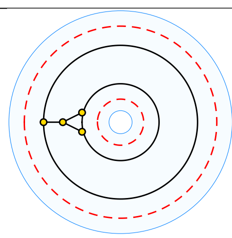
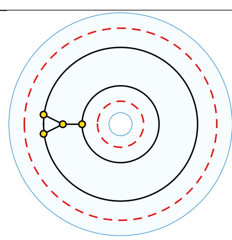
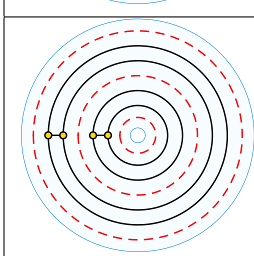
General Structure	Unique Reduced Annular Strand Diagrams Generated
	1×2 choices for edge directions 2×2 choices for free loops $= 8$ unique reduced annular strand diagrams
	1×2 choices for edge directions 2×2 choices for free loops $= 8$ unique reduced annular strand diagrams
	1×2 choices for edge directions 2×2 choices for free loops $= 8$ unique reduced annular strand diagrams
	1×2 choices for edge directions 2×2 choices for free loops $= 8$ unique reduced annular strand diagrams
	1×2 choices for edge directions 2×2 choices for free loops $= 8$ unique reduced annular strand diagrams
	2×2 choices for edge directions $2 \times 2 \times 2$ choices for free loops $= 32$ unique reduced annular strand diagrams
	Total $= \ 8+8+8+8+8+32$ $= 72$ unique reduced annular strand diagrams

Table 3.10.2. Reduced annular strand diagrams with four vertices. The number of unique reduced annular strand diagrams corresponding to each general structure is a power of 2.

Conjugacy classes with four vertices: Table 3.10.2 shows the six general structures for reduced annular strand diagrams with four vertices. These structures show that there are 72 unique reduced annular strand diagrams that have four vertices. This also agrees with Table 3.10.1 where the column $p = 4$ has its highest value at 72 when $n = 12$.

We note that this result can be further verified by increasing n beyond 12. However, we already have a total of 32,035 conjugacy classes (the sum of entries in row 12 of Table 3.10.1). At $n = 13$ there are 1,594,324 new cyclically reduced words, and we decided that comparing them all with existing conjugacy classes is infeasible.

3.10.2 Shortest Cyclically Reduced Word for the Identity

The first occurrence of the free loop conjugacy class happens at $n = 10$ (and $p = 0$). Recall that a free loop represents the identity. The first word that ConjugacyF sorted into the free loop conjugacy class was

$$x_0x_0x_1x_0^{-1}x_0^{-1}x_1x_0x_1^{-1}x_0^{-1}x_1^{-1}.$$

Observe that the shortest relation in the presentation for F shown in Proposition 1.3.4 is $x_1x_2 = x_3x_1$, where $x_2 = x_0x_1x_0^{-1}$ and $x_3 = x_0^2x_1x_0^{-2}$. Then, the above word becomes,

$$(x_0^2x_1x_0^{-2})x_1(x_0x_1^{-1}x_0^{-1})x_1^{-1} \qquad = \qquad x_3x_1x_2^{-1}x_1^{-1} \qquad = \qquad (x_3x_1)(x_1x_2)^{-1}$$

which is the identity.

In fact, the smallest cyclically reduced word in F representing the identity has length 10. This statement can be proved using forest diagrams [2], which are beyond the scope of this paper.

3.11 The Software

In this section, we discuss the variants of our software for the conjugacy problem in F that we have provided online, with directions on how to interact with the user interface.

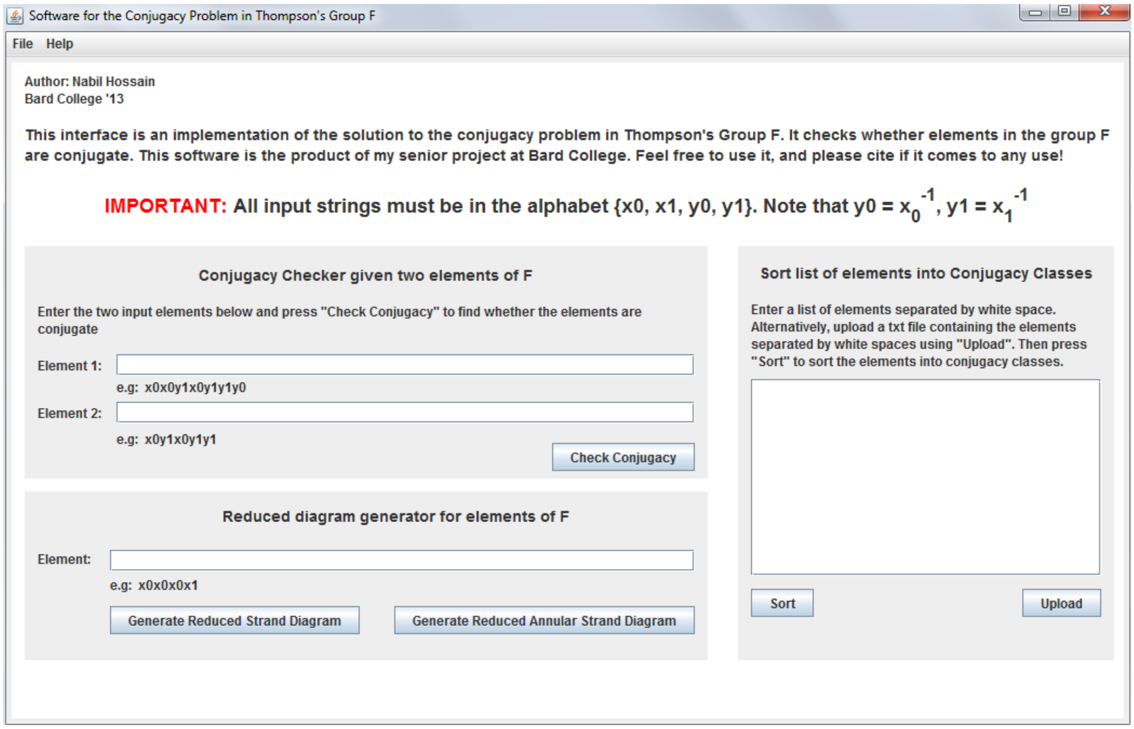


Figure 3.11.1. The user interface of the application for the conjugacy problem in F.

3.11.1 The Variants and the Implementation Details

We make ConjugacyF available as an executable JAR file on [11] that can be downloaded freely and used offline. However, note that the application has been compiled on the Windows 7 environment, and therefore it might not work properly on other operating systems such as Unix or Mac OS. Users of these operating systems are advised to use the alternative online application provided in the form of a Java applet on [11]. Note that for ease of user interaction, our software accepts input files and creates output files, which Java applets do not permit due to security reasons, and these features are not available on the web application. A screen shot of the application's user interface is shown in Figure 3.11.1.

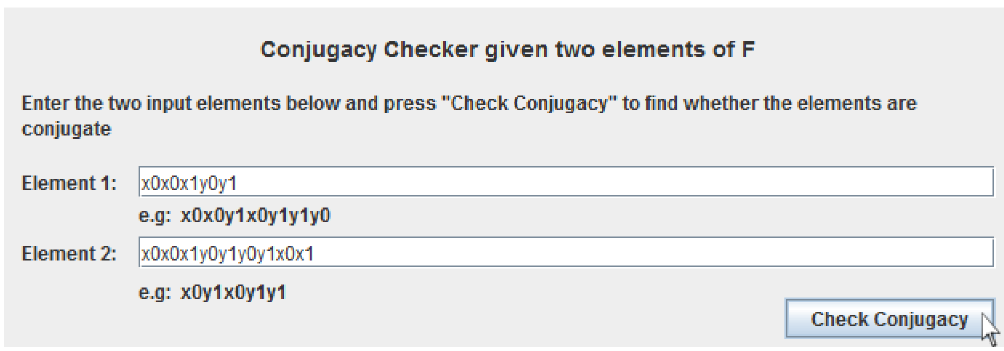
We have also shared the source code on [11] to allow users the opportunity to customize the software based on their own needs.

3.11.2 Using the Application

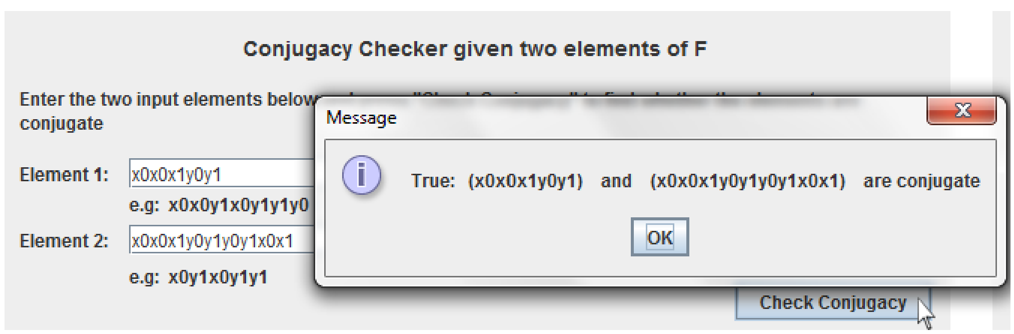
Note 3.11.1. In the application, all input words must be in the alphabet {`x0`,`x1`,`y0`,`y1`}, where `y0` represents the element x_0^{-1} and `y1` represents the element x_1^{-1}. ◊

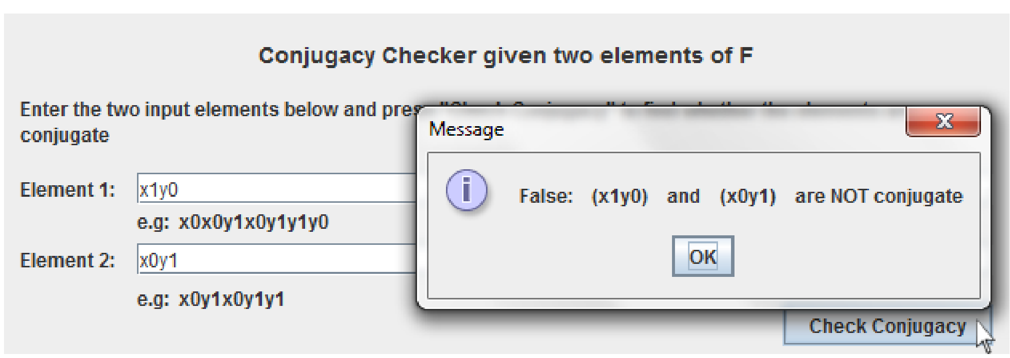
The application allows the following five functions (functions 4 & 5 are the same except they take the input differently):

1. **Input:** two words (e.g. `x0x0x1y0y1` and `x0x0x1y0y1y0y1x0x1`):

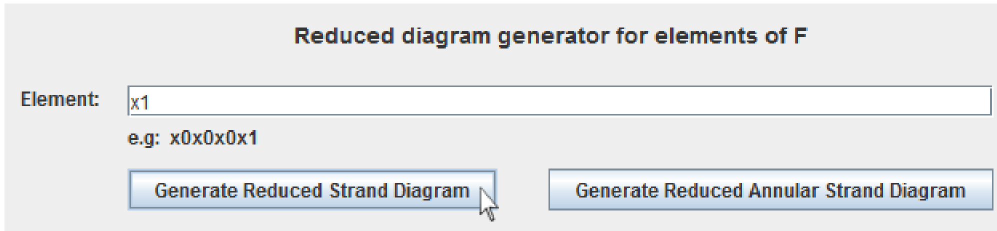


Output: whether the elements corresponding to these words are conjugate:





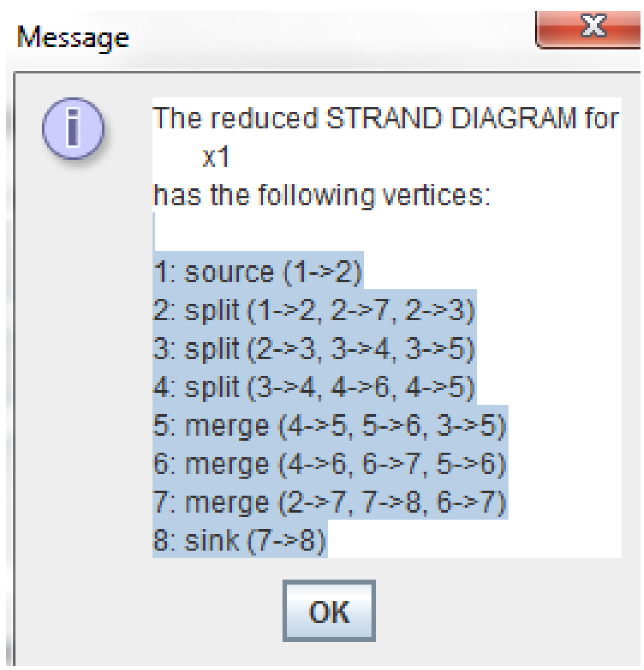
2. **Input:** a word (e.g. `x1`):



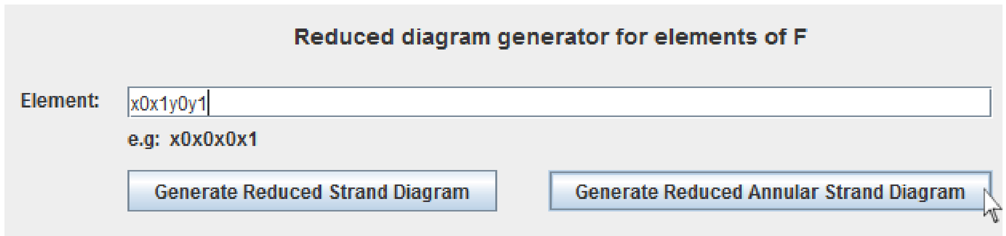
Output: the corresponding reduced strand diagram, described using vertices and edges. Each vertex is uniquely numbered, its type is labeled, and its connected edges are shown in counterclockwise order.

- For a split, the edges are listed starting with the lone parent.
- For a merge, the edges are listed starting with the left parent.

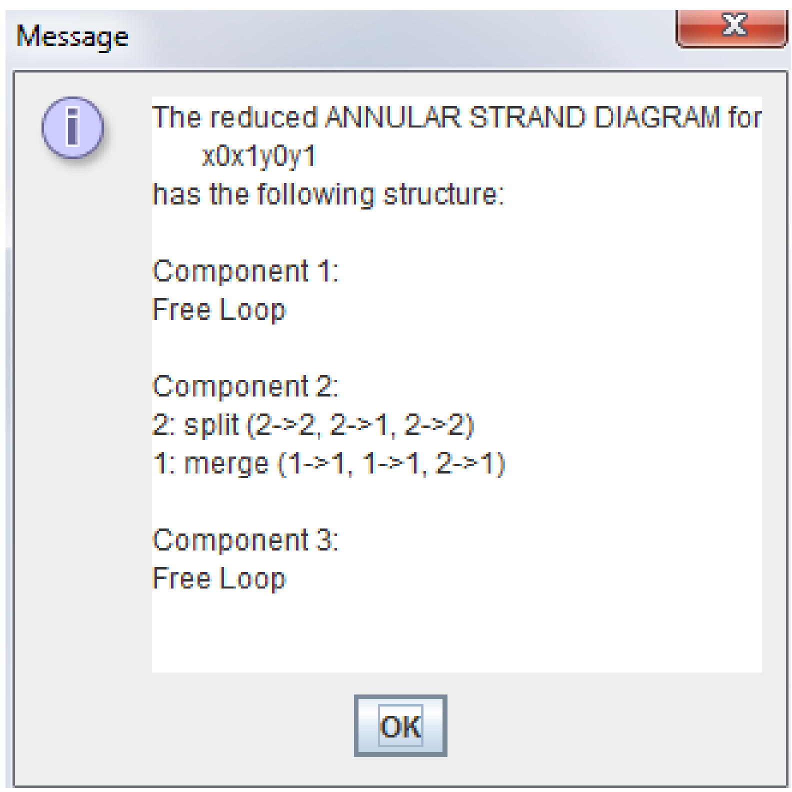
The output text is editable, so it can be selected and copied (use Ctrl+c).



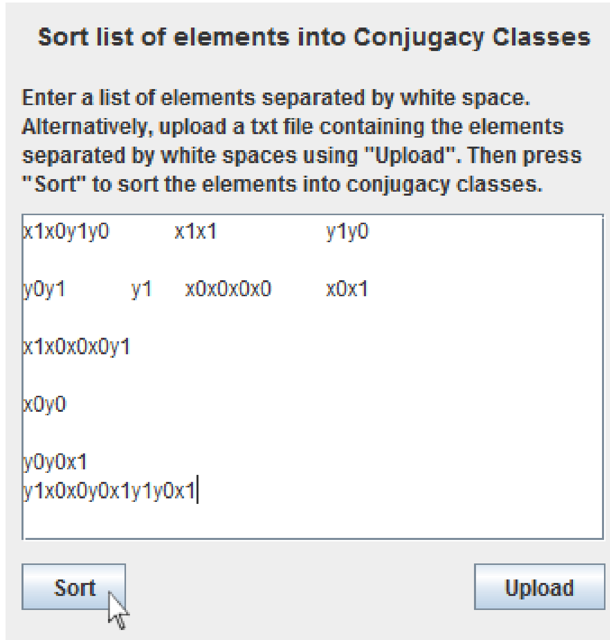
3. **Input:** a word (e.g. `x0x1y0y1`):



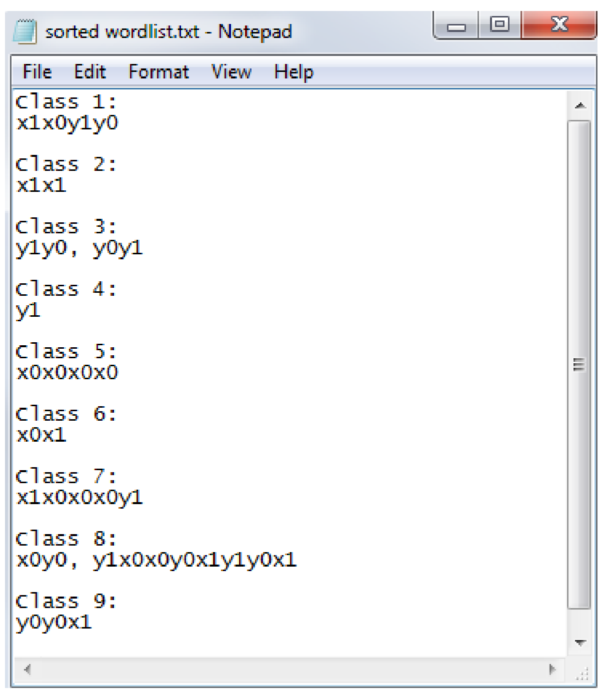
Output: the corresponding reduced annular strand diagram, described using components, vertices, and edges. Each component is uniquely numbered, and in each component, each vertex is described as in function (2) above. Similar to function (2), the output text is editable.



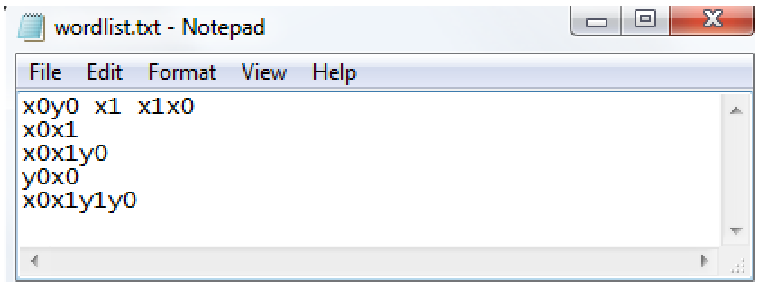
4. **Input:** a list of words separated by white spaces:

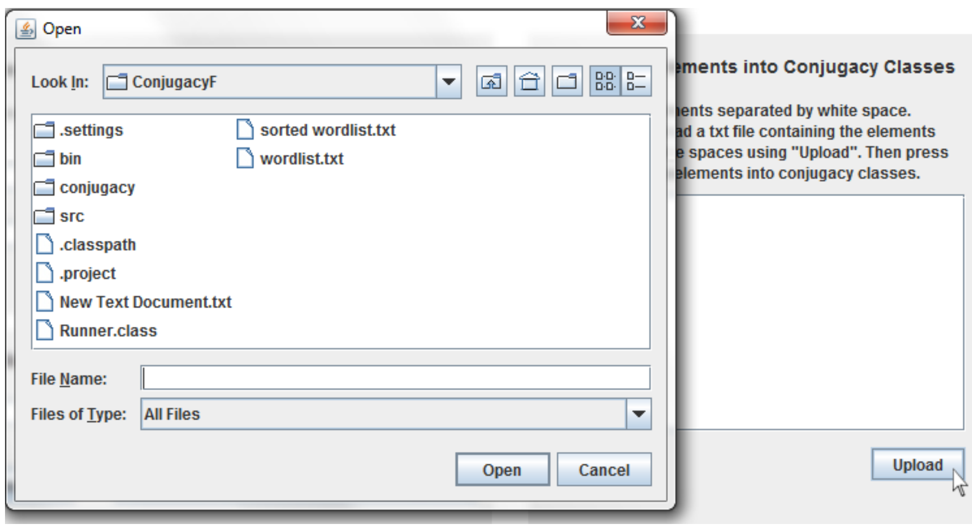


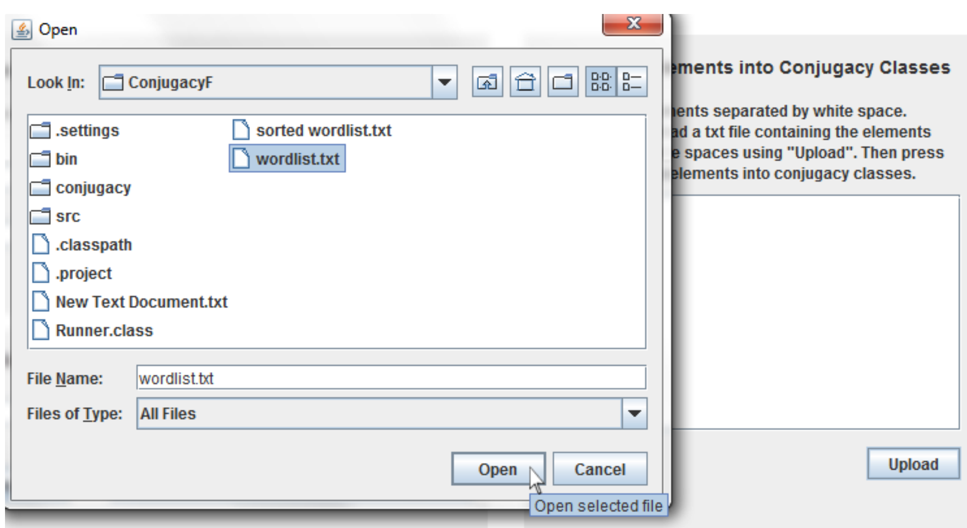
Output: a TXT file called "sorted wordlist.txt" (located in the directory that contains the executable JAR) that has the input words sorted into conjugacy classes:

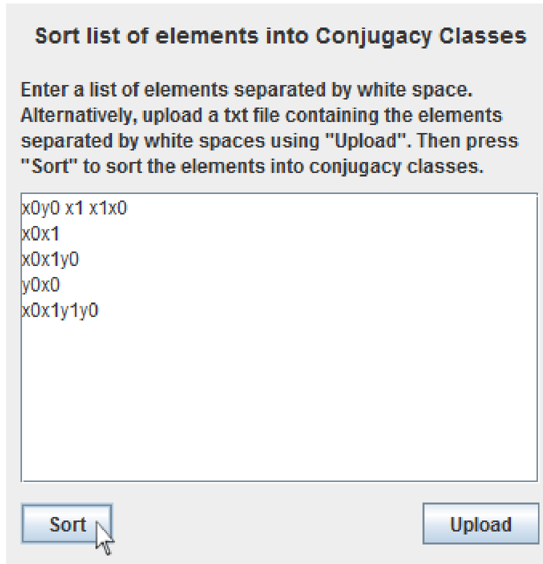


5. **Input:** a TXT file having a list of words that are separated by white spaces. This file is selected using the "Upload" button. Next, the user has to press "Sort":

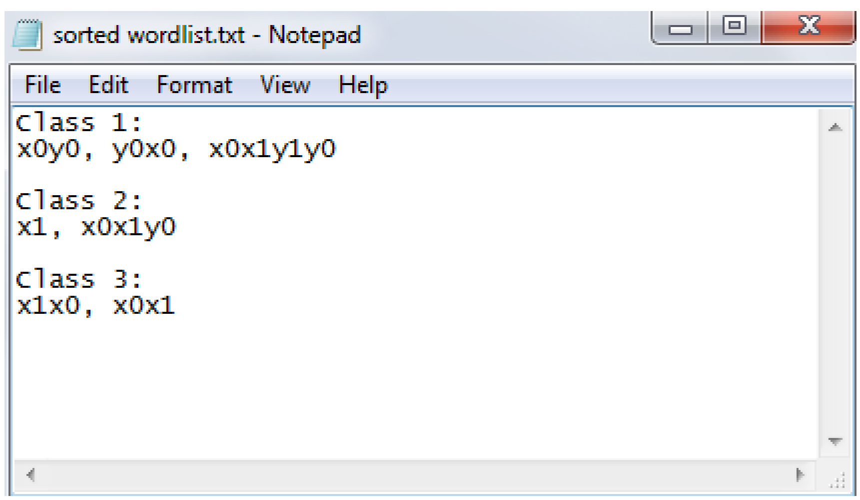








Output: a TXT file called "sorted wordlist.txt" (located in the directory that contains the JAR application) that has the input words sorted into conjugacy classes:



Note that the web application neither accepts input files, nor produces output files. So, it cannot perform function (5), but it can run function (4) where it creates an output window instead of an output file.

4
Conclusion and Future Work

In this project, we presented a linear time algorithm that solves the conjugacy problem in Thompson's Group F using directed graphs called strand diagrams. We also provided an efficient data structure for storing strand diagrams and perform operations on them.

The proposed algorithm converts strand diagrams into directed graphs embedded on the annulus called annular strand diagrams, reduces these graphs, and then compares the two reduced annular strand diagrams to see whether they are the same, or more appropriately, whether they are isotopic. We believe that this is the fastest possible algorithm solving the conjugacy problem in F.

Due to the impractical nature of the linear algorithm for the isomorphism problem in planar graphs which is part of our solution algorithm, we implemented a quadratic solution that directly determines whether two reduced annular strand diagrams are isotopic. As a result, our implementation called ConjugacyF is quadratic in the length of the input.

We released our implementation in the form of a web application in a Java applet, and a graphical interface programmed in Java that can be freely downloaded and run offline. Furthermore, we provided the source code to allow users with more flexibility in using our program. To the best of our knowledge, this is the first implementation of an algorithm for the conjugacy problem in F. It is our hope that our software will be useful to the research community in Thompson's Groups.

For future work, we believe that it would not be too hard to modify our software to create algorithms for the conjugacy problems in Thompson's Groups V and T. The cutting path used in the algorithm for F is representative of the *cutting class* [3] used in solving the conjugacy problem in V. However, the algorithm for V is not expected to be linear because checking whether two cutting paths represent the same cutting class requires gaussian elimination [7] (i.e., row reduction), which possibly takes $O(n^5)$.

For Thompson's Group T, there is no reason to believe that an algorithm solving its conjugacy problem will be linear. Its strand diagrams are embedded on a cylinder, and its conjugacy problem is solved using a similar approach as in V.

Appendix: Algorithm Descriptions

Input: `Edge this` (which invokes this method), `Edge` e_2
Output: An `Edge` that merges `this` with e_2

```
if this.node ≠ null then
    if e2.node ≠ null then
        e2.node.previous.next = e2.node.next
        e2.node.next.previous = e2.node.previous
    e2.node = this.node
else
    if e2.node ≠ null then
        this.node = e2.node
    else
        this.node = new Node<Edge>()
        e2.node = this.node

this.target = e2.target
this.class[1] = e2.class[1]
if e2.class[1] == 1 then
    this.target.leftParentEdge = this
else if e2.class[1] == 2 then
    this.target.rightParentEdge = this

return this
```

Algorithm 7: The `combineEdge(Edge)` method for `Edge` objects.

Input: `Strand this` (which invokes this method) `Strand s`
Output: The `Strand` object which is the concatenation of `this` to `s`

```
Vertex merge = this.sink.getLeftParent()
Vertex split = s.source.getLeftChild()
merge.leftChildEdge.combineEdge(split.leftParentEdge)
this.vertices.remove(this.sink)
s.vertices.remove(s.source)
this.vertices.add(s.vertices)
this.sink = s.sink
this.stackReduceSplits.add(split)
split.inStack = true return this
```

Algorithm 8: The method `concatenate()` for `Strand` objects. Because `vertices` is a linked list, the method `add()`, which joins two linked lists in this case, executes in constant time.

Input: `LinkedList<Node> cuttingPath`, the cutting path
Output: `List<Annular> cc`, the list of concentrically ordered connected components

```
LinkedList<Annular> cc = new LinkedList<Annular>()
Node current = cuttingPath.getFirst()
while current.next ≠ null do
    Edge e = current.data; current = current.next
    if e.isFreeLoop then
        cc.add(new Annular(new LinkedList<Vertex>()))
        continue
    if !e.flagged then
        LinkedList<Vertex> ccVertices = new LinkedList<Vertex>()
        Stack<Vertex> s = new Stack<Vertex>()
        ccVertices.add(e.source)
        s.push(e.source); e.source.inStack = true
        while !s.isEmpty() do
            Vertex v = s.pop()
            if v.type == "merge" then
                edgeSet = {leftParentEdge,leftChildEdge,rightParentEdge}
            else
                edgeSet = {leftParentEdge, leftChildEdge,rightChildEdge}
            for E in edgeSet do
                if !E.flagged then
                    if !E.source.inStack then
                        s.push(E.source); E.source.inStack = true
                    if !E.target.inStack then
                        s.push(E.target); E.target.inStack = true
                    E.flagged = true

            end
        end
        cc.add(new Annular(ccVertices))

end
return cc
```

Algorithm 9: The `getComponents()` method to extract connected components from reduced annular strand diagrams.

Input: `Annular c` (the connected component which invokes this method)
Output: `Graph` p_c, the corresponding planar graph encoding of `c`

```
Graph p_c = new Graph()
if c.vertices.isEmpty() then
    // the component is a free loop
    Vertex v = new Vertex()
    v.add(new Edge(v,v))
    p_c.vertices.add(v)
    return p_c
for id = 0 → c.vertices.size() − 1 do
    p_c.vertices.add(new Vertex (id))
    c.vertices.get(id).setID(id) // ensures that vertices with the same
    ID in p_c and c correspond to each other
end
Set the flagged fields of all edges in c to false
for v in c.vertices do
    if v.type == “merge” then
        edgeSet = {leftParentEdge,leftChildEdge,rightParentEdge}
    else
        edgeSet = {leftParentEdge, leftChildEdge,rightChildEdge}
    for e in edgeSet do
        if !e.flagged then
            e.flagged = true
            Vertex v_1 = e.source
            Vertex v_2 = e.target
            Obtain corresponding vertices v'_1 and v'_2 in p_c using the ID of v_1 and v_2
            Create new vertices u and w in p_c
            Find which one of the nine classes (in Table 3.6) e falls into
            Then perform the encoding by adding the edges to p_c as described by that
            class

    end
end
return p_c
```

Algorithm 10: The method `encodeToPlanarGraph()`

Input: `Graph` p_s
Output: `Annular s`, the corresponding annular strand diagram for p_s

`Annular s = new Annular()`
if p_s*.vertices.size() == 1* **then**
  create a free loop in `s`
  return `s`
end
for v_1 *in* p_s*.vertices* **do**
  for *each vertex* w *which shares exactly 1 edge with* v_1 **do**
    Find `u` such that $u \neq v_1$ and `(w,u)` is an edge
    Find v_2 such that $v_2 \neq w$ and (`u`,v_2) is an edge
    Compute c_n `= |u|-2` `//` c_n `= # of edges between` u `and` v_2
    Add v_1 and v_2 in `s` if they are not in `s`
    In `s`, add the highlighted directed edge (v_1, v_2), which belongs to the corresponding class of c_n in Table 3.6
  end
end
return `s`

Algorithm 11: The algorithm to retrieve a connected, reduced annular strand diagram given its planar graph encoding

Input: `Annular` c_1`, Vertex ref, Annular` c_2`, Vertex corr`
Output: Whether c_1 is isotopic to c_2 given `ref` $\in c_1$ corresponds to `corr` $\in c_2$: `true` or `false`

```
if ref.type ≠ corr.type then
    return false
Stack<Vertex> stack = new Stack<Vertex>()
stack.push(ref)
ref.correspondent = corr
ref.isPaired = true; corr.isPaired = true
while !stack.isEmpty() do
    Vertex v1 = stack.pop()
    Vertex v2 = v1.correspondent
    if v1.type == "merge" then
        vertexSet =
        {v.getLeftParent(),v.getLeftChild(),v.getRightParent()}
    else
        vertexSet =
        {v.getLeftChild(),v.getRightChild(),v.getLeftParent()}
    for node in vertexSet do
        v1n = v1.node; v2n = v2.node
        if v1n.type ≠ v2n.type then
            return false
        else if v1n.isPaired ≠ v2n.isPaired then
            return false
        if v1n.isPaired == true then
            if v1n.correspondent ≠ v2n then
                return false
            else
                v1n.isPaired = true; v2n.isPaired = true; stack.push(v1n)
    end
end
return true
```

Algorithm 12: The method `isotopyHelper()`, which determines whether two connected, reduced annular strand diagrams c_1 and c_2 are isotopic given vertices `ref` $\in c_1$ and `corr` $\in c_2$ as correspondents

Input: `Annular` c_1`,` `Annular` c_2
Output: Whether c_1 and c_2 are isotopic: `true` or `false`

if c_1*.vertices.size()* $\neq$ c_2*.vertices.size()* **then**
    return false
else if c_1*.vertices.isEmpty()* **then**
    return true `// both` c_1 `and` c_2 `are free loops`
`Vertex reference =` c_1`.vertices.get(0)`
for *v in* c_2*.vertices* **do**
    Set `isPaired` field of all vertices in c_1 and c_2 to `false`
    Set `correspondent` field of all vertices in c_1 to `false`
    `reference.correspondent = v`
    if *isotopyHelper(s1,reference, s2, v))* **then**
        `// See Algorithm 12`
        return true

end
return false

Algorithm 13: The method `isIsotopic()`

Bibliography

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

[2] James Belk, *Thompson's group F*, PhD thesis, Cornell University, 2004.

[3] James Belk and Francesco Matucci, *Conjugacy and Dynamics in Thompson's Groups*, preprint (2013).

[4] James W Cannon, William J Floyd, and Walter R Parry, *Introductory notes on Richard Thompson's groups*, Enseignement Mathématique **42** (1996), 215–256.

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

[6] D.S. Dummit and R.M. Foote, *Abstract Algebra*, John Wiley & Sons Canada, Limited, 2004.

[7] Jack Edmonds, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Standards Sect. B **71** (1967), 241–245.

[8] Victor Guba and Mark V Sapir, *Diagram groups*, American Mathematical Soc., 1997.

[9] Victor Sergeevich Guba and Mark Valentinovich Sapir, *On subgroups of R. Thompson's group F and other diagram groups*, Sbornik: Mathematics **190** (1999), no. 8, 1077.

[10] John E Hopcroft and Jin-Kue Wong, *Linear time algorithm for isomorphism of planar graphs (preliminary report)*, Proceedings of the sixth annual ACM symposium on Theory of computing, 1974, pp. 172–184.

[11] Nabil T Hossain, *Algorithm for the Conjugacy Problem in Thompson's Group F*, 2013, `http://www.asclab.org/asc/nhossain/conjugacyF`. Online; accessed 30-April-2013.

[12] Donald E Knuth, James H Morris, and Vaughan R Pratt, *Fast pattern matching in strings*, SIAM journal on computing **6** (1977), no. 2, 323–350.

[13] Klaus Madlener and Jürgen Avenhaus, *String Matching And Algorithmic Problems In Free Groups*, Revista colombiana de matematicas **14** (1980), 1-16.

[14] Bojan Mohar and Carsten Thomassen, *Graphs on surfaces*, Vol. 2, Johns Hopkins University Press Baltimore, 2001.

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

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

[17] Mark Allen Weiss and Susan Hartman, *Data structures and problem solving using Java*, Vol. 204, Addison-Wesley Reading, 1998.