INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Computational Experiments in Braids

by

Mike Brenner

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1998

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9820516

UMI Microform 9820516
Copyright 1998, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Copyright 1998

Mike Brenner

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

11/28/97 Michael Anshel
Date Chair of Examining Committee

12/9/97 Stanley Habib
Executive Officer

Supervisory Committee:

Dr. Christina Zamfiresco

Dr. Scott James

THE CITY UNIVERSITY OF NEW YORK

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Computational Experiments with Braids

by
Mike Brenner

Advisor: Professor Mike Anshel

ABSTRACT

Computational experiments were conducted on a class of problems in braid group theory. Using various data structures, certain braid group algoritihms were implemented, forming a braid group laboratory. During the course of the experiments, the impact of the various data structures was assessed and used to analyze and improve the performance of the algorithms on braid groups and on other groups. The effects of cohesion and coupling within briads were examined to relate the complexity of a braid to other forms of complexity. Examples and counterexamples were developed for the complexity of the algorithms needed to work with certain braids. One of the braid algorithms that developed from these experiments is a braid word problem eliminator (as opposed to a reducer) which approaches being a braid word problem solver. The states and patterns of this algorithm require space that is linear in the number of crossings. The worst case time is cubic in the length of the braid and linear in the number of strands. This is not as fast as the quadratic time in which the braid word problem is known to be solvable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

Some of the software used in this research was created for this paper, and some was downloaded from the Net. The authors of the software are acknowledged in the Braid Laboratory source code in the Magnus Archives URL.

The Braid Laboratory is Free software, subject to the General Public Library License of the Free Software Foundation (FSF). The FSF gcc compiler compiles a number of computing languages (FORTRAN, C, Ada, etc.) onto a number of targets (including personal computer and workstation operating systems). The examples and experiments in this paper were developed on a PC using the version of the gcc compiler at the Ezload URL, and on other hardware and other operating systems using other versions.

`\\http:magnus.ccny.ccny.edu`	Magnus Archives
`\\http:gnat.nyu.edu::\pub\gnat\ez2load`	Compiler Download
`gcc -c -g -O3 *.*`	Compile Command

Thanks to those who inspired me and those who provided comments on the drafts.

Part of this research was supported by:

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Section — Page

List of Tables ix

List of Figures x

1 The Patterns **1**

1.1 Introduction 1

1.1.1 What a Braid Looks Like 3

1.1.2 What a Braid Is 11

1.1.3 Historical 13

1.2 Patterns 14

1.3 FST Correctness Theorem 39

2 The Braid Elimination Data Structures **42**

2.1 Skeins and Algorithm A 42

2.1.1 Horizontal and Vertical Linkage Lemmas 43

2.1.2 Vertical Linkage Lemma 43

2.1.3 Eighty Uncovered Eliminations Lemma 44

2.1.4 Algorithm A Effectiveness Lemma 46

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.5 Algorithm A Quadratic Time Lemma 46

2.2 Algorithm F 47

2.2.1 Algorithm F Cubic Time Theorem 47

2.2.2 Stream Repetition Lemma 48

2.3 Completeness and Conjectures 49

2.3.1 Algorithm F Completeness 49

2.3.2 Algorithm F Completeness Corollary 49

2.3.3 $O(N^2)$ Conjecture 50

2.3.4 Future Research 50

3 The Braid Computational Experiments **51**

3.1 Introduction 51

3.1.1 Data Structures 52

3.1.2 Finite State Automata 54

3.1.3 Reduced Words 58

3.1.4 The Patterns in the Braid Lab Software 58

3.2 The Data Structure Experiments 59

3.2.1 Generating Random Objects 60

3.2.2 Data Structures for Group Algorithms 63

3.2.3 Summary of Data Structures 64

3.2.4 Linear Double Linked Lists 69

3.2.5 Time Saved By the Linked Lists 72

3.2.6 Circular Linked Lists 74

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.7 Building and Collapsing the Cayley Graph 83
3.3 Levels of Deception 84
3.3.1 Level 0 Deception for Pure Braids 84
3.3.2 Level I Deception for Local Reference Braids 85
3.3.3 Level II Deception for Semi-Global Reference 85
3.3.4 Preliminary Upper Bound 88
3.3.5 Level III Deception 90
3.3.6 Collapsing B, using the Blockage Lists 103
3.3.7 Conjugacy 106
3.3.8 FSA 110

4 Braid Complexity Metrics **116**
4.1 The Metrics 116
4.1.1 Heavy Cohesion and Cabling 116
4.1.2 Braid Coupling 117
4.1.3 Braid Counterexamples 121
4.1.4 Group Counterexamples 121
4.2 Software Maintenance 125
4.2.1 Coupling 127
4.2.2 Is there a Positive Return on Investment 129

5 Conclusion **134**

6 References **136**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

Number		Page
3.1	Sample Schedule Sizes	66
3.2	Swaps Following the Arrow Diagram Clockwise	70
3.3	Final DLL Non-Linked Algorithm	71
3.4	Cases for Moving D After B	72
3.5	Algorithm for Moving D After B	73
3.6	Linked List Searches	75
3.7	Post Conditions A	78
3.8	Interference vs. Reduction	86
3.9	Final FSA (Top Half Only)	110
3.10	Legend	112
4.1	Growth of PR-Braid	119
4.2	Counterexamples	119
4.3	Partial List of Braid Counterexamples	121
4.4	Partial List of Group Counterexamples	122
4.5	Braid Group Types	127
4.6	Metrics Collection Form	129

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

Number | Page

1.1 The Identity Braid on 5 Strands 4

1.2 The Braid abc with Crossings [1 2 3] 4

1.3 Braid abdD with the Crossing[1 2 4 -4] 4

1.4 Identity Braid 1 -1 5

1.5 Braid [3, 4, 3] or *aba* 5

1.6 Braid [4, 3, 4] or *bab* 5

1.7 Braid bdcbC where c=3 6

1.8 Vertical Loss of Light Cohesion 9

1.9 Horizontal Loss of Light Cohesion 9

1.10 The Braid (3 3 2 1 1) 10

1.11 The Smallest True Cable: *ccbbcc* 10

1.12 Given Braid cb^nCB 19

1.13 Braid $Bbcb^nCB \rightarrow$ 19

1.14 Braid $Bc^0bcb^nCB \rightarrow$ 19

1.15 Braid $Bc^1bcb^{n-1}CB \rightarrow$ 19

1.16 Braid $Bc^nbcb^0CB \rightarrow$ 19

1.17 Braid $Bc^nbcCB \rightarrow$ 20

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.18 Braid $Bc^nbB \rightarrow$ 20
1.19 Braid Bc^n 20
1.20 Finite State Automaton To Recognize the Patterns 35
1.21 Transducer for Pattern 1 36
1.22 Transducer for Pattern 2 37
1.23 Transducer for Pattern 3 37
1.24 Transducer for Pattern 4 38
1.25 Transducer for Pattern 5 38

3.1 FSA L (5 states) and M (3 states) 56
3.2 Methods Operating on FSAs 56
3.3 Antepenultimate FSA 57
3.4 DdcdCD 74
3.5 Singleton 75
3.6 Move Circle 76
3.7 Insert Circle 77
3.8 Insert Singleton Into Circle 78
3.9 Insert Circle Into Singleton 79
3.10 Insert Singleton After Singleton 79
3.11 Braid bdcbC where c=3 86
3.12 Braid bcdbC 87
3.13 Vertical and Horizontal Links 90
3.14 Two-Crossing Reductions 91

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.15 Old Algorithm (Inefficient Version) 99
3.16 First Optimization 100
3.17 Old Algorithm (More Efficient Version) 100
3.18 Algorithm using FSAs to replace the inner loop 101
3.19 Fourth Improved Algorithm with Blockage Lists 102
3.20 aB^nAB Arrivals 103
3.21 Proof of the Horrible Theorem 110

4.1 Function Estimate 118
4.2 Estimate of Growth of PR-Braid 118
4.3 Mitigating Sequence 120
4.4 Braid dbCDDccdccdCDBDD 121

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

The Patterns

1.1 Introduction

This paper documents the results of computational experiments investigating a class of problems in braid group theory. The focus of the experiments was on the role played by data structures and complexity metrics in speeding up braid group problems such as the word problem. The purpose of the investigation is to gain the following:

- a better notation for braid group relators to help understand why braid group theorems tend to have deceptively hard proofs, where non-obvious consequences of the relators are displaced by a large distance from their sources, as in Lemma 1.2.20,
- understanding of why certain braid algorithms cannot be done quickly in log space, as in Lemma 2.1.1,
- improvements in some braid group algorithms, and an attempt to construct an $O(N^2)$ braid word problem algorithm from the empirical results of the experiment, as in Lemma 2.1.4,
- an understanding of the limitations of experimental research in data structures to develop algorithms, discussed in Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- an appreciation of how braid groups may or may not apply to certain areas, such as cryptography (how hard braids are to crack using the word problem or the conjugate problem to create a one-way braid), and such as the relationship between a coupling metric in braid groups and a coupling metric in software maintenance processes (propagating bugs through global variables), discussed in Chapter 4.
- and development of a laboratory of Braid Lab Software to:
  - compute Cayley graphs out to the limit of a computer's virtual memory
  - to do braid crossing reductions
  - to do braid crossing eliminations, even to the point of solving the braid word problem
  - to comb a braid towards a particular minimum or towards the identity braid
  - to make attempts at finding words which conjugate a given word into another given word, and at solving the conjugacy problem (deciding whether two words are conjugate).

Abstract Data Types (ADTs) for common mathematical structures tend to have operations whose implementation is heavily dependent on having an efficient equality operator. As computing evolves, it works with increasingly complex ADTs, and the cost of equality operators on these more complex ADTs grows. On early machines it was simple (one assembler operation) and efficient (one CPU cycle with everything current in the cache) to determine the equality of ADTs implemented as exact numbers (extremely tiny integers limited, say, to an absolute value less than 2^{127} or whatever the electronics supports efficiently). Approximate numbers, whether implemented as fixed point or floating point), introduced algorithms that were ill-conditioned, in part because equality was no longer a binary operator, requiring knowledge of a precision within which to test the equality. Lack of knowledge of the correct precision makes problems like solving for the sides and angles of a triangle, aiming a rocket ship at Mars, or solving quadratic equations unreliable. Nevertheless this was not surprizing, as it was anticipated by engineering problems using manual approximations prior to electronic and optical computers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Another step was coherent compounding, permitting containers with multiple components of the same type, as in arrays. This introduced the problem of possible empty space between the components, so that, for instance, due to ignorance of the value of that uninitialized empty space, two arrays each equal to [103, 107, 96] might not be bitwise equal to each other[1]. Recursive ADTs nesting ADTs inside each other, such as sets within sets [**1**], can involve extremely time-consuming algorithms to test for equality of objects. The exponential time taken by naive equality algorithms of combinatorial structures tempts implementors to use metrics, such as hash codes, to get a quick approximation which eliminates most instances quickly. In some ADTs, the worst case may remain exponential or NP-complete. But the trend to more difficult equality testing times does not stop there at exponential times.

Mathematicians now use computers to investigate properties of mathematical ADTs with unsolvable equality operators, for which there is not, and will never be an algorithm to check for equality that is guaranteed to finish in a finite amount of time. One example is comparing the words in mathematical groups. A particular kind of group, the braid group is theoretically capable of determining equality in a polynomial (in this case, quadratic [**2**]) amount of time, but designing an algorithm and the data structures that will achieve quadratic time in the worst case is not trivial.

1.1.1 What a Braid Looks Like

Braid groups are motivated as sets of braids with K strands under the operation of concatenation. Additional concepts of crossings and beads are used to give the following geometric

[1]This could occur if the array is represented as an array of 4-bit twos-complement numbers centered on 100 to save storage like the Year 2000 Problem, but, for efficiency of loading and storing, are the lower 4 bits of 128 bit words in a computer. To get their actual value, the 4 bits must be cracked off, for example, by using an **and** operation, and the sign must be extended. Comparing the array involves similarly cracking and sign-extending each of the elements of the array. This effect worsens with data structures comprising non-homogeneous parts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

representation of a braid drawn horizontally.

The Identity Braid ε on K strands is the braid with no crossings. It is represented as K horizontal, parallel strands (of wire, say) pegged down at both ends. We will shortly define a way of identifying braids which are equal to the identity braid, but which are not the identity braid. The identity braid on 5 crossings looks like Figure 1.1.

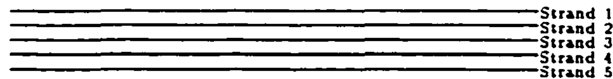


Figure 1.1: The Identity Braid on 5 Strands

A positive crossing, like those in Figure 1.2, is represented by a strand crossing over the next strand below it, with the slope of the top strand being positive. This matches some of the intuitive characteristics of physical braids.

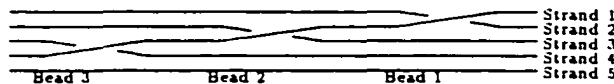


Figure 1.2: The Braid abc with Crossings [1 2 3]

The representation can be enhanced by visualizing a colored place-keeping bead put on a strand at the point where a crossing occurs. The bead's color designating whether it is a positive or negative crossing. The braid [1 2 4 -4] or *abdD* in Figure 1.3 shows that the beads are numbered from right to left.

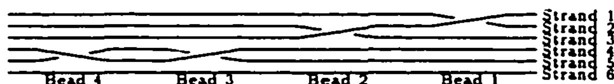


Figure 1.3: Braid abdD with the Crossing[1 2 4 -4]

Since the ends of the physical braid are pegged down, no strand has a free end, and physical

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

braids can only accept two kinds of maneuvers: commutative *cancelling pairs*, and semi-commutative *flips*.

A *cancelling pair* is when a strand G that is currently on top of (or, respectively, beneath) another strand H is minimally pulled below (resp. above) H, creating one positive and one negative crossing. With the *bead number* as the index of the crossing in the braid, and with braids drawn with bead number 1 at the right, bead number 2 to the left of bead number 1, etc., the braid $[1, -1]$, which is equal to the identity braid, is shown in Figure 1.4.

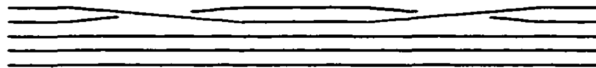


Figure 1.4: Identity Braid 1 -1

A flip[2] occurs when a braid G that is currently on top of (or, respectively, beneath) another braid H is minimally flipped over remaining on top of (resp. beneath) H. This is shown in Figures 1.5-1.6.

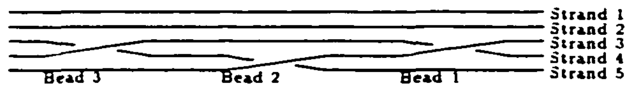


Figure 1.5: Braid $[3, 4, 3]$ or aba

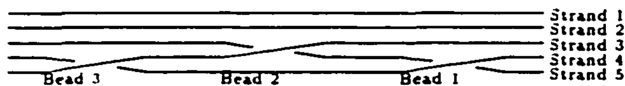


Figure 1.6: Braid $[4, 3, 4]$ or bab

A commutative move is permitted when two crossings do not share a strand. For example, in Figure 1.7 the first 2 crossings, 2 and 4, that is, b and d may commute, since b is involved with only strands 2 and 3, while d is involved with strands 4 and 5. Physically these can be

[2]Later to be called an $aba = bab$ relator.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pushed past each other without *interference.* The fact that some strands do interfere with each other is the source of calling braids and some braid relators semi-commutative.

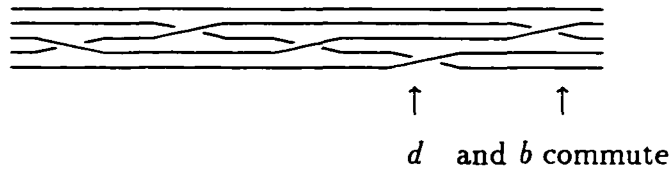


Figure 1.7: Braid bdcbC where c=3

Braid are equal when sequences of these moves take them into each other. As often occurs in group theory, these moves producing equal braid representation correspond to relators producing equal braids.

To create a non-identity braid in this representation, crossings must be performed **before** pegging down the ends. After being pegged down at the ends, these strands with their crossings represent a mathematical braid. Most crossings in most braids cannot be uncrossed without breaking or unpegging a strand, which are forbidden in these braids. That is another was of saying that most braids are not equal to the identity braid.

This representation provides an x-y coordinate system. The x-coordinate is the unique order of the bead horizontally counting the rightmost crossing as 1. Because of the braid relators in Equation 1.1 below, two crossings (beads) at the same bead number would only be ambiguous if they were adjacent, that is a p, $p+1$ pair. However, even for non-ambiguous pairs, the theorems and algoriths are easier to describe under unique bead x-coordinates. The y-coordinate of a crossing is the strand number crossing over or under the strand beneath it, counting the first (top) strand as 1. This is equivalent to a different representation which has an infinite number of strands, only a finite number of which have crossings, if the first strand with a crossing is called the first strand, strand $K-1$ is the last strand that crosses the strand below it, and the strands above strand 1 and those below strand K are ignored.

Both x- and the y-coordinates in this representation are unusual. The x-coordinate, always positive, is unusual because

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it starts at the right. The y-coordinate is unusual because negative crossings have a special meaning. A positive crossing is where the slope of the strand that is on top in this representation of the crossing is positive, when moving from right to left along the braid. A negative crossing is where the slope is negative. Consequently, a positive crossing p twists strand p over strand $p+1$, and a negative crossing $-p$ twists strand p under strand $p+1$. Further, in a braid of N strands there can be no crossing $N+1$ (or its negative), since there is no strand $N+1$ for strand N to cross over.

The existence of a crossing blocks relators on the strand it crosses, and it also blocks relators on other neighboring strands. To handle a single application of the semi-commutative relators (such as $pqp = qpq$), the representation must include the effects of blocking the strand above and below the strand whose number is the crossing number. In order to handle multiple applications of the semi-commutative relators, the representation must include the effects of blocking additional strands. This is done using the beads. At each bead number (x-coordinate) there is a bead of one of the following three bead types: *original crossings*, *fences*, and *eliminated crossings*.

The *original crossings* are beads representing the numbers from the sequence of crossings in a given K-strand braid. An elimination theorem or algorithm replaces certain original crossings and fences with *eliminated (inactive) crossings*. If all original crossings become eliminated crossings, then the word problem is solved, and the given braid can be declared to be equal to the identity braid.

The *fences* are additional beads added for each original crossing. The experiments in this paper and the lemmas proven below show that (except for original crossings on strands 1, 2, $K-1$, and K) four fence beads are needed for each crossing to model the effects a crossing has on blocking the use of relators on neighboring strands. Thus a braid crossing, say -512, is represented as an original crossing -512, and fence crossings -510, -511, -513, and -514, because a crossing on strand 512 can, under various circumstance shown in Lemma 1.2.20, block relators on those four strands, in addition to strand 512.

End effects are imaginary beads, as follows. Crossings on strand 1 (that is, the crossings

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1 and -1) can be thought of as having fence beads on the non-existent strands 0 and -1, crossings on strand 2 on the non-existent strand 0, and crossings on strand n on the non-existent strand $N+2$. In addition, crossings on strand $n-1$ and n can be thought of as having impossible fences on strand n, which can not be a crossing, because a crossing n would have to cross over or under the non-existent strand $n+1$.

Ignoring these end effects, there are $5n$ beads ($4n$ fence beads) in the representation of a braid with n crossings. Including these end effects, there are between 0 and $4n$ fence beads. The lowest case 0, occurs when all of the crossings are simultaneously on strand 1 and strand $n-1$, that is, in a 2-strand braid. In summary, it takes $O(n)$ fences[3] in addition to the n original crossings to represent a braid.

The original crossing bead and its 4 corresponding fence beads, always span 5 consecutive strands, called the *floating 5-strand window*. It is convenient to abbreviate the strand numbers in floating 5-strand windows using the letters *abcde* for positive crossings and $ABCDE$ for negative crossings. In the example above, the crossing -512 is negative, so beads negative beads are placed on strands 510 through 514. These negative beads have the numbers -510 through -514, but they are abbreviated ABCDE. Thus strand 510 has a bead A, and so forth, down to strand 514 getting a bead E.

How to Measure Braids: Braid Cohesion

Braid cohesion is related to the portion of themselves they cross. There are two kinds of braid cohesion metrics, light cohesion and heavy cohesion.

[3]These fencing beads will evenutally (Lemma 1.2.17) force the patterns of the relators used in the braid algorithms to become more complex when used on the fenced beads instead of just the original crossing. The complexification of the relators is accomplished by Tietze Transformations of the group representation which add additional possible reductions, but at the expense of additional group elements. The play between elements and relators is critical for speed and determines the nature of the data structures needed to get that speed in algorithms based on a particular set of Tietze Transformations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Light Cohesion

Light cohesion is simply the number of strands involved in crossings with others strands. If the light cohesion is less than the number of strands, then the braid acts like a disconnected set, that is, it seems to fall apart vertically. We say that it loses vertical cohesion, as in the braid (1 10 15 2 11 1 10 -2 -15 -11 -1 -10 -2 -11). That means that Word and conjugate problems can be broken into two smaller problems on independent strands. Similarly, some braids are factorable into identities horizontally, and the parts could be processed separately, as in the Braid (1 -1 20 -20 1 40 -40 1). This is called the loss of horizontal cohesion. These look a little complicated in log space (for example, in the format just given). However, when visualized geometrically, as in the skein data structures, it is easier to see that they are factorable vertically (Figure 1.8) and horizontally (Figure 1.9). The human eye acts like a skein in that its pattern recognition capability can follow simple groups of strands that have no interference from neighboring strands. However, as the cohesion increases, the human senses become less reliable and eventually useless for factoring braids.

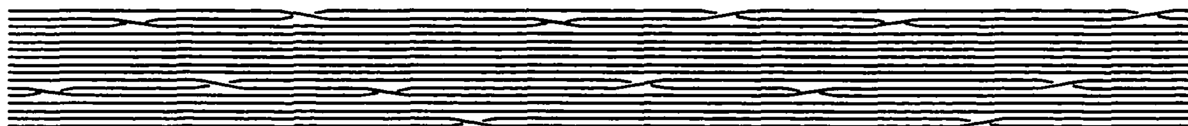


Figure 1.8: Vertical Loss of Light Cohesion

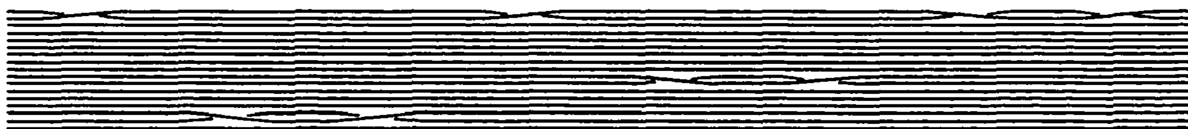


Figure 1.9: Horizontal Loss of Light Cohesion

It is expected, that when the number of crossings is significantly larger than the number of strands, then cohesion will tend to be high, since the probability of a vertical loss of cohesion is the probability that a strand never crosses. Similarly, the probability of a horizontal loss

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of cohesion is the probability that there is a crossing (bead) that partitions the braid such that there is an identity on both sides of that crossing, which gets less likely as the number of strands increases in proportion to the length of the braid.

Heavy Cohesion

Heavy cohesion is the number of instances of a strand a wrapping around another strand b. That is, a goes over b, then immediately goes under b, or in the reverse order. Heavy cohesion instances sometimes form paths through the braid in pairs, forming cohesive segments. Each cohesive segment includes two heavily cohesive instances, together with all possible intermediate crossings. For example, (3 3 2 1 1) in Figure 1.10 forms a cohesive segment from 3 to 1, because strand 3 wraps over 4, under 3 through 1, and back over 1, symbolically enclosing those strands in its wings.

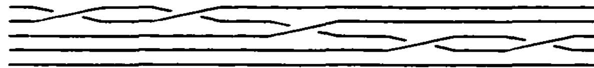


Figure 1.10: The Braid (3 3 2 1 1)

The heaviest cohesion is true cabling, where a strand wraps completely around a number of other strands. This is a consecutive pair of cohesive segments on the same pair of initial and terminal strands. The smallest example is (3 3 2 2 3 3) in Figure 1.11, which is *ccbbcc*.

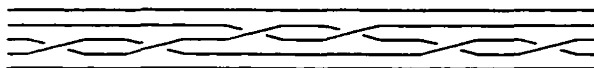


Figure 1.11: The Smallest True Cable: *ccbbcc*

In a braid, there can be regions of very dense heavy cohesion or cabling which can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

examined apart from the rest of the braid, or the metric could be computed on the whole braid.

The elimination lemmas and theorems below. and the elimination algorithmic experiments in the next Chapter were driven by the goal of reducing the cohesion of braids to the point where they reveal their nature as identity braids, if they are indeed identity braids. This contrasts with reduction algorithms which increase the cohesion until the braids reach a so-called "normal form" according to certain theorems (not discussed in this paper) would then apply.

1.1.2 What a Braid Is

A braid group is a finitely presented group [**3**] and [**4**]. The braid group on K strands has the presentation

$$< C \mid R_{\frac{c}{2}} \cup R_{p^2} > \tag{1.1}$$

where

- C is the set of positive crossings $\{\tau_1, \tau_2, ..., \tau_{K-1}\}$;
- group multiplication is concatentation of braids;
- the semi-commutative relators (which permit all crossings to commute unless they are on adjacent strands) are the words:

$$R_{\frac{c}{2}} = \{\tau_i\tau_j = \tau_j\tau_i \mid j > i + 1, i \in (1..K - 3)\}; \tag{1.2}$$

- the twisting relators (which permit certain sequences of relators to intermingle and thus make the braid word problem interesting) are the words:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$R_{p^2} = \{\tau_i\tau_j\tau_i = \tau_j\tau_i\tau_j, i = j + 1, j \in (1..K - 2)\}. \tag{1.3}$$

.

These relators in Equation 1.1 are responsible for:

- all the attributes of braid groups, braid algorithms, and the braid abstract data types needed to reduce or cancel braids:
- the ease or difficulty of the word, conjugate, and isomorphism problems:
- all requirements for algorithmic steps or data structures:
- the form of the patterns used for braid reduction [**5**]
- the form of the patterns used for braid elimination implemented in this paper and in the Braid Lab Software.

[**6**] presents proofs that braid groups have the following additional properties:

- a solvable word problem.
- a quadratic isoperimetric inequality.
- a quadratic time word problem solution, possibly using non-log space,
- a log space word problem solution. possibly using greater than quadratic time.
- a solvable conjugacy problem.
- a minimal normal form which, unfortunately, is co-NP-complete [**7**], for a varying number of braids. but which is polynomial for a fixed number of braids [**8**].
- The braid group is not a small cancellation group [**9**].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.1.3 Historical

Algorithm F [The Elimination Method presented in this paper] has worst case time $O(N^3)$ and conjectured to be $O(N^2)$. The formulation of this algorithm gives new insights into why many braids are quick to solve, while others are deceptively difficult. Algorithm F is advancing, in the sense that each elimination move of the algorithm brings an identity braid one step closer to the null braid. Algorithm F always [Theorem 1.3.1] eliminates crossings in pairs, that is, two at a time. Advancing elimination will not shorten all braids to a unique minimum [**10**], because there can be local minima to which it might shorten them, and because the set of unique minimum braids is co-NP-complete. Nevertheless, advancing braid algorithms provide one method to identify identity braids. Algorithm F takes time conjectured to be $O(N^2)$ but proven [Theorem 2.2.1] to be bounded by $O(N^3)$, when the given data structures are used. An important trade-off in Algorithm F, in order to circumvent the combinatorial number of possible braid relators to apply at each crossing of a braid, is that Algorithm F uses auxiliary data structures of a size linear in the length of braid being processed, and therefore does not operate in Log Space.

This contrasts with reduction to normal form, achieved by other braid reduction algorithms, which almost always lengthens the braid in order to achieve that form. In addition to lengthening versus shortening the braid, there is a fundamental difference between Algorithm F and other published braid algorithms of any speed, which is the different guidance strategy. When Algorithm F eliminates a pair of crossings in pairs, it does so with no strategy of ordering the crossings on the braid, for example, to put the negative crossings on one side and the positive crossings on the other side. Rather, Algorithm F simply organizes the application of the set of patterns discovered by the experiments, with the only goal of finding the most efficient path to the next pattern which can be used to eliminate another pair of crossings (modulo application of an arbitrary number of braid group relators which are inherent in the patterns).

Several publications give algorithms to solve the braid group word problem (or, equivalently,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

comparing braids, since braids P and Q are equal precisely when $PQ^{-1} = \varepsilon$). Some of these publications contain conjectures that those algorithms are particularly fast. However, none of those publications contain proofs that any of these algorithms is faster than Algorithm F at either its proven speed or its speculated speed. However, it is known that an $O(N^2)$ algorithm can be built. It is possible that applying the data structures and patterns from this paper may enable a proof that existing algorithms already run in $O(N^2)$.

1.2 Patterns

Tietze Transformations

The experiments discussed in Chapter 3 motivated changes called *Tietze Transformations* in the braid group presentation. The transformed presentation then contained additional relators. These new relators are the patterns discussed in this Section.

Definition 1.2.1 (Group Isomorphism). *Let $G =< A_g, * >$ and $H =< A_h, \odot >$ be groups. G and H are* **isomorphic** *if there is a mapping ϕ between their objects that preserves the operation, that is $g * h = \phi(g) \odot \phi(h), \forall g, h \in G$.*

Definition 1.2.2 (Tietze Transformation). *Given a finitely presented group G where $G =< \Gamma \mid R >$, a* **Tietze Transformation** *is an isomorphism of G that adds or deletes a single generator or a single relator.*

Tietze Transformations [**11**] are used to translate from one group presentation to another. If the word problem for a group is unsolvable, it may take an infinite number of guesses at Tietze Transformations to arrive at the particular presentation being sought, or even to determine that a particular presentation is equivalent to a given presentation. This makes it conceivable that there could be a methodology similar to Fermat's Little Theorem that would make groups with unsolvable word problems serve as an encryption key, when

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

combined with large sequences of Tietze Transformations. The decoding key would be a co-domain of the key placed through the inverse Tietze Transformation in the reverse order. If the sequence was long enough and random enough, the intractability or impossibility of inverting the sequence (in the worst case) might be provably greater than the current effort required to factor very large prime numbers, the basis of some modern encryption algorithms. A cryptographic application of Tietze transformations, is then encoding messages by changing presentations, thus transforming the way the words or the letters in the word appear, a process that may or may not be invertible in a finite amount of time, if you do not know the exact Tietze Transformations which were applied, and in which order. Cryptography has a goal to make it as difficult to decode without the key as possible, while keeping a reasonable encoding and decoding time with the key.

The patterns in Lemma 1.2.8 represent the opposite goal. In order to make it as easy as possible to translate a braid that is equal to the null braid (the identity) into a form that has no crossings, the relators of an ordinary braid group are transformed by adding group elements (additional crossing-like object called fences), with additional relators. It turns out, with the new elements and relators, braid elimination can proceed faster than in algorithms based on the untransformed braid group presentation.

Lemma 1.2.3. *Let G be the group $< \Gamma \mid R >$.*

A Tietze Transformation ϕ changing $< \Gamma \mid R >$ into $< \Gamma' \mid R' >$ exists iff there is a sequence of moves of the following 4 types:

- *Adding a Generator: If $\Gamma' = \Gamma \cup \{\alpha\}$ then $R' = R \cup \{\alpha^{-1}\beta\}$, where $\alpha \notin \Gamma, \beta \in (A^* - \varepsilon)$.*
- *Deleting a Generator: If $\Gamma' = \Gamma - \{\alpha\}$ then $R' = R - \{\alpha^{-1}\beta\}$, where $\alpha \in \Gamma$, $\beta \in < \Gamma - \{\alpha\} \mid >$, and $\alpha^{-1}\beta$ is the ONLY member of R containing the generator α in its product list.*
- *Adding a Relator: If $R' = R \cup \{r\}$, then $r \in \overline{R} - R$.*
- *Deleting a Relator: If $R' = R - \{r\}$, $r \in R \cap \overline{R - \{r\}}$.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note that this lemma does not guarantee that these conditions can be checked algorithmically.

Lemma 1.2.4. *Any finite presentation of a group can be obtained from the other by a finite sequence of Tietze Transformations. There is no algorithm for determining whether two given finite presentations yield isomorphic groups.*

Proof. Proved in [**12**]. □

Theorem 1.2.5. *The braid group word problem is solvable in $O(N^2)$ time.*

Proof. Proved in [**13**]. □

Corollary 1.2.6. *Therefore, there is a way of determining the isomorphism of the group in Lemma 1.2.4, and that will lead to effective elimination algorithms.*

Lemma 1.2.7. *If the word problem is solvable, the problem of whether two words in G are equal to each other is solvable, and vice versa.*

Proof. $\Rightarrow$ Let x and y be words in group G with solvable word problem. Then x and y are equal, iff $xy^{-1} = \varepsilon$, which is solvable because it is the word problem. Concatenation can be done in time linear in the length of the words being concatenated, and so does not change the finiteness of the computation. $\Leftarrow$ if there is an algorithm to solve whether two words are equal, the same algorithm can be used to solve the word problem for word w by checking whether $w = \varepsilon$. □

Mathematical Induction Notation

The following derivations use mathematical induction in the following manner. Let V, X, Y, and Z be words in a braid group (say, the braid group with K strands). Then VY^iXYX^jZ

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is also a word in that braid group for all natural numbers i and j. All of the inductions below maintain the invariant condition that $i + j = m$, for a given m. Thus, the Inductive Start establishes, a fact for the word $VXYX^nZ$ which equals VY^0XYX^nZ. Then the Inductive Step establishes that for all natural numbers i, if the fact is true for $VY^iXYX^{n-i}Z$ then it is also true for $VY^{i+1}XYX^{n-i-1}Z$.

In all cases, this Inductive Step is established by a single application of a relator of the type $bcb = cbc$, which substitutes YXY for XYX in VY^iXYX^jZ which equals $VY^iXYXX^{j-1}Z$ to get $VY^iYXYX^{j-1}Z$ which equals $VY^{i+1}XYX^{j-1}Z$. This maintains the invariant condition that $i + j = n$.

The Inductive Finish concluded the fact is true for VY^nXYX^0Z which equals VY^nXYZ.

Pattern Equivalency Lemma

Lemma 1.2.8. *Taking n to be a natural number and abcde to be a floating 5-strand window on a braid replacing each of the following Elimination Patterns leaves braids equivalent, and, in particular, they leave identity braids identity braids:*

1. $cb^nCB \to Bc^n$
2. $cbc^nB \to b^nc$
3. $cbC^nB \to B^nc$
4. $cB^nCB \to BC^n$
5. $cC \to \varepsilon$

Proof. These eliminations are derivable from the braid group relators, as shown in the derivations below, so they follow from repeated applications of $xX = Xx = \varepsilon$, $xy = yx$, $xyx = yxy$,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and $XYX = YXY$. In proving these patterns, identity pairs such as xX and other artifacts[4] are added, in order to cause future eliminations.

1. Show: $cb^nCB \rightarrow Bc^n$ (symmetrical to $cd^nCD \rightarrow Dc^n$)

   $cb^nCB \rightarrow$ (Figure 1.12)

   $Bbcb^nCB \rightarrow$ (add an identity Bb to prime the pump, Figure 1.13)

   $Bc^0bcb^nCB \rightarrow$ (add another identity c^0 to start the induction on n, Figure 1.14)

   $Bc^1bcb^{n-1}CB \rightarrow$ (inductive step using bcb=cbc, Figure 1.15)

$\vdots$

   $Bc^nbcb^0CB \rightarrow$ (finish the fixed induction, Figure 1.16)

   $Bc^nbcCB \rightarrow$ (eliminate the identity b^0, Figure 1.17)

   $Bc^nbB \rightarrow$ (eliminate the identity cC, Figure 1.18)

   Bc^n (eliminate the identity bB, and complete this portion of the proof, Figure 1.19)

---

[4]Adding these artifacts would add a significant time and space penalty during execution. However, these artifacts add no time or space penalty here in the proof. The fact that these patterns automatically **encode** these artifacts, therefore, forms the basis of the speed of the algorithm. Compressing multiple patterns together is basically what makes any algorithm tractable even though there are an exponential number of permutations, when the data structures shown in Section 2 are used. This observation could also become the basis of using these patterns to speed up other Braid Algorithms (see Future Research Directions below).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

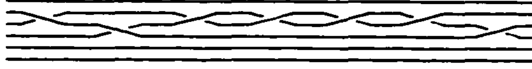


Figure 1.12: Given Braid cb^nCB

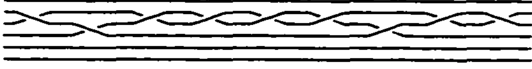


Figure 1.13: Braid $Bbcb^nCB \rightarrow$

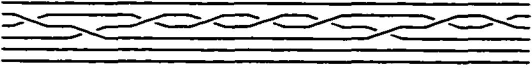


Figure 1.14: Braid $Bc^0bcb^nCB \rightarrow$

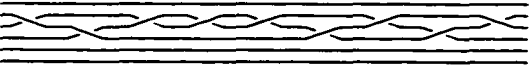


Figure 1.15: Braid $Bc^1bcb^{n-1}CB \rightarrow$

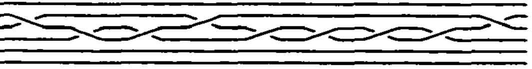


Figure 1.16: Braid $Bc^nbcb^0CB \rightarrow$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

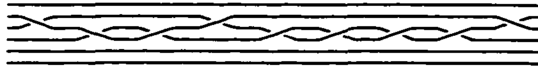


Figure 1.17: Braid $Bc^nbcCB \rightarrow$

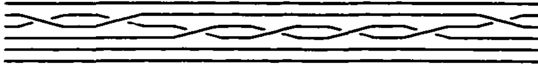


Figure 1.18: Braid $Bc^nbB \rightarrow$

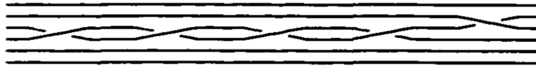


Figure 1.19: Braid Bc^n

2. Show: $cbc^nB \rightarrow b^nc$ (symmetrical to cdc^nD)

   $cbc^nB \rightarrow$

   $b^0cbc^nB \rightarrow$ (start the induction on n)

   $b^1cbc^{n-1}B \rightarrow$

   $\vdots$

   $b^ncbc^0B \rightarrow$ (complete the induction)

   $b^ncbB \rightarrow$

   b^nc

3. Show: $cbC^nB \rightarrow B^nc$

   $cbC^nB \rightarrow$

   $cbC^nBCB^0c \rightarrow$ (start the induction on n)

   $cbC^{n-1}BCB^1c \rightarrow$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\vdots$

$cbC^0BCB^nc \rightarrow$ (complete the induction)

$cbBCB^nc \rightarrow$

$cCB^nc \rightarrow$

B^nc

4. Show: $cB^nCB \rightarrow BC^n$ (symmetrical to cDC^nD)

   $cB^nCB \rightarrow$

   $cB^nCBC^0 \rightarrow$ (start the induction on n)

   $cB^{n-1}CBC^1 \rightarrow$

   $\vdots$

   $cB^0CBC^n \rightarrow$ (complete the induction)

   $cCBC^n \rightarrow$

   BC^n

5. $cC \rightarrow \varepsilon$ follows directly from the fact that braids are groups.

□

For algorithmic design purposes, the eliminated crossings are always the first and last crossing of each pattern, and the substitution should be thought of as changing the internal crossings of the patterns, in order to gain the speed advantage provided by Lemma 1.2.20 below. The $ABCDE$ strands shift right and left across the full set of strands. For each crossing, one c or C bead is added onto the current strand, for a positive or negative crossing respectively. Similarly, one b or B bead is added on the strand above (unless the crossing is strand A, and there is therefore no prior strand), and the same for fences A, D, and E. The reason these 5 beads are called fences is because they block patterns from being recognized along a strand, while at the same time, permitting patterns to be recognized by looking only at the current strand, because they encode all possible interferences from neighboring strands.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Shiftable Patterns Lemma

Lemma 1.2.9. *Shifting the elimination patterns of Lemma 1.2.8 to the next left or next right strand also eliminates correctly.*

Proof. Since none of the patterns involve the strand A or E, shifting the pattern to the left or right one strand does not affect the validity of substituting according to the pattern. For example, the first pattern is $cb^nCB \rightarrow Bc^n$, so we also have $dc^nDC \rightarrow Cb^n$. The limited symmetry causes the example to act exactly like its neighbor did in Lemma 1.2.8:

$dc^nDC \rightarrow$

$Ccdc^nDc \rightarrow$

$Cd^0cdc^nDC \rightarrow$ (start the induction)

$\vdots$

$Cd^ncdc^0DC \rightarrow$

$Cd^ncC \rightarrow$

Cd^n

Thus

$$\begin{array}{lcl} cd^nCD & \rightarrow & Dc^n \\ cdC^nD & \rightarrow & d^nc \\ cdC^nD & \rightarrow & D^nc \\ cD^nCD & \rightarrow & DC^n \end{array}$$

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reversible Patterns Lemma

Lemma 1.2.10. *Forward to backwards symmetries of the elimination patterns of Lemma 1.2.8 leave braids equivalent.*

Example: The first pattern is $cb^nCB \to Bc^n$, so we also have $BCb^nc \to c^nB$, and by Lemma 1.2.9, $CDc^nd \to b^nC$.

Proof. Since all of the braid group relators are forwards-backwards symmetric, they can be applied forwards and backwards, and therefore all braid eliminations are valid on the same braid reversed forwards to backwards. This limited symmetry causes the example to act exactly like its reverse did in Lemma 1.2.8:

$BCb^nc \to$

$BCb^ncbB \to$

$BCB^ncbc^0B \to$ (start the induction)

$BCb^{n-1}cbc^1B \to$ (inductive step)

$\vdots$

$BCb^nbcc^nB \to$ (finish the induction)

$BCcbc^nB \to$

$Bbc^nB \to$

c^nB

Thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
CBc^nb &\rightarrow b^nC \\
Cb^ncb &\rightarrow bc^n \\
CB^ncb &\rightarrow bC^n \\
CBC^nb &\rightarrow B^nC
\end{aligned}
$$

□

Capitalizable Patterns Lemma

Definition 1.2.11 (Conjugate). *A conjugate of a word w in a group G is a word formed by multiplying on the left by a particular word $t \in G$, and on the right by t^{-1}, that is, twt^{-1}.*

Definition 1.2.12 (Conjugate Closure). *The conjugate closure of a set of words R is the set $\overline{R}$ of all conjugates of R.*

Note that if the elements of R happen to be chosen to all be equal to ϵ, then $\overline{R}$ would include all products of elements of R.

All groups are closed under inverses. The symmetric nature of the braid group's relators permit the group to be generated by the negative generators (the inverses of the generators), as well as the generators themselves.

Lemma 1.2.13. *Capitalization symmetries of the elimination patterns of Lemma 1.2.8 leave braids equivalent.*

Example: The first pattern is $cb^nCB \rightarrow Bc^n$, so we also have $CB^ncb \rightarrow bC^n$.

Proof. Since, in any group, all conjugates and inverses of all relators are also relators, the braid relators $ab = ba$ and $aba = bab$ are conjoined with the braid relators $AB = BA$ and $ABA = BAB$, as follows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$ab = ba \rightarrow (ab)^{-1} = (ba)^{-1} \rightarrow BA = AB$

$aba = bab \rightarrow (aba)^{-1} = (bab)^{-1} \rightarrow ABA = BAB$

While all rotations are relators because they are conjugates. in most groups capitalization symmetry only exists for certain patterns. However. with the braid group. these 4 patterns (only) generate patterns that are capitalization symmetric because in these 4 patterns. capitalization symmetry happens to be equal to rotational equality followed by a shift. In the case of the example:

$CB^ncb \rightarrow$

$bBCB^ncb \rightarrow$ (add an identity to prime the pump)

$b^0BCB^ncb \rightarrow$ (add another identity start the induction)

$bC^1BCB^{n-1}cb \rightarrow$ (inductive step)

$\vdots$

$bC^nBCB^0cb \rightarrow$ (finish the induction)

$bC^nBCcb \rightarrow$ (eliminate identity)

$bC^nBb \rightarrow$ (eliminate identity)

bC^n (eliminate identity and complete this portion of the proof)

Thus

$$\begin{array}{lcl} CB^ncb & \rightarrow & bC^n \\ CBC^nb & \rightarrow & B^nC \\ CBC^nb & \rightarrow & b^nC \\ Cb^nCb & \rightarrow & bc^n \end{array}$$

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Rotatable Patterns Lemma

Lemma 1.2.14. *Rotational symmetries of the elimination patterns of Lemma 1.2.8 leave braids equivalent.*

Example: The first pattern is $cb^nCB \to Bc^n$, so we also have $cd^nCD \to Dc^n$.

Proof. By Lemmas 1.2.9, 1.2.10, and 1.2.13 the $aAeE$ portions of any pattern may be swapped as a whole with $eEdD$ respectively.

Thus

$$\begin{array}{lcl} cd^nCD & \to & Dc^n \\ cdc^nD & \to & d^nc \\ cdC^nD & \to & D^nc \\ cD^nCD & \to & DC^n \end{array}$$

□

Divisible Pattern Negative Exponents Lemma

Lemma 1.2.15. *Replacing the positive exponents in the elimination patterns of Lemma 1.2.8 with negative exponents will leave the braids equivalent.*

Proof. This is just notational. It happens that formal replacement of B^n with b^{-n} and C^n by c^{-n} leaves the derivations intact. Thus the four patterns work with n any integer, and therefore can be represented by only two patterns plus the trivial pattern.

Note: it is known that these two patterns can further be combined into a single pattern $cbc^n = b^ncb$ which would shorten the above derivations, but would not help in visualizing where the complexity of braid elimination is, nor in setting up an efficient FSA.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus the 5 original patterns are equivalent to the following 3:

$$\begin{aligned} cb^nCB &\rightarrow Bc^n \\ cbc^nB &\rightarrow b^nc \\ cC &\rightarrow \varepsilon \end{aligned}$$

It turns out that in programming Algorithm F it does not help to notice this. Noticing this cuts the number of states in half, but at the expense of the time complexity of another layer of code which interprets whether to invert the exponent (and therefore the second and third elements of the 4-element pattern). In an $O(N^2)$ algorithm with linear space, time is much more valuable than space.

These three patterns are further equivalent to only 2, for example, by changing $cbc^nB \rightarrow b^nC$ to $cbc^n \rightarrow b^nCb$, which can be proven by the following derivation.

$cbc^n \rightarrow$

$b^0cbc^n \rightarrow$

$b^1cbc^{n-1} \rightarrow$

$b^ncbc^0 \rightarrow$

b^ncb

Now, from this one pattern, $cbc^n \rightarrow b^nCb$, the four non-trivial patterns from Lemma 1.2.8 can be derived.

1. $cbc^n \rightarrow b^ncb$

   $cbc^{-n} \rightarrow b^{-n}cb$

   $cbC^n \rightarrow B^ncb$

   $bC^n \rightarrow CB^ncb$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$Bc^n \rightarrow cb^nCB$

2. $cbc^n \rightarrow b^ncb$

   $c^n \rightarrow BCb^ncb$

   $c^nB \rightarrow BCb^nc$

   $c^nD \rightarrow DCd^nc$

   $b^nC \rightarrow CBc^nb$

   $B^nc \rightarrow cbC^nB$

3. $cbc^{-n} \rightarrow b^{-n}cb$

   $cbC^n \rightarrow B^ncb$

   $bC^n \rightarrow CB^ncb$

   $C^n \rightarrow BCB^ncb$

   $C^nB \rightarrow BCB^nc$

   $c^nb \rightarrow bcb^nC$

   $c^nd \rightarrow dcd^nC$

   $b^nc \rightarrow cbc^nB$

4. $cbc^n \rightarrow b^ncb$

   $bc^n \rightarrow Cb^ncb$

   $BC^n \rightarrow cB^nCB$

□

Length Four Completeness Lemma

Lemma 1.2.16. *No other patterns of length 4 lessen the number of crossings.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The relators for the braid group, as usually written (e.g., $aba = bab$), are intuitively convenient and pedagogically sound, but not presented in a manner that minimizes the number of states in the FSAs which recognize them, nor are they presented in a manner that makes apparent the difficulties in recognizing patterns in them. Rewriting the relators (using the variables ABDE to represent the strands next to the current strand C) by carrying them out one step on either side happens to give a set of symmetries which are easier to see where the difficulties lie and, therefore, easier to design correct, efficient, recognizers.

Proof. Ignoring shift symmetry (b versus d), there are combinatorially exactly 16 constrained patterns beginning with strand c, because there are 2 possibilities for the first letter (c or C), there are 2 possibilities for the second component (b or B), there are 2 places to put the exponent (after the second or third component), and there are 2 possibilities for the third component (c or C). These result (combining the tables for Lemma 1.2.8 and Lemma 1.2.13) in the following 8 right sides which eliminate two crossings from the original.

$$\begin{array}{ll} Bc^n & bC^n \\ b^nc & B^nC \\ B^nc & b^nC \\ BC^n & bc^n \end{array}$$

The 8 patterns which give these eliminations which contract the length of the braid are:

$$\begin{array}{ll} cb^nCB & CB^ncb \\ cbc^nB & CBC^nb \\ cbC^nB & CBc^nb \\ cB^nCB & Cb^ncb \end{array}$$

The 8 anti-patterns which expand instead of contracting the number of crossings are thus:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cb^ncB CB^nCb

cBc^nB CbC^nb

cBC^nB Cbc^nb

cB^ncB Cb^nCb

An FSA can be designed which fails on the 8 expanding ("bad") patterns, but works on the 8 contracting ("good") patterns.

For example, the pattern cb^ncB is not usable as a pattern because it increases the number of crossings by two. As can be seen by its derivation below, it and the other 7 non-usable combinatorially constrained patterns are equal to one of the usable patterns concatenated with a two additional crossings.

$cb^ncB \rightarrow$

$Bc^0cb^ncB \rightarrow$

$Bc^1bcb^{n-1}cB \rightarrow$

$\vdots$

$Bc^nbc^2B \rightarrow$

$[Bc^nbc][cB]$ (but Bc^nbc is a pattern reducing to cb^n)

Alternatively, deriving on the right:

$cb^ncB \rightarrow$

$cb^n[CBbc]cB \rightarrow$ (adding the identity in brackets)

$[cb^nCB]bccB \rightarrow$ (noticing the pattern in brackets)

$[Bc^nbc]cB \rightarrow$ (but Bc^nbc is a pattern reducing to cb^n)

Thus, deriving on either side just makes it longer, not shorter. In particular, like the other 7

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"bad" patterns, it makes the braid longer by expanding 2 of the crossings to the 4-crossing pattern from which they derive. This is exactly the reverse of the contracting nature of the 50 percent of patterns which eliminate crossings.

□

Notational Preparation for the Augmentation Lemma

To prepare the notation for Lemma 1.2.20, the appearance of a repeated (starred) set of characters in square brackets on both the right and left of the equation will serve as a static variable. This denotes that the pattern recognized by that regular expression in brackets is replaced by itself (just copied) on the right side of the expression. This is done in its current place, while the other patterns around it are substituted as ususal.

Preliminary Fence Augmentation Lemma

Lemma 1.2.17. *The elimination patterns of Lemma 1.2.8 may be augmented with the following additional ignorable fences, without impairing their ability to eliminate crossings. While these are copied over, the same substitutions and the same elimination of two crossings occur as in Lemma 1.2.8.*

$$\begin{aligned}
c[aAcE]^{*}C &\rightarrow [aAeE]^{*}\varepsilon \\
c[aAeE]^{*}b^{n}[eE]^{*}C[dDeE]^{*}B &\rightarrow [aAeE]^{*}B[eE]^{*}c^{n}[dDeE]^{*} \\
c[aAeE]^{*}b[eE]^{*}c^{n}[dDeE]^{*}B &\rightarrow [aAeE]^{*}b^{n}[eE]^{*}c[dDeE]^{*} \\
c[aAeE]^{*}b[eE]^{*}C^{n}[dDeE]^{*}B &\rightarrow [aAeE]^{*}B^{n}[eE]^{*}c[dDeE]^{*} \\
c[aAeE]^{*}B^{n}[eE]^{*}C[dDeE]^{*}B &\rightarrow [aAeE]^{*}B[eE]^{*}C^{n}[dDeE]^{*}
\end{aligned}$$

Proof. Each of these is just an expression of the fact that the commutative braid relations of the form $xy = yx$ permit incursions of strands two or more to the left or right of the strand participating in a crossing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

□

Preliminary Abbreviated Fence Augmentation Lemma

Further notation condensation occurs using a curly arrow to denote that during substitution, the sets in square brackets on the left are ignored, that is, not replaced or deleted, but rather left alone, while eliminating the end crossings as usual and substituting the right side for remaining left side **in their place** in the braid. The purpose of introducing this new notation for substituting in place, while ignoring the ignorable fences, is to make it easier to see how to implement algorithms based on the notation more efficiently. The notation symbolizes an easy way to save $O(N)$ copies, which are each $O(n)$ long, in the FSA that will be executed $O(N^2)$ times. In the worst case n approaches N, and $O(n) = O(N)$. Thus, a naive quartic $O(N^4)$ algorithm might perform in cubic time $O(N^3)$ by engineering such substitution in-place instead of dynamically allocating copies.

Lemma 1.2.18. *The elimination patterns of Lemma 1.2.17 may be further augmented with the following additional ignorable fences, without impairing their ability to eliminate crossings.*

$$
\begin{array}{lcl}
c[aAeE]^{*}C & \rightsquigarrow & \varepsilon \\
c[aAeE]^{*}b^{n}[eE]^{*}C[dDeE]^{*}B & \rightsquigarrow & Bc^{n} \\
c[aAeE]^{*}b[eE]^{*}c^{n}[dDeE]^{*}B & \rightsquigarrow & b^{n}c \\
c[aAeE]^{*}b[eE]^{*}C^{n}[dDeE]^{*}B & \rightsquigarrow & B^{n}c \\
c[aAeE]^{*}B^{n}[eE]^{*}C[dDeE]^{*}B & \rightsquigarrow & BC^{n}
\end{array}
$$

Proof. Each of these expresses exactly the same substitution of Lemma 1.2.17, however, after the commutative moves and the substitution moves, the crossings that were commuted are moved back. This has the same effect as not having commuted them to begin with, but rather leaving them in their original position, while doing the substitutions and eliminations. This is always valid because all conjugates of relators are relators.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

□

Preliminary Ignorable Fencing Lemma

Lemma 1.2.19. *The elimination patterns of Lemma 1.2.8 may be augmented with the following additional ignorable fences, without impairing their ability to eliminate crossings.*

$$\begin{array}{lcl}
c[aAeE]^*(b[eE]^*)^n C[dDeE]^* B & \rightsquigarrow & Bc^n \\
c[aAeE]^* b[eE]^*(c[eE]^*)^n[dDeE]^* B & \rightsquigarrow & b^n c \\
c[aAeE]^* b[eE]^*(C[eE]^*)^n[dDeE]^* B & \rightsquigarrow & B^n c \\
c[aAeE]^*(B[eE]^*)^n C[dDeE]^* B & \rightsquigarrow & BC^n \\
c[aAeE]^* C & \rightsquigarrow & \varepsilon
\end{array}$$

Proof. Each of these eliminations expresses the fact that the additional commutative braid relations of the form $xy = yx$ are permitted, because in these additional cases the strands participating in the incursion are still a distance of two or more away from the strands being replaced by the substitution or the strands being eliminated in each pattern.

□

Fence Augmentation Lemma

Lemma 1.2.20. *The elimination patterns of Lemma 1.2.8 may be augmented with the following additional ignorable fences. No additional augmentations may be added, without forbidding valid eliminations.*

$$\begin{array}{lcl}
c[aAeE]^*(b[eE]^*)^n C[dDeE]^* B & \rightsquigarrow & Bc^n \\
c[aAeE]^* b[eE]^*(c[eE]^*)^n[dDeE]^* B & \rightsquigarrow & b^n c \\
c[aAeE]^* b[eE]^*(C[eE]^*)^n[dDeE]^* B & \rightsquigarrow & B^n c \\
c[aAeE]^*(B[eE]^*)^n C[dDeE]^* B & \rightsquigarrow & BC^n \\
c[aAeE]^* C & \rightsquigarrow & \varepsilon
\end{array}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof. These are simply the maximal incursions that a, b, d, and e crossings can make into the patterns according to the $ab = ba$ relators, by exhaustion of the 5 possible fences which may potentially cause an incursion into the 5 possible parts of a pattern: (1) between the first and second element, (2) in the middle of the second element when it happens to repeat, (3) between the second and third elements, (4) in the middle of the third element when it happens to repeat which only happens when the second element does not repeat, and (5) between the third and fourth element. For example, in the augmented elimination $c[aAeE]^*(b[eE]^*)^n C[dDeE]^* B \rightsquigarrow Bc^n$ it is not possible for the D strand (crossings d and D) to cause an incursion in the first or second part, such as $c[aAdDeE]^*(b[eE]^*)^n C[dDeE]^* B$ or $c[aAeE]^*(b[dDeE]^*)^n C[dDeE]^* B$ because the D strand has distance of only 1 from the C strand in the first and third elements, and can not commute past it. As a result, the FSA cannot recognize the as a pattern for the FST to replace, until the D strand fence is eliminated, at which time this pattern becomes uncovered. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The final FSA looks like Figure 1.20, combined with mirror image states for C which is the inverse of c, and rotated states for each of those substituting the D strand for the B strand.

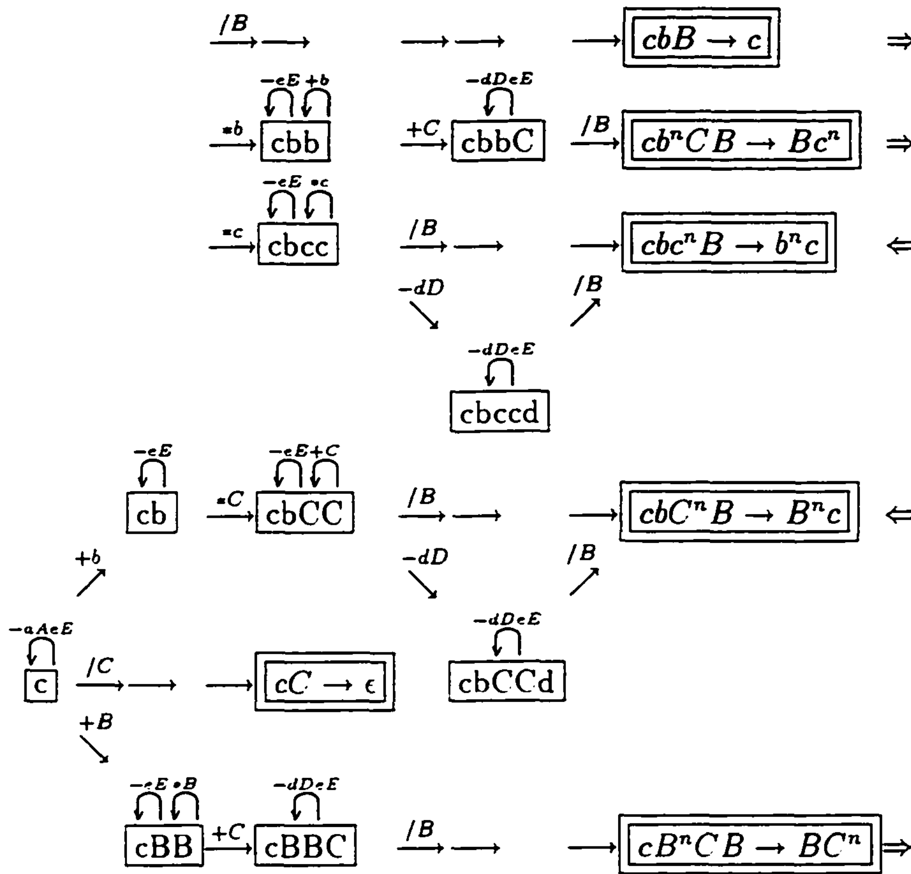


Figure 1.20: Finite State Automaton To Recognize the Patterns

LEGEND

-	ignore
/	delete
+	transmute (null or capitalize)
*	transmute (null or swap $B \leftrightarrow C$
⇒	requires the transducer to run forwards
⇐	requires the transducer to run backwards

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

FSA Equivalency Lemma

Lemma 1.2.21. *The elimination patterns of Lemma 1.2.8, augmented as in Lemma 1.2.20. are equivalent to FSA in Figure 1.20.*

Proof. Standard conversion of regular expressions to FSA. □

Look-Ahead Requirement Lemma

Lemma 1.2.22. *For transducing (replacing), the recognition FSA cannot be used because it would sometimes require a look ahead of one crossing (at the end of the x^n).*

Proof. Half the patterns have no additional intermediate nodes after the repeated (x^n) portion of the pattern. This requires searching up to the full length of the braid. because at each repeated crossing x. starting with the second one (first repetition). the transducer does not know whether to modify (modify and invert) or to keep (keep and invert) the crossing, until after the next repetition of the crossing is seen. □

Transducer Effectiveness Lemma

Lemma 1.2.23. *After running the FSA to know which pattern was recognized (from the final state of the FSA). the following FSTs perform the required eliminations.*

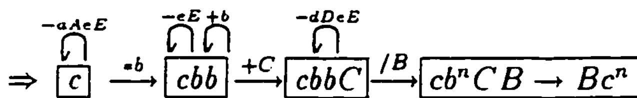


Figure 1.21: Transducer for Pattern 1

Proof. The two patterns which are marked ⇐ in Figure 1.20, the FST can work without a look ahead. if and only if run backwards. The two patterns which are marked ⇒, the FST can work without a look ahead, if run forwards. To see this, notice that in the patterns to be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

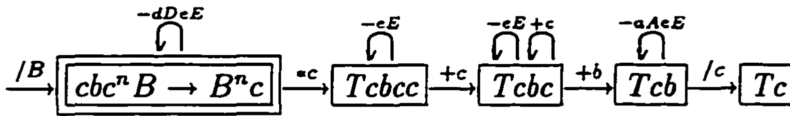


Figure 1.22: Transducer for Pattern 2

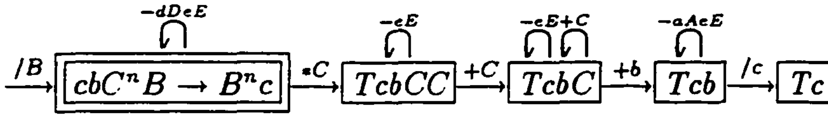


Figure 1.23: Transducer for Pattern 3

run backwards. the repeated element of the pattern is the third element. The replacement for the first $(n-1)$ of the N repeated patterns is a substitution where C becomes the next strand, for example, B. The nth or last crossing on the C strand remains on the C strand. possibly changing case (sign). In these two patterns, if the FST were run forwards. the FST would not be able to ascertain when it was on the nth repeating crossing until after it had left the third element and gone into the fourth. Therefore. a FST going in the wrong direction would suffer the efficiency loss of either keeping memory for looking ahead or back tracking to repair an incorrectly substituted element. To prevent these undesirable characteristics, the FST is run forward on two of the patterns, and backwards on the other two. The 2 forward transducers match the FSA exactly. The 2 backwards transducers match the FSA except that their inputs and outputs are reversed, and that forces the repeated node to be split into two nodes for purposes of avoiding the look ahead, precisely as shown in these FST state transition diagrams. □

Replacement Decoupling Lemma

Lemma 1.2.24. *If the FSA recognizes it, and the FST transducer replaces it, no additional fences will be introduced because the entire expression is bounded by consecutive-strand crossings which breech no additional interference between them.*

Proof. There can be no extra incursions, if the commutative relators were executed to move the augmented crossings to one side of the portion of the braid being replaced, before re-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

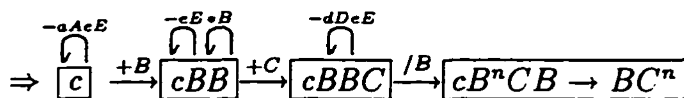


Figure 1.24: Transducer for Pattern 4

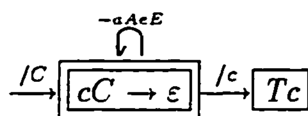


Figure 1.25: Transducer for Pattern 5

placement takes place. But whether they are physically moved or just ignored has no effect on their movability, before, during, or after the move.

Each object of the replacement is bounded in the recognition string by either

- both Bs and Cs or
- both Ds and Cs.

Therefore,

- Before the move they begin as movable, as shown in Lemma 1.2.20.
- During the move each replacement involves only strand C and one of its neighbors Y, where Y is either strand D or strand B. This move is bracketed entirely by a C and a Y strand because a 4-crossing pattern is being recognized. At each point during the move each C crossing is bracketed by Cs and each Y crossing is bracketed by Ys. Therefore, no incursions may cross that bracketing (boundary).
- After the move nothing has changed that could prevent anything from moving out that was permitted to be in, in the first place, because only $C \leftrightarrow Y$ substitutions have been made.

The 4-crossing elimination thus uses this bracketing boundary around the pattern through which no effects pass which could stop the recognition and the replacement.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore, no extra incursions of crossings can occur. □

1.3 FST Correctness Theorem

Theorem 1.3.1. *The FSA (search) and FST (transducer=replace) implement the given eliminations. Each execution of the FSA either:*

- *returns FAILURE if no reductions are possible starting at that crossing and proceeding in that direction or*
- *stops in a state that identifies one of the patterns. In this case, execution of the FST for that pattern, in the correct direction according to the arrow signs in Figure 1.20 eliminates one pair of crossings from one of the four patterns and, possibly, additional pairs, if any xX pairs happen to be found.*

Proof. The patterns in Lemma 1.2.8 are cyclically reduced and complete in the following sense. They are:

- shiftable by Lemma 1.2.9,
- reversible by Lemma 1.2.10,
- invertible (negatable) by Lemma 1.2.13,
- rotatable by Lemma 1.2.14,
- exponent invertible by Lemma 1.2.15,
- effective (requiring no other patterns) by Lemma 1.2.16, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- fully considerate of all possible incursions of irrelevant but non-interfering crossings in the middle of the patterns by Lemma 1.2.20.

.

The FSA described in Lemma 1.2.21 recognizes any pattern beginning on the current crossing in any identity braid, if that pattern exists at that crossing. The FST described in Lemmas 1.2.22 and 1.2.23 correctly eliminates the required two crossings and makes the correct substitution according to the patterns in Lemma 1.2.8 as those patterns were augmented in Lemma 1.2.20.

The process of performing this recognition and transduction do not take any actions which might require additional fences to be generated according to Lemma 1.2.24. Therefore, the combination of

- the complete FSA,
- the correct algorithm which uses the final state of the FSA to start the selected FST at the pattern's start or finishing crossing, as recognized by the FSA, according to the directional arrow in Figure 1.20, and
- the correct FST

will leave identity braids identity braids, while conducting all possible eliminations at the crossing the FSA is started at.

□

FST Timing Lemma

An interesting question is why the horizontal linked lists do not shorten the search to a constant time. It would seem that the FSA could recognize $cbCB \rightarrow Bc$ in constant time,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

given the vertical (fence) linked lists per crossing and the horizontal (strand) linked lists per strand to do all side strand checking and forward crossing checking in constant time. Although true for this example, this argument is deceptively false in general. and the form of the patterns in Lemma 1.2.8 brings out the problem. The actual pattern in Lemma 1.2.8 is $cbC^nB \rightarrow B^nc$ with an unlimited number of repetitions. These repetitions can span the braid. making them take a linear amount of time, that is, $O(N)$. instead of a constant amount of time.

Lemma 1.3.2. *Each search (FSA execution) takes time linear in the length of the braid. that is, $O(N)$. Each transduction takes time linear in the length of the braid and eliminates 2 crossings. and adds a constant number of additional searches to do.*

Proof. The FSA advances one bead per move, and so does the transducer. Each transduction implements a pattern which eliminates 2 crossings by Lemma 1.2.8. Each reduction schedules up to 80 FSA searches for potentially uncovered eliminations by Lemma 2.1.4). □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

The Braid Elimination Data Structures

2.1 Skeins and Algorithm A

A data structure which supports braid elimination efficiently is a skein containing the following information, for each crossing in a braid:

- the crossing number
- the bead code, which is one of [$abcdeABCDE$]
- a forward and backwards entry in the vertical linked list
- a forward and backwards entry in the horizontal linked list
- a forward and backwards entry in the schedule linked list

This data structure has a size proportional to the number of crossings and to the number of strands, that is, $O(N + K)$. While a linear amount of space is sometimes feasible, this is not as space efficient as algorithms which run in $\log n$ space.

After eliminating 2 crossings using one of the 4 patterns, other patterns might appear which can also cause crossings to be eliminated. These new patterns which were not there before,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

but are there after the elimination are called *uncovered eliminations.*

2.1.1 Horizontal and Vertical Linkage Lemmas

Horizontal Linkage Lemma

Lemma 2.1.1. *The horizontal linked lists on each strand eliminate a combinatorial explosion of consideration of the ab=ba rules by automatically passing over them to the next applicable crossing (bead) on a strand.*

Proof. The linked lists save a potential pass through the braid each time they are used, because they instantaneously point to the next relevant crossing for this strand and for this pattern. Looking across a linked list for a strand. the next bead on the linked list is the next fence on that strand. This replaces an exponential number of permutations with a linear structure. This removes things from consideration that do not affect the pattern without adding to the execution time, by removing up to a pass per FSA run and a pass per FST run. The patterns are thus aggregated and protected from other patterns interfering with them. due to Lemmas 1.2.20 and 1.2.24.

It takes a linear amount time and space to set up these linked lists. Thus, this algorithm fails to run in log space. but the $O(N)$ setup time is negligible because it is dominated by the $O(N^2)$ search time. □

2.1.2 Vertical Linkage Lemma

Lemma 2.1.2. *The vertical linked lists implement the required fences.*

Proof. Using a linked list per set of 5 fences permits constant time insertion and deletion of these sets of 5 fences, which will appear as beads at differing positions along the horizontal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

linked lists. Thus these linked lists prevent searches, up to the full length of the braid, each time the corresponding fence on the next braid must be located for insertion or deletion purposes. Only 5 fences are sufficient because the cross strand effects are limited by Lemma 1.2.20.

□

2.1.3 Eighty Uncovered Eliminations Lemma

Lemma 2.1.3. *There are at most 80 uncovered eliminations induced by each elimination. That is, the FSA/FST must be scheduled to try or re-try at most 80 additional beads upon the elimination of a pair of crossings.*

Proof. The key is that by (q) Lemma 1.2.10, (b) the distance constraints on the $xyx = yxy$ braid relators, and (c) the fact that no elimination starts with a repeated crossing such as c^n, no elimination can be uncovered if it starts more than a distance of 2 away from the two crossings that were eliminated.

Initially uncovered eliminations can be searched for as follows. Eliminating a crossing involves eliminated all 5 of its fence beads from the linked lists. The patterns uncovered by an elimination are within 2 crossings on either side of the elimination since uncovered patterns cannot start in the middle of the x^n portion of the pattern on the left. Therefore, there are at least four places an uncovered pattern can begin on the left (forwards) and four on the right (backwards). These 8 places are multiplied by the 5 fencing strands on which an uncovered pattern may appear. Thus there are at least 40 potential uncovered eliminations to check for at each elimination. For example, after transducing, the pattern $cb^nCB \rightarrow Bc^n$, applied to the braid $wcbbbbbbbCBv$ which eliminates to $wBcccccccv$, the following spots at a minimum must be checked:

- the first c backwards

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- B backwards
- w forwards
- B forwards
- the first c forwards (only if $n = 1$)
- the last c forwards
- the last c backwards (only if $n = 1$)
- v backwards

Note that w backwards and v forwards might contain eliminations, but they would not be uncovered by the current execution of the FST, therefore, they would have already been scheduled by the primary thrust of the algorithm rather than scheduled as part of the uncovering process.

In addition to these uncovered eliminations, the 5 eliminated fences could have been blocking unrelated eliminations on each of their respective strands. Therefore, it is necessary to treat the following spots as potentially uncovered:

- for each of the two sets of 5 eliminated fences (10 beads total),
- for each of the two directions,
- for each of the next two beads in that direction starting at the eliminated fence bead,
- execute the FSA/FST against that bead.

This requires $10 \times 2 \times 4 = 80$ examinations. Since these 80 examinations include the 40 above, the total is 80, not 120. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus it is possible to algorithmically repeat the FST against the braid at the correct spots located by the FSA. and other spots uncovered by the FST by checking (re-running the FSA) at 80 beads.

2.1.4 Algorithm A Effectiveness Lemma

Lemma 2.1.4. *The following algorithm (Algorithm A) advancingly eliminates crossings from an identity braid:*

- *apply the FSA repeatedly across each crossing of the braid.*
- *when the FSA finds a pattern. apply the FST, then schedule the 80 potential uncovered patterns for FSA examination. unless they are currently scheduled.*
- *if any xX pairs are discovered along any of the linked lists by FSA. eliminate them immediately. either backtracking or restarting the FSA.*

Note: A scheduling linked list is kept to avoid a bead from being checked more than once after it was scheduled. if it had not already been checked. Upon selecting a bead to check. it would be removed from the scheduling table. Beads which have already been eliminated from the braid are removed from the scheduling table without checking them with the FSA.

Proof. The algorithm peels away the relators by repeatedly recognizing and eliminating according to the patterns implemented in the FSA in Lemma 1.2.21 and the FSTs in Lemma 1.2.23. That this is done advancingly is implied by Theorem 1.3.1 and Lemma 2.1.3, and by the nature of the patterns as validated in Lemmas 1.2.22 and 1.2.23. □

2.1.5 Algorithm A Quadratic Time Lemma

Lemma 2.1.5. *Algorithm A takes time bounded by $O(N^2)$.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof. The algorithm from Lemma 2.1.4 starts a linear FSA on each bead of each of the N strands once, which is $NO(N)$, which is $O(N^2)$. From Lemma 1.3.2. each reduction starts a constant number (80) additional linear FSAs, and there are at most $N/2$ eliminations. Thus there are most $80N/2O(N)$ moves, which is $O(N^2)$. The sum of two quadratic times is again quadratic. □

2.2 Algorithm F

2.2.1 Algorithm F Cubic Time Theorem

Theorem 2.2.1. *Algorithm F (multiple passes of Algorithm A in Lemma 2.1.4, repeated until it reaches a pass that eliminates no crossings) stops operating within $N/2$ passes and thus operates in $O(N^3)$ time.*

Proof. The skein data structure effectively blocks the proper eliminations with its fences according to Lemma 2.1.1. and efficiently eliminates the need for the algorithm to explicitly handle the commutative relators according to Lemma 2.1.2. The advancing Algorithm A of Lemma 2.1.4 solves many identity braids (and it is conjectured that it solves all identity braids). If there is an identity braid which is not solved by Algorithm A, then because it is an identity braid, it will have at least one elimination recognized by re-running Algorithm A. Since each elimination removes two crossings, all identity braids are eliminated within $N/2$ runs of Algorithm A. As soon as a run of Algorithm A completes (exhausts the schedule), without discovering any eliminations, then Algorithm F may stop. When it stops it reports that the current braid is an identity braid if it has no crossings left, otherwise it reports that it is not the identity braid.

If there is a complex deception causing a single pass of Algorithm A from Lemma 2.1.4 not to recognize all possible uncovered eliminations, then at most $N/2$ passes are required, since at least one elimination is uncovered by each pass, eliminating at least 2 crossings per pass.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore, the repeated algorithm will take $N/2$ times $O(N^2)$ which is $O(N^3)$.

□

2.2.2 Stream Repetition Lemma

Lemma 2.2.2. *To avoid having to use the full strength of a push-down automaton in order to decode streams of consecutive repeated* c *and* C *crossing in the middle of the patterns* $cbC^N \rightarrow B^n c$, $cbc^n B \rightarrow b^n c$, $cb^n CB \rightarrow Bc^n$, $cB^n CB \rightarrow BC^n$, *their rotations, their inverses, and their inverse rotations, it is sufficient to use a rocking back-and-forth state in the FST, because it takes less than* $2N$ *links for each elimination which adds only* $O(N)$ *time.*

Proof. There is a strong relationship among the elimination $cbC^N \rightarrow B^n c$, $cbc^n B \rightarrow b^n c$, $cb^n CB \rightarrow Bc^n$, $cB^n CB \rightarrow BC^n$, their rotations, their inverses, and their inverse rotations. Using the first pair as the example, these two patterns need care when streams of c and C crossings appear, for example, $cccCCCcCcccc$, where it is not known whether the stream eliminates down to c^n or C^n. That is, it is not known by looking at the first element of the stream, whether the exponent will be positive or negative (since $C = c^{-1}$). The stream can also cancel out completely, corresponding to an exponent of 0. But a simple FSA cannot count the total. A PDA could count it, at the expense of an extra storage stack or its equivalent, but that is not needed in this case.

Using the following concept of a rocking back-and-forth state, they can be cancelled on the fly. That is the patterns are combined into $cb[c|C]^n B$, in a special hybrid automaton which notices and eliminates cC and Cc pairs along the way. At the end of a successful recognition pass, the transduction will have deleted all postive (c) or all negative (C) crossings in the stream, and it can find out by executing one additional linear search backwards at the end, what the sign of the exponent is, whether negative, zero, or positive.

Each cancellation along the way must be accompanied by a backwards transversal of the string to see what the previous one was. Of course, a forward or backwards link can take

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$O(N)$ time in the worst case because of skippable fences along the way. However, since this adds less than $2N$ links in the worst case ($O(N)$ forwards and $O(N)$ backwards), it adds at most $O(N)$ links per cancellation. This does not increase the order of magnitude of $O(N^2)$ for Algorithm A.

□

2.3 Completeness and Conjectures

2.3.1 Algorithm F Completeness

Conjecture 2.3.1. Algorithm A will eliminate at least one pair of crossings from any braid which is equivalent to the identity braid.

Proof. Every identity braid is built up by repeatedly applying the braid relators, their inverses, their reverses, their reverse inverses, and conjugates of the preceding. These can be peeled away by one relator at a time. This conjecture would be proven inductively by comparing the results of an existing braid word algorithm to Lemmas 1.2.20 and 1.2.24, and Lemmas 2.1.1 through 2.1.3.

□

2.3.2 Algorithm F Completeness Corollary

Theorem 2.3.1. *Conjecture 2.3.1 implies Algorithm F solves the braid word problem in $O(N^3)$ time.*

Proof. Algorithm A eliminates at least one pair of crossings from non-identity braids equivalent to an identity braid, by Conjecture 2.3.1. Each pass of Algorithm F is just Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A in $O(N^2)$ time. There are N crossings, so $N/2$ passes are sufficient to to eliminate all crossings. This will not take more than $O(N^3)$ time by Theorem 2.2.1. □

2.3.3 $O(N^2)$ Conjecture

Conjecture 2.3.2. Algorithm A solves the braid word problem in $O(N^2)$ time.

2.3.4 Future Research

Several paths stand out for future research:

- Continue research to determine if Algorithm F is in $O(N^2)$.
- Continue analytical research to prove Conjecture 2.3.1, perhaps by an induction over Artin's braid word algorithm.
- Continue empirical research to use these data structures to speed up the braid conjugacy problem, discrete geometry applications on braids, and workings on Class B Braids (which add a strand number 0 and an extra relator).
- Attempt to combine these elimination techniques with other constrasting techniques such as:
  - approaching braids via reductions to normal forms instead of eliminations, such as [25], to see how fast those methods would run with these data structures, and on which additional classes of braids those algorithms would become effective
  - combining these techniques with the topological techniques of [26] to apply these data structures and eliminations to knots and links, to speed up those algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

The Braid Computational Experiments

3.1 Introduction

The Braid Lab Software was designed to work with certain groups. especially braid groups. The following definitions. capabilities. and functions motivate and reflect how the software works. and often follow the notation. semantics of modern group theory. Differences from modern group theory include: limitation to the volume of virtual memory and CPU time available. an operational (dynamic) interpretation of functions. non-static variables which can take on progressively different values at different times (and the concept of time itself). and emphasis on data structures which reduce the time (defined as the number of primitive operations) it takes to compute certain functions.

In a computer. a portion of a group can be modelled as a **Cayley graph**.

- The Cayley graph for a group G is a graph with a distinguished vertex (or node) for the identity word itself, at least one vertex for each element of the group, and at least one edge at each vertex for each generator.
- In a Cayley Graph, the action of the product, $z = x \cdot y$ is represented by two nodes labelled x and z. respectively, connected by an edge labelled y. Thus elements are both

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the nouns (objects) and the verbs (processes) in this action.

- A word in a Cayley Graph is a path of successive edges. The identity in a Cayley Graph is equal to any path that leads back to its starting node.
- Each relator $r \in \overline{R}$ appears as a closed path in the Cayley graph.
- The **distance** between two words in the group is the minimum number of edges separating them in the Cayley graph. Points along the edge can participate in the distance function by creating an arbitrary mapping of the unit interval to the edges.

In the Braid Lab Software relators are represented as words, and they are combined into an array of relators, and further combined into a structure comprising an alphabet and an array of relators, with appropriate counters for the number of elements and number of relators. The non-braid group experiments use this structure for the input data. The function `tsgroup` in the Braid Lab Software demonstrates how to set up presented groups, perform group operations, and attempt to test for the word problem in groups by continually building and collapsing the Cayley graph of the group until one of the following events occurs.

- time or storage run out
- the Cayley graph of the word collapses to the identity
- the maximum number of Cayley graph levels requested is reached.

3.1.1 Data Structures

Many data structures can represent sets of data objects with sets of methods operating them. The skein used to represent groups and braids in this paper builds on these structures. The Braid Lab Software includes and uses code for common ADTs, such as the following containers:

- array and packed arrays of objects,
- Huffman trees,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- random number generation,
- associative lists implemented via balanced trees, skip list, or hash codes,
- finite state automata and finite state transducers,
- linked and double-linked lists,
- circular linked lists where each traversal keeps track of where it started (and therefore where it finishes),
- schedules implemented as multiply-double-linked lists through the crossings, with the addition of a binary existence flag corresponding to each bead in the list, and with the **previous** and **next** pointers held in the bead corresponding to each crossing,

In addition, the Braid Lab Software provides some unusual ADTs, such as:

- beads that hold a crossing and associated linked list pointers, fences, codes, flags, the finite state automaton state, and so forth, so that, in addition stepping their way through the following the strand beads and fences, they can also use random access to determine whether a given bead number is alive without stepping through the other beads (This schedule data structure is used in the Braid Lab Software in group functions `collapse` and `mega-collapse`, and in braid function `skein_control` which is used by all of the `test_braid` functions. The Schedule alive bits are usually stored outside the linked list, contiguously and packed, for smallest working set size.),
- group presentations, some of whose methods might not terminate, because, for example, generating a random Tietze Transformation or finding conjugates cannot be programmed without using a solution to the group word problem.
- Cayley graphs for a word in a group with two methods, `collapse` and `mega-collapse`, of which only the `mega-collapse` is visible, used to check the group, and which works by building the Cayley graph out to any particular finite level and repeatedly collapsing where possible will ever be able to reduce an identity word to the identity, useful for experiments on small groups whose Cayley graphs fit in virtual memory,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- braid databases represented as Skeins which contain a group as a braid with a simplified Cayley graph and the required linkages and schedules for efficiency and multi-level finite state automata.
- circularly linked skeins, with `linking` is set to `True`, where the double linked lists implementing the strands act as though they are a circular list instead of a linear list, that is, the ends act as though they are connected, useful for experimenting with concepts related to conjugacy, because a circular list representing a combinatorial braid mimics the geometry of conjugate rotation on that braid,
- and combinatorial braids implemented as a concatenation of permutations.

3.1.2 Finite State Automata

A **Finite State Automaton** (FSA) is a six-tuple $< A, S, s, \Omega, \delta, \varepsilon >$ where

- A is a set of letters, the alphabet;
- S is a set of states including the state FAIL;
- s is a distinguished state, the START state;
- Ω is a subset of S, the final states;
- ε is a letter not in A;
- $\delta : S \times (A \cup \varepsilon) \rightarrow 2^S$ is the transition function

The transition function δ of a **deterministic FSA**:

- maps all tuples with ε as the second element onto the singleton set containing only the first element, and
- maps all tuples onto singleton sets.

A **nondeterministic FSA** is any FSA that is not deterministic.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The **language accepted by a FSA** is the set of words that cause the machine to enter a final state other than FAIL:

- reading the words into the machine one character at a time.
- following the transition function by moving from state to state,
- reading one character per non-ε transition.

Extensions to FSAs include the following:

- In practice one almost always uses Finite State Transducers, rather than Finite State Automata. A Finite State Transducer is a FSA along with an action table which permits the FSA to do output.
- If a Transducer is permitted to access additional memory, other than the current state and current character, then it is possible to leave the limited world of FSAs, and enter the more complex (time consuming) world of push-down automata or Turing machines, losing some of the guarantees of termination on finite inputs. This memory access occurs in Section 3.3.6, at the moment the braid FSAs transition into a `reduce` state. The algorithms and data structures are then examined to prevent this complexification.
- A much more complex machine, that is almost always more complex than an FSA, is an FSA with conditions on the transitions. For example, if the transition matrix is replaced by a procedure that decides where to transition to, by accessing memory other than the current state and the current character. These conditions may be used to block (terminate with failure) faster, to reduce (terminate with success) faster, to go to different states, or to condense a character. The simplest of these more complex machines, a push-down automaton, is disciplined to access a single stack of memory. More complex machines would permit arbitrary conditions on the transitions, and arbitrary increases in time complexity.

Lemma 3.1.1. *FSAs recognize the Regular Languages which are built recursively by noting that they are closed under concatenation, alternation, and Kleene starring.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

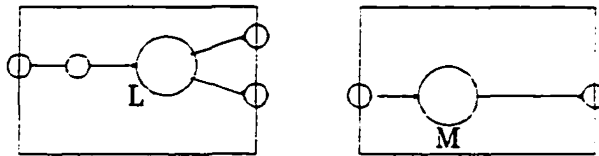
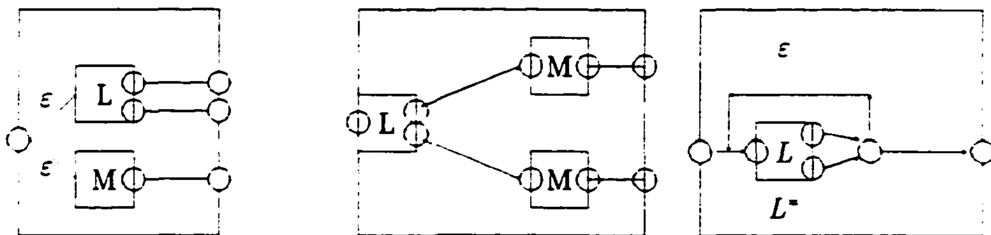


Figure 3.1: FSA L (5 states) and M (3 states)

For example, FSAs L and M from Figure 3.1 are used below in Figure 3.2 to construct new FSAs:

- alternated (recognizing the language of strings from either L or M
- concatenated (strings from L concatenated with strings from M)
- starred (recognizing the language comprising zero or more copies L)



Alternated $L \cup M$ Concatenated LM Starred L^*

Figure 3.2: Methods Operating on FSAs

Lemma 3.1.2. *For every nondeterministic FSA recognizing a language, there is a deterministic FSA recognizing the same language.*

Proof. From the nondeterministic machine N, the deterministic machine D is constructed. The states of D are subsets of the states of N. Moves in N involving ε transitions, and transitions with

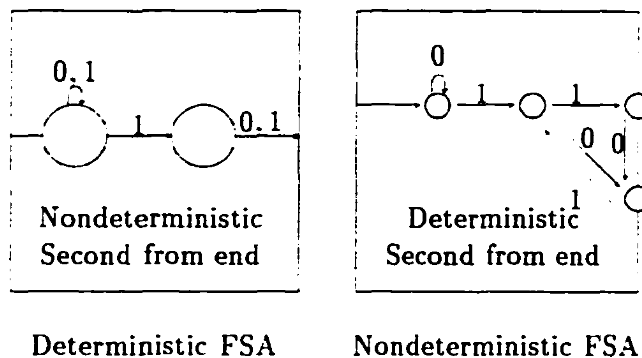
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

multiple choices can be split into the states of D. Then redundant states and transitions can be eliminated. □

Corollary 3.1.3. *Continuing from the above Lemma, the number of states in the deterministic FSA may be exponentially greater than the number of states in the nondeterministic FSA in the worst case.*

For example, consider the language ANTEPENULTIMATE consisting of the all words with a 1 as the nth letter from the end of the word. One can implement a FSA to recognize the ANTEPENULTIMATE language in the two ways shown in Figure 3.3

- On the left is shown a nondeterministic FSA with n states.
- On the right is a deterministic FSA with 2^n states.



Deterministic FSA Nondeterministic FSA

Figure 3.3: Antepenultimate FSA

An FSA is an algorithmic trade-off between space and time. At the expense of a very large number of states, an FSA may be the fastest way to repeatedly look things up in a list (see Section 3.3.4).

Lemma 3.1.4. *Each FSA has a pumping number. If it recognizes a word longer than the pumping number, it recognizes an infinite number of words.*

The pumping lemma is a simplification of the pumping lemma for context-free languages. It follows from the fact that a deterministic machine has a finite number of states, each of which condenses

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an input character. If it recognizes a word of length greater than the number of states, there must have been a loop.

FSAs are important in this discussion because:

- They make a trade off between: (1) an exponential amount of time with a small number of states and (2) a linear amount of time with an exponential number of states.
- They terminate in a finite amount of time, when fed a finite input stream. Additionally, deterministic machines terminate in a linear amount of time (with respect to the length of the inputs). Processes modelled by FSAs are therefore finite and "fast."
- They search for multiple patterns in parallel on a sequential machine.

3.1.3 Reduced Words

A **reduced word** in a group G is a word with no consecutive occurrences of a letter and its inverse.

A **cyclically reduced word** in a group G is a nontrivial reduced word in G whose first element is not the inverse of its last element.

Braid reductions can be dealt with by exploiting the fact that the braid can be reduced by injecting identity braids between two crossings then applying relators. However, the resulting minimal braid sometimes reduces differently based on which identity braid is injected and where. An important exception to this is that a braid equivalent to the identity will always reduce down to the identity.

3.1.4 The Patterns in the Braid Lab Software

The Braid Lab Software considers that the braid word problem is solved when the braid can be combed (reduced) so that each strand goes to its corresponding peg on the other side without crossing another strand. If two braids can be combed to have the same crossings they are isotopic. The isotopies among geometrically represented braids are exactly analogous to the word transformations induced by applying the braid relators and reducing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each braid fixes a permutation in the strands from one end to the other, although it may go through many intermediate permutations as it is read from right to left.

Note: only 3 of the first 4 patterns are truly necessary. The historical evolution of this research shows that no braid required all four patterns, which shows a large amount of empirical interdependence among the four patterns. Any of the 4 is derivable from the other 3, and any 3 will make an advancing algorithm (always deleting crossings, never adding crossings). Nevertheless, all 4 are presented here because it is more compact to make the FSA in Algorithm F symmetric. In addition, that FSA will recognize an elimination in fewer moves when it does not need to wait for combinations of the 3 patterns to subsume the fourth.

3.2 The Data Structure Experiments

This Section describes the experiments, and the next Section describes the resulting levels of deception, counterexamples, observations and some applications of the results. The research described in this paper experimentally determined data structures to assist in giving greater performance to group algorithms, concentrating on braid groups. Performance includes bringing the algorithms closer to solving the main braid group problems, and making the algorithms run faster. To carry out the experiments, various group algorithms were created which are described here.

Some of the data structures and algorithms generate random groups, random braids, and, most importantly, random identity braids. Without software to do this, most of the experiments are impossible. Therefore, these will be described first.

The random groups, braids, and identities generated were not used as part of a randomizing algorithm, nor were they used in a simulated annealing to re-run an algorithm randomly. Rather, they are used to generate test cases.

These test cases have two purposes.

1. The ordinary test case purpose, is to add credence to the efficacy of the algorithm by running it on random cases. For this purpose, it is important that the random generator is capable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of generating interesting cases that will break an incomplete algorithm.

2. An unusual purpose, is to discover counterexamples. This research intends to discover experimentally the data structures required to get the desired performance from computing a solution to a braid problem. To do this, a robust and relatively complete random object generator is essential to success. Any weakness in the randomness of the identity braids generated, for example, would invalidate any automatic verification of braid algorithms examined.

3.2.1 Generating Random Objects

Random Elements, Words, Relators, and Groups

Random elements are selected by a uniform random number generator. The code included in the Braid Lab Software happens to be the Fibonacci sequence generator described in [14]. It can be replaced by other uniform integer generators.

Random Tietze Transformations for Groups

The following recipe for using a group U with unsolvable word problem, as a cryptographic system is flawed.

1. Let a key be a random sequence of Tietze Transformations for U.
2. Encrypt a message (translated into generators of U) by applying the key.
3. It will be difficult or impossible to decode the message without the key.

The problem is right at the beginning in step 1. Random Tietze Transformations are problematical for groups with unsolvable word problems. In order to choose a relator to add, that relator r must be in the transitive closure of the other relators, that is, $r \in \overline{R} =$ the set of conjugates of $R^* - R$. To determine if this is true requires determining if $r = s$ for some $s \in R$, in order to enforce

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the minus part, which involves solving the word problem for the unsolvable group. The same characteristic that makes it difficult or impossible to solve (and, therefore, useful for cryptography) makes it difficult or impossible to set up keys for different users. Thus, unsolvable groups may be inapplicable for this particular cryptographic application.

Random Identity Braids

Random braids (not identity braids) are easy to spin up. To get a random braid on K strands, roll numbers in the range $1..K-1$. Then roll a two-sided die to get a sign, either - or +. If desired, get rid of consecutive xX and Xx sequences. Keep doing this until the braid is long enough. Random identity **braids** that provide useful counterexamples, on the other hand, are not that easy.

The experimental determination of braid data structures to solve word and conjugate problems requires random identity braids. This random identity capability must be of such quality as to point out the deficiencies in the data structures. As more efficient algorithms are evolved in other research, they can be compared to the data structures in this paper. In addition, those new algorithms can be tested against the random braid generator, taking this research one step further, to detect any weaknesses in those proposed algorithms, for example, at preprint time.

It is not sufficient to generate identity braids by generating random braids and concatenating them onto their inverses. These would indeed be identity braids, but they would be remarkably easy to solve, and therefore are not random enough for this research. Similarly, many concatenations of small identity braids make an identity braid, but again useless for this research, which aims to discover flaws in braid data structures and algorithms.

Looking at the definition of the braid group, there a number of relators to apply as substitutions over symbols in the word being examined.

Therefore, it might be possible to consider applying randomly selected relators. Unfortunately, this would be similar to random walk generation, which, with some probability, returns to its starting point. Randomly selected relators might apply the reverse of the transformations which were earlier applied, returning the braid to its original form. Even worse, most of the relators have to do with the fact that the braid group is almost commutative (on all but neighboring strands). That means

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that as the number of strands increases, the probability increases that a random relator is just an innocent migration or commute of two non-touching strands, which may not contribute much to the difficulty of solution of the word or conjugate problem, and, therefore, might not provide a powerful counterexample. However, it is true (Lemma 1.2.4) that every braid isotopic to the Identity Braid is generated by substituting a sequence of relators starting from the Identity Braid.

Using only relators of the form $aba = bab$ might make a braid more complex. However, repeated use of only these relators may not discover the most complicated twists needed as counterexamples.

Maximally twisting a braid may make a braid more complicated by adding an identity braid in the middle of it at random, then using the almost commutative properties to flee the participants in the added identity braid the maximal distance from each other, along the braid.

The final, combined strategy used to develop the counterexamples in this paper is to generate random braids of length N in the following manner.

1. Generate a random braid of length $\frac{N}{4}$ by spinning random numbers.
2. Juxtapose it with its inverse, forming a braid of length $\frac{N}{2}$.
3. Repeat the following step, inserting an identity pair $\frac{N}{4}$ times.
   (a) Select a random generator, x, which may be positive or negative.
   (b) Select a random position in the braid.
   (c) Insert xX at the random position in the braid.
   (d) Using the almost commutative relators, move x as far left as possible in the braid. Move X as far right as possible. It is interesting that profiling the code shows that this step takes the longest, and for most braids, the time to spin the random identity braid takes longer than to solve that identity braid.
   (e) Note in the data structure that the braid is now 2 elements longer.

The efficacy of this random identity braid procedure is to be judged by comparing the data structure evolved by solving its random braids with other currently and eventually published braid algorithms. The efficiency is enhanced by contiguous heap addressing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2 Data Structures for Group Algorithms

Group Presentation Data structure

Group presentations are implemented with a library of data structures and methods. The basic objects for experimenting with groups would include the following:

- Monoid Elements which could be represented as:
  - $\{0, 1, 2, ...\}$ or
  - $\{\varepsilon, a, b, c, d, e, f, ...\}$ or
  - $\{e^0, e^1, e^2, ...\}$ or
  - or other preferred styles

  They have the following facilities:

  - a unique identity element
  - a concatenation method
  - an image method (for debug display).

- Group elements which would include the INVERSE of non-identity monoid elements, which could be represented as:
  - $\{-1, -2, ...\}$, or
  - $\{A, B, C, ...\}$, or
  - $\{A^{-1}, B^{-1}, C^{-1}, ...\}$ or
  - $\{a^i, b^i, ...etc.\}$
  - etc.

  Group elements must have methods for the following.

  - catenate
  - negate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

– convert them to and from monoid elements

– image function to return the display image of the signed element

3.2.3 Summary of Data Structures

- Braids are multiple copies of an exponential number of permutations, so random wanderings through the naive representation of a braid in memory (10000 42 -17586 -1 -42 ...) can be expected to take an exponential time in log space for solving the braid problem.

- More intelligent meandering may or may not be possible in log space. Epstein proves that there is an $O(N^2)$ algorithm for the braid word problem using deterministic automatic group techniques, which represent the segments of up to n geodesics up to n long as states in finite state machines, taking up as much as $O(N^2)$ space.

- Instead, this research uses linear space data structures, whose size is $O(N + K)$, that is, linearly proportional to the number of crossings plus strands. For example, although there are several linked lists per strand, not every bead can be a member of every list. Members of these lists are shared, such that each bead is member of at most 5 of these linked lists, severely limiting the total space taken by the linked lists. A small amount of data per bead is stored, including the links for all lists with each bead, making the total space proportional to the number of beads. However, there is also a table of strands giving the first bead on each strand, so the total space is proportional to the sum of the strands and the crossings, making the data structures linear space. The action tables and finite machines are of fixed size, not expanding with the number of strands or the number of crossings, and thus do not contribute to the order of magnitude of the space for the data structures.

- The skein of linear linked strand lists (storing both active crossings and fences) catches palindrome reductions immediately. They avoid having to search across strands to find the next bead on the strand. Since this search is done each time a crossing is examined, and examinations occur on the neighbors of each collapse or accumulation of a crossing, this saves a fraction, f of the beads on 5 strands, for each collapse. Each time a crossing is accumulated onto a strand (and that happens exactly N times), the neighboring 4 strands are examined (2 on each side). There at most $N \times 5$ crossings (beads) on those 5 strands (to account for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the fence beads). Collapses happen up to $N/2$ times in an advancing algorithm. Each time a collapse occurs, the neighboring strands are examined for additional collapses. Therefore. if there were no data structure equivalent to these linked lists, searches through the braid for potentially up to $20fN^2 + (20/2)N^2$ would occur. which is an $O(N^2)$ process. although with a small coefficient. Each of these searches is replaced with examining the **next** bead from the starting bead. making these searches each take an amount of time proportional to the size of the collapse being recognized, not the number of crossings or the number of strands. However. this is not a constant amount of time. it is a linear amount of time repeated fN times, because one of the patterns (aB^nAB) and its symmetries, results in the FSA examining, at a maximum, every crossing. Now this could only be done five times for each other crossing not in the pattern, but that is still $O(N)$. To see this. assume that an aB^nAB pattern is being checked where n is 400 long. Then that will be checked at most five times for each **other** crossing, which is $-n$. Then only $5n(N - n)$ examinations are needed. Again. even though the coefficient will be small, this is $O(N^2)$ examinations. This coefficient is actually so small that it might be beneficial to abandon worst case timing and do average timing. since this peaks when n is $C/2$ and is a small multiple of C in the majority of cases. However, this is just one source of timing complexity greater than linear.

- The scheduling table avoids having to re-check beads for reductions. by scheduling only those beads that are in the neighborhood of the beads that were recently reduced or accumulated. Without an equivalent data structure (such as the program stack if implemented recursively) to connect these neighbors together, they would have to be recomputed based on certain boundaries. such as a rectangle containing a certain fraction of the beads and a certain fractions of the strands. each time a node is examined. This rectangle would be a fraction of the braid varying from three strands to the whole braid, and from three beads to the whole braid minus 1. To reduce it fully would take an amount of work that would similarly vary among tiny fractions of the whole braid. but that fraction would be multiplied by a linear number of beads to examine, so it would take at least linear time to do the extra work and probably more. The scheduling table does this by setting everything to be examined into the scheduling container once and the main loop repeatedly takes the next item to process from the schedule in constant time per scheduled item. The number of scheduled items remains small. increasing with the number of strands and the number of crossings, whenever cohesion is lost, where independent horizontally and vertically factored braidlets proceed in parallel,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Braid	K	N	Max Slots
Non-ID	32764	32764	12
Identity	120,000	1.200	8
Identity	120.000	1.200	14
Identity	50	120.000	18
Max Schedule PC	200	84,000	25
Max Schedule PC	5	100.000	25
P-R Example of order 1	-	6	3
P-R Example of order 2	-	128	3
P-R Example of order 3	-	810	3
P-R Example of order 4	-	3,072	3
P-R Example of order 5	-	8.750	3
P-R Example of order 6	-	20.736	6
P-R Example of order 7	-	43,218	6
P-R Example of order 8	-	81,920	6
P-R Example of order 9	-	144.342	-
P-R Example of order 10	-	240.000	6
P-R Example of order 11	-	380.666	-
P-R Example of order 12	-	580.608	-

Table 3.1: Sample Schedule Sizes

using up more scheduling space. The Braid Lab Software optionally monitors usage of the schedule space. Table 3.1 gives some sample schedule sizes. in terms of the maximum number of schedule slots used to schedule beads for examination after consuming or reducing nodes.

The P-R counterexamples themselves would take 3 schedule slots no matter how big they are. What was run here was Fully Twisted Identity braids constructed using the P-R counterexample of the given order. Since the P-R counterexamples are extremely cabled, that is highly cohesive, there was little knotting that could occur, so few schedule slots were taken. Some of the examples in this table have not been run yet with scheduling monitored, and the number of strands in the P-R counterexamples were not counted, just the crossings. Since the P-R counterexamples grow a number of cables of cables, the number of strands increases at least quadratically with the order of the counterexample, and the number of crossings

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

increases with the fifth power of the order.

- The bit matrix added to the scheduling table linked list provides instant random access to the schedule so that nothing is multiply scheduled. This avoids having to search the schedule each time an event occurs. Such a search would be linear in the number of items scheduled. and it would be repeated for each search, which is linear in the number of crossings. since three beads are scheduled for each crossing as it is accumulated into the advancing braid. and 6 beads as each crossing is reduced. Therefore the number of searches that this data structure saves is $9NS_{ch}$. where S_{ch} is the number of scheduled items. which in the worst case is probably logarithmic in the number of crossings and strands. Since. even with many beads. there may be schedule sizes of only a few dozen. this might be considered linear in the number of crossings for the range of braid sizes that are practical for cryptographic work on a uniprocessor. However, the logarithmic increase is multiplied by the number of parallel processors that successfully break the cohesion of the braid and actually start reducing in parallel. in which case the worst case would be 9 per crossing all at once if the processors ensure that they never touch a bead some other processor is working on. or unlimitedly large if an unlimited number of processors are permitted to simultaneously schedule the same bead. However. no processor needs a scheduling table of more than $9N$. so. with P processors. the scheduling structure will never exceed $9PN$ slots. This is interesting since the skein. the action matrix. the finite state machine. and the braid itself could all be in shared memory and would not require additional storage; however, there would have to be a structure, perhaps an interrupt. or a bit array. or some way to tell everyone immediately a crossing has been reduced. and. in addition. there would have to be a way to negotiate two different processor who are working on contradictory reductions. which in turn would require still more data infrastructure, namely, an entire copy of the skein per processor in this case.

- The action matrix concentrates attention on one strand at a time. taking the place of a large number of IF statements which would be necessary if this little decision table did not exist to handle things like $aBAB$ is reducible if the BAB portion is safe from fences, that is. if there is no $aBcAB$ or $aBACB$, etc., but $acBAB$ is okay because it commutes with the initial a. Rather than actually checking for fences, strands can just be examined until a reduction appears possible. then use the action matrix to check the fences all at once. It takes a small constant amount of space (about 1000 bytes) and will not increase until this research

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

discovers additional patterns which the current data structures do not correctly reduce.

- The finite state machine originally just recognized the four levels of deception the research forced it to, by repeatedly generating identity braids until one was found which had an ambiguous reduction. Then a new state or states was added to the finite state machine. In its current state it takes up about 200 bytes. It is implemented as a meta-FSA, which is a FSA implemented on top of another FSA. This happened because of the symmetric nature of the states. The meta-FSA implements reverse symmetry (frontwards and backwards $aBAB$ becomes $Abab$), inverse symmetry (a becomes A), and reverse-inverse symmetry ($aBAB$ becomes $babA$), and takes up about 100 bytes of data. Removing meta-FSA to combine it with the FSA as a single level FSA would make it run a small linear factor faster.

  However, additional research, not completed at this time, has expanding it to subsume the function of the action matrix, using a meta-meta-FSA that simultaneously checks fences as well as the current strand. Fence checking now only occurs when reductions appear possible which is not often, at the worst case is $9N$ times. So combining an advancing reduction recognizor with an advancing fence checker could shave off some more coefficient from the linear amount of time this takes, but will not reduce the order of work being done. The meta-meta-FSA would need to subsume all symmetries of 6 or fewer length words on an alphabet with double fences extending 2 in each direction: $abcdeABCDE\epsilon$. Thus there would be 11^6 = 1,771,561 states. This is currently in the process of implementation as an FSA on top of the FSA on top of the FSA that does the recognition. However, if a proof ever arises and stands that all braids can be recognized by a certain FSA, then a million states is not a lot under current technology, particularly if it gives an advancing algorithm in linear space. However, if the million states is just the front end of a quadratic space algorithm, emulating many parallel insertions of reductions at many parts of the braid at the same time, then it is extremely cumbersome and beyond the reach of current memory banks.

- The Circular List. Since the circular list opens up additional possible reductions faster, it would make identity braid recognition faster, even though the circular list was invented for the purpose of examining the conjugacy problem. Certainly conjugacy operations such as rotating the list around a circle would make non-identity braids reduce to possibly non-isotopic braids. However, the case is easily shown for identity braids that conjugacy reductions will not change the fact that they are still equal to the identity, because anything conjugate to the Identity

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is equal to the Identity, as follows:

$$\begin{aligned} a^{-1}ba &= \epsilon \\ ba &= a\epsilon \\ ba &= a \\ baa^{-1} &= aa^{-1} \\ b\epsilon &= \epsilon \\ b &= \epsilon \end{aligned} \tag{3.1}$$

Therefore, conjugate reductions (rotating the lists around the circle and adding identities at any point) are permitted, and will reduce an identity braid faster, even though it invalidates a non-identity braid. If one were examining minimal word reductions of non-identity braids, then conjugacy reductions are forbidden.

3.2.4 Linear Double Linked Lists

Since enhanced double linked lists are part of the skein data structure, and, in particular, are what makes it possible to have a linked (circular) skein, it will be interesting to note a resemblance the permutations the braid winds through and permutation bugs in double linked list code. Since existing tested linked list packages were used, the author was amazed that almost all the development time for the Braid Lab Software was spent in using those linked list packages to tie things together, such as a bead with its fences, a bead with its ancestors, and a bead with its dependent bead, that is, the bead which blocked the FSA from going into its next state towards recognizing a reduction.

Code implementing (or using) double linked lists, like most software code, has order dependencies on the lines of code. An example of order dependencies in elementary algorithms is swapping two elements, B and D, in a double linked list (DLL). A DLL can be thought of as a string (line or circle of beads).

HOME$- A - B - C - D - E -$ HOME

If iterations across home are permitted, instead of forcing termination there, the list is a geometric circle instead of a line. In other words, ignoring HOME, if $A = E$ there is a circle. In this swapping

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$B\leftarrow.\rightarrow$	:=	D	$A.\rightarrow$	:=	D
$B.\rightarrow$	:=	$D.\rightarrow$	$B.\rightarrow$	:=	$D.\rightarrow$
$D\leftarrow.\rightarrow$	:=	B	$C.\rightarrow$	:=	B
$D.\rightarrow$	:=	$B.\rightarrow$	$D.\rightarrow$	:=	$B.\rightarrow$
$D\rightarrow.\leftarrow$	:=	B	$E.\leftarrow$	:=	B
$D.\leftarrow$	:=	$B.\leftarrow$	$D.\leftarrow$	:=	$B.\leftarrow$
$B\rightarrow.\leftarrow$	:=	D	$C.\leftarrow$	:=	D
$B.\leftarrow$	:=	$D.\leftarrow$	$B.\leftarrow$	:=	$D.\leftarrow$

Table 3.2: Swaps Following the Arrow Diagram Clockwise

example. there is no algorithmic difference if $A = E$ because only the eight internal arrows are updated.

$A \leftarrow B \rightarrow C \rightarrow D \leftarrow E$ or$A \leftarrow B \leftarrow C \leftarrow D \rightarrow A$

There is no conflict since. on the left. the two pointers to the left of A or the right of E are ignored.

However. identifications or interchanges between A and E could affect the algorithm. There are 4 possibilities:

- C=ε
- B=D
- A=ε. C=ε. and E= ε. so that B and D are separated
- C=ε or A=ε or C=ε. so B and D touch.

If B and D are separated then the following algorithm swaps the 8 links connecting B and D to the list. Following the arrow diagram clockwise. the swaps in Table 3.2 occurs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B← . →	:=	D	A.→	:=	D
D←.—	:=	B	B.—	:=	B
B.→	:=	D.→	B.→	:=	D.→
D.→	:=	B	D.→	:=	B
D→ . —	:=	B	E.←	:=	B
B→.—	:=	D	D.—	:=	D
D.—	:=	B.—	D.←	:=	B.←
B.←	:=	D	B.←	:=	D

Table 3.3: Final DLL Non-Linked Algorithm

Example DLL-1

Note that aliasing A to E does not change this algorithm because only A.— and A.→ are changed in one place each, with no cross-impact.

Note also that changing the order of B and D in the algorithm does not change the algorithm because of its symmetry (meaning it is circle capable).

Note also that if B=D the algorithm still works because the nonexistent C is first (in step 3) aliased to D.— which points at B, and in the second case, it is aliased to B.*rightarrow* which points at D.

However, if B and D touch, two kinds of aliasing problems are introduced into the solution, metathesis and invalidation. The metathesis required is to swap the order of Steps 2↔3 and Steps 6↔. The invalidation is that Steps 4 and 8 need to be de-referenced, because their aliasing (B=D) would cause loops in the list. Thus, A—B—D—E changes to A↔D↔B↔E with no C.

Therefore these indexes must be de-looped (de-referenced). The final algorithm is represented in Table 3.3.

The order change works whether B and D touch or not. But the dereference is an algorithmic difference requiring an IF statement. Now, examining the possible temporal aliasing due to the order of assignment, the old value of B and D must be stored. The assignment problem can now

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Case I	$A \leftrightarrow B \leftrightarrow C \leftrightarrow D \leftrightarrow E$,	$C \neq \varepsilon$
Case II	$A \leftrightarrow B \leftrightarrow D \leftrightarrow E$,	$C = \varepsilon$
Case IV	$\varepsilon \leftrightarrow D \leftrightarrow C \leftrightarrow B \leftrightarrow A$,	$C \neq \varepsilon$
Case V	$\varepsilon \leftrightarrow D \leftrightarrow B \leftrightarrow A$,	$C = \varepsilon$
Case III	$A \leftrightarrow b \leftrightarrow \varepsilon$,	$B = D$. $C = \varepsilon$

Table 3.4: Cases for Moving D After B

be partitioned: there is no interference between the forward and backward arrows (next and prior pointers).

In the case where B touches D, there are no temporal anomalies - the assignment to D.← follows the reference to D.→ and similarly for B.←. However, the extra level of reference to B.→ in the case where B does not touch D causes an assignment anomaly. The anomaly involves two situations - the swaps of B.→ and D.→ and the swaps of B.← and D.←. Therefore temporary storage of one pointer to be reused in these 2 situations is required for swapping. Now, it is an interesting question, how this kind of analysis can become more automatic.

Moving a node D after another node b has the 5 proximity cases given in Table 3.4.

The algorithm for moving D after B is given in Table 3.5.

3.2.5 Time Saved By the Linked Lists

Each linked list saves an amount of time at least equal to searching half the length of that linked list once, and at the worst case $N/2$ times, for example, in $dfbc^nBCFb^ncD$, where $bc^nBC \to CB^n$ (requiring $n+2$ crossing look-ahead), reduces $dfbc^nBCFb^ncD \to dfCB^nFb^ncD \to dCB^nb^ncD \to dCcD \to \varepsilon$. Further, the linked lists can point as far as from one end of the braid to another, for example, in Dca^kA^kCd, although in an advancing algorithm, no link extends longer than length $n/2$ in an identity braid. However, in a non-identity braid, links can extend the entire length of the braid, for example, Db^nd.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm	I	II	III	IV	V
B→:=D	B→:=D	B→:=D	B→:=B B→:=E	B→:=D	B→:=D
D←.→:=D→	C→:=E	B→:=D→ B→:=D	B→:=B→	E→:=D←	E→:=D←
D→:=B←	D→:=C	D→:=B→ D←:=B→.→	B→:=B→	D→:=B←	D→:=B←
B→.←:=D	C←:=D	D←:=D D←:=D←	E←:=B	A←:=D	A←:=A
D←:=B	D←:=B	D←:=B	B←:=B B←:=B←	D←:=B	D←:=B
D←.←:=D←	E←:=C	E← :=D← E←:=D	E←:=B← E←:=B	C←:=E	B←:=E
	No errors or metathesis	2 errors 2 metatheses handle as NOOPS: No errors or metathesis	1 error 2 metatheses handle as NOOPS: No errors or metathesis	No errors or metathesis	No errors or metathesis

Table 3.5: Algorithm for Moving D After B

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

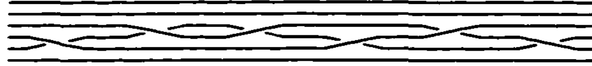


Figure 3.4: DdcdCD

Therefore, based on these examples, while the braid word problem can be solved in $O(logN)$ space, and it can be solved in $O(N^2)$ time, it probably cannot be solved in $O(logN)$ space with an $O(N^2)$ time for all braids. If such an accomplishment occurs, then that algorithm has the effect of emulating all these horizontal and vertical linked lists in $O(logN)$ space in $O(N^2)$ time.

The reduction $cdCD \rightarrow DdcdCD \rightarrow DcdcCD \rightarrow Dc$ is illustrated here.

3.2.6 Circular Linked Lists

A circular linked list requires little code beyond a linked list, and, as a matter of fact, the circular linked list code used in this research is quite a bit shorter than the linear code because there is no HOME node to start and end the list and to check for at each stage of each method. The list is a true circle, and deleting an item from the list simply makes it a singleton circle that points to itself.

The usage of circular linked lists is almost identical to that of linear lists. For example, searching a circular list merely stops when the search arrives at the current node again, instead of when it arrives at the HOME node. The two algorithms are given in Table 3.6.

The interesting part about circular linked lists is not so much how similar they are to linear linked lists, but how similar they make algorithms that use them similar to algorithms that use linear linked lists. In a linear linked list there are methods to get and set the prior and next nodes, methods to retrieve and set the data referenced by each node, search loops, methods of moving nodes from one part of the list to another, deletion of nodes from the list, and finally, insertion of nodes from outside the list. This latter method, insertion, is the only one which differs. In a linear linked list a node is inserted that usually has null pointers for prior and next. In circular linked lists, since entire circles can be merged, a singleton is inserted, that is, a node whose prior and next

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
LINEAR LINKED LIST SEARCH

stop := HOME
loop
  current := loop.get_next;
  exit when current = stop;
  braid.process (current);
end loop;
```

```
CIRCULAR LINKED LIST SEARCH

stop := current;
loop
  current := loop.get_next;
  exit when current := stop;
  braid.process (current);
end loop;
```

Table 3.6: Linked List Searches

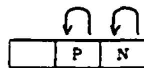


Figure 3.5: Singleton

pointers point to itself as in Figure 3.5.

Implementing Methods for Circular Linked Lists

Circular linked lists can be handled one node at a time, with the following methods. **Delete_node** unlinks the node **index** from the circular list and makes it into a singleton. **Move_node** moves the node **index** from the same linked list after the node after.

```
delete_node (index);
move_node   (index, after);
```

Circular linked lists have an additional method. **Move_circle** moves the entire circular list of **index** after node after, combining two independent lists.

```
move_circle (index, after);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

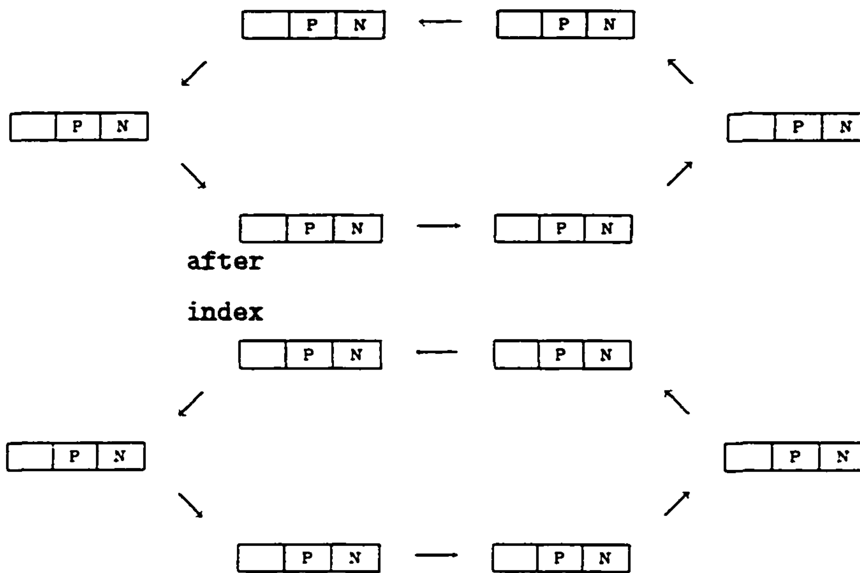


Figure 3.6: Move Circle

For this example, two separate circular linked lists are drawn clockwise, with only the **next** arrows showing. The bottom list is to be inserted at the node labelled **index** after the node in the upper list labelled **after** in Figure 3.6.

Define **move_circle** to connect the two circular lists in such a way that if the two circular lists are drawn with their **next** arrows clockwise, and placed next to each other with the **index** node of the circle to be inserted nearest the **after** node of the circle into which it will be inserted, then the result will be a single circle with clockwise **next** arrows. The consequences of this definition are that four of the arrows are rotated 90 degrees, keeping all the other arrows the same, and the expression of the postconditions must reflect this happening. Two of the arrows are shown because they are **next** arrows, and the other two are the inverse of these (**prev** arrows). The end result will look like in Figure 3.7.

In accomplishing these postconditions, these postconditions are interdependent. This interdependency may vary, depending on whether the **index** circle, the **after** circle, or both are singletons. In all cases, simple dependencies in the directed acyclic graph of the consecutive equations result from the fact that something **indexed** by a variable cannot change, after that variable itself has changed. Two ways to handle these interdependencies are either to express them absolutely or operationally. Absolutely, the postconditions would be expressed in terms of copies of the values the arrows had

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

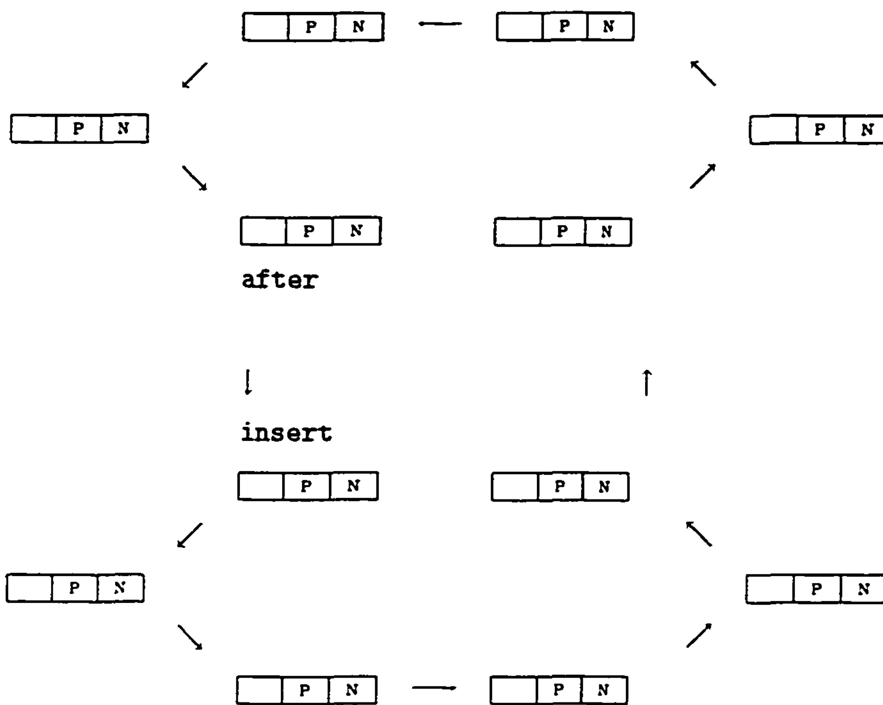


Figure 3.7: Insert Circle

before they started to be rearranged. Operationally, the postconditions are expressed in a particular order using the arrow values as the postconditions are made true sequentially. To eliminate the copies, which in this method happen to be needless, the postconditions are expressed here operationally, to be executed sequentially. The main difference, is that things like `index.prior` must not be changed until things that depend on its previous value are changed, such as `index.prior.prior` or `after.next.prior`.

The postconditions in Table 3.7 result in the correct 4 links being changed, unless additional dependencies crop up when `index` or `after` are singletons. There are 4 ways this can happen in a circle.

A. Neither circle is a singleton (displayed above).

B. The `index` circle is a singleton. A single node is to be inserted into an existing circle as in Figure 3.8. However the node must first be changed into a singleton by setting its `prior` and `next` to point to itself.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Preconditions
1.  index is not in the circle of after

Postconditions

 1.  index.prior.prior = unchanged
 2.  index.prior.next  = after.next
 3.  after.prior.prior = unchanged
 4.  after.prior.next  = unchanged
 5.  index.next.prior  = unchanged
 6.  index.next.next   = unchanged
 7.  after.next.prior  = index.prior
 8.  after.next.next   = unchanged
 9.  index.prior       = after
10.  index.next        = unchanged
11.  after.prior       = unchanged
12.  after.next        = index
```

Table 3.7: Post Conditions A

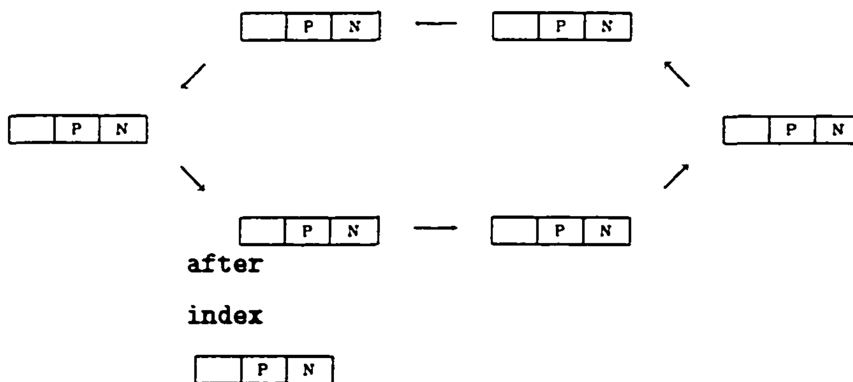


Figure 3.8: Insert Singleton Into Circle

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

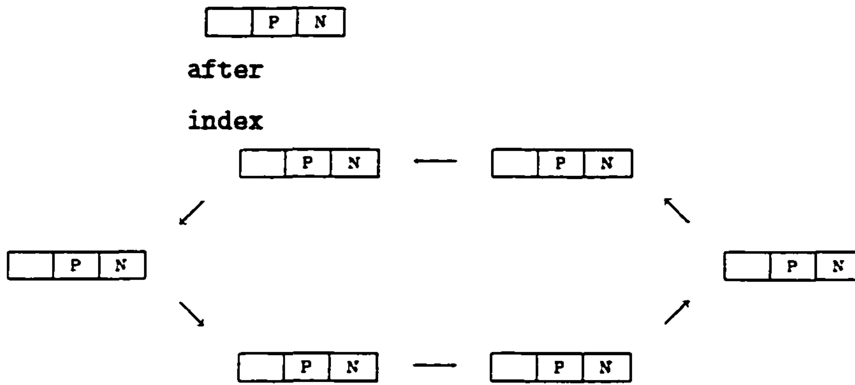


Figure 3.9: Insert Circle Into Singleton

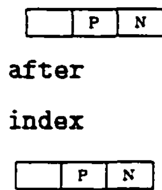


Figure 3.10: Insert Singleton After Singleton

C. The `after` circle is a singleton, as in Figure 3.9. A circle is to be inserted after an existing singleton. This has the end result identical to inserting the singleton `after` before the node `index` in the `index` circle. However, the code must work when presented this seemingly backwards way, because the algorithm should not need to check whether the circles are singletons, so that a single algorithm works in all 4 cases.

D. Both the `index` and the `after` circle are singletons, as in Figure 3.10. In this case an unordered doublet is created. The double it unordered because when a circle has only two nodes each of them is both clockwise and counterclockwise from each other.

Note that these circles are unusual algebraic structures in that there is no identity circle. Deleting a node from a singleton circle first destroys the circle, then turns the node into a singleton, recreating exactly the same circle.

In Case A, the four conditions, 2, 7, 9, and 12, which change can be implemented by assignment statements, because the postconditions were expressed operationally.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Case B, when **index** is a singleton, interference comes from the fact that, in this case, **index.next** = **index.prior** = **index**. These facts cause additional dependencies. Any earlier postcondition involving **index.next** or **index.prior** could be affected by later postconditions which might contradict it. The sequence of Conditions 2, 6, and 10 leaves **index.next** set to **after.next**, which is correct for Case B. The sequence of Conditions 1, 5, and 9 leaves **index.prior** set to **after**, which is correct for Case B. Therefore, no changes are required for Case B.

In Case C, when **after** is a singleton, interference comes from the fact that, in this case, **after.next** = **after.prior** = **after**. These facts cause additional dependencies. The sequence of Conditions 3, 7, and 11 leaves **after.prior** set to **index.prior**, which is correct for Case C. The sequence of Conditions 4, 8, and 12 leaves **after.next** set to **index** which is correct for Case C. Therefore, no changes are required for Case C.

In Case D, when both are singletons, interference could come from the facts that (**index.next** = **index.prior** = **index**) as well as the facts that (after.next = after.prior = **after**). These cause precisely the same interferences handled in the discussion of Cases B and C above, neither of which interfere (cause additional dependencies) with each other. Therefore, no code changes are required.

The same assignment statements in the same order that implemented Case A will also implement Cases B, C, and D.

Therefore, the entire move of one independent circle into another circle is implemented by the 4 assignments:

```
index.prior.next  := after.next;
after.next.prior  := index.prior;
index.prior       := after;
after.next        := index;
```

While moving one circle into another circle is thus simple and efficient, it is more complex to move a node in a circular list to another spot in the same circular list. This is because there are additional second order dependencies to be handled to break links, since running the above 4 assignments would double link the circle into a figure 8, confusing any iterations that run around the circle. All of these additional dependencies in a circle are as small as a → b → c → d → a. The postconditions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are not only different from the case of moving circular lists, but they must be checked to see if they differ among the four cases:

E. Move a nontouching node after a node (e.g. move c after a) F. Move the prior node after that node (e.g. move d after a) G. Move the next node after that node (e.g. move b after a) H. Move the node after itself (e.g. move a after a)

In Case E. the following postconditions must be established.

```
PRECONDITION
1.  index is in the circle of after

POSTCONDITION
1.   index.prior.prior = unchanged
2'.  index.prior.next  = index.next
3.   after.prior.prior = unchanged
4.   after.prior.next  = unchanged
5'.  index.next.prior  = index.prior
6.   index.next.next   = unchanged
7'.  after.next.prior  = index
8.   after.next.next   = unchanged
9.   index.prior       = after
10'. index.next        = after.next
11.  after.prior       = unchanged
12.  after.next        = index
```

These postconditions result in 6 links being changed, and in this order, they are both absolute and operational. However, in the other cases they will only be operational because of the additional cross dependencies. The four primed postconditions represent differences between moving an independent circle versus moving a node in the same circle.

In Case F, after.prior is **index**, and **index.next** is **after**. This affects Postcondition Sequences (3, 4, 9, 10) and (5, 6, 11, 12). No changes are required operationally, when the postconditions are enforced in the order given here. So, no changes are required for Case F.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Case G, which is a no-op, because **after.next** is already **index** and **index.prior** is already **after**, the operational conditions result in changes which are undone (2 is undone by 12), and (7 is undone by 9), which is satisfactory, and changes which are not undone, namely 5' and 10'. So this must either be special-cased as a no-op, or the equivalent of these two postconditions must be changed to:

```
 5'. index.next.prior = index (that is, it must be unchanged)
10'. index.next       = after.next.next (i.e., unchanged)
```

An example of an equivalent, rather than actually changing them both, would be to change only 5' and take care of 10' by moving 12 before 10' in the list.

In Case H, which is also a no-op, has **index=after**. Now 5' is undone by 7'. If it is not to be special-cased as a no-op code, the following postconditions must, therefore, be changed:

```
 2'. index.prior.next = after (or index)
 9'. index.prior      = unchanged
12'. after.next       = unchanged
```

It appears that the easiest way to establish the postconditions in cases E, F, G, and H is to special-case G and H to no-ops, resulting in the following code:

```
if AFTER=INDEX or AFTER.NEXT=INDEX then
   noop;
else
   index.prior.next := index.next
   index.next.prior := index.prior
   after.next.prior := index
   index.prior      := after
   index.next       := after.next
   after.next       := index
end if;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Moving the individual node, **index**, to a new position in same circle is equivalent logically to (satisfies the same absolute preconditions as) deleting that node, **index**, from the circle so it becomes a singleton, then moving that singleton after node **after**. So, the code for moving in the same circle can be replaced with:

```
if AFTER=INDEX or AFTER.NEXT=INDEX then
   noop;
else
   delete (index);
   move_circle (index, after);
end if;
```

The only possible special case for this short-cut would be the case where **index** was the only thing in the circle to begin with, so deleting it creates a new singleton circle containing itself. Since the only node in that circle to move it after is itself, the IF statement takes care of this case, technically avoiding executing **move_circle** when the precondition for **move_circle** is not met, because the node is already in the circle.

3.2.7 Building and Collapsing the Cayley Graph

One way to attack the (sometimes) unsolvable group word problems is to incrementally build a Cayley Graph for the given word, hoping it collapses to a finite topological disk.

Cayley Graphs are interesting from several points of view:

- They are usually infinite, so there is no obligation to try to finish printing all of them.
- Any given portion of them can always be drawn as a planar graph. If a line crosses another line, just trace a word (list of edges = list of characters in word) to the outside of the Cayley graph (latest currently drawn boundary), say *abccccb*, and draw it backwards going outwards, *bccccba*, duplicating as many nodes as desired.
- There are different levels of algorithms to collapse a Cayley Graph (reduce a word problem).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Collapse nearby nodes because the edges from a circle, for example, $a \xrightarrow{x} b \xrightarrow{x^{-1}} c$ implies that $a = c$.
- Collapse nearby edges because the nodes from a circle, for example, $a \xrightarrow{x} a^{-1} \xrightarrow{y} a$ implies that $x = y^{-1}$.
- schedule collapses, for example: $a \xrightarrow{x} b \xrightarrow{y} c \xrightarrow{x^{-1}} d \xrightarrow{x^{-1}} g \xrightarrow{y} g^{-1} \xrightarrow{x^{-1}} g$ when $g \xrightarrow{y} g^{-1} \xrightarrow{y^{-1}} g$ shows that $y = x$, then $a \xrightarrow{x} b \xrightarrow{x} c \xrightarrow{x^{-1}} d \xrightarrow{x^{-1}} g \xrightarrow{x} g^{-1} \xrightarrow{x^{-1}} g$ so $a = g$, $b = g$, $c = g$, $d = g$, and $x = e$. But there is no limit to the amount of time that might be spent building and traversing the Cayley Graph before the appearance of the lucky simplification required to trigger this collapse.
- schedule mega-collapses, where multiple sets of collapses impinge on each other, triggering further collapses inside existing collapses that are still in the process of collapsing, if one imagines that a collapse is done by scheduling one node or one edge at a time.

- As a Cayley-Graph-build progresses, a greater sense of assurance is felt that some kind of progress is being made in understanding the word for which the Cayley Graph is being built. This assurance is faulty because the word might not be the identity, so the loop might search forever for a path through the Cayley Graph that forms a closed circle (disk). Nevertheless, this gives something to input to a visualization tool at each level to which the Cayley Graph is built.

3.3 Levels of Deception

The level of deception is initially taken to be the number of levels of mega-collapse required to solve the braid. This will be refined in the next Section, to consider the cohesion and coupling of the braid, to derive more useful measures of braid complexity.

3.3.1 Level 0 Deception for Pure Braids

The first data structure for a braid is a simple array of crossings (bB). It is useful for Level 0 braids, containing only pairs of identities, ready to reduce. An advancing algorithm is possible for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Level 0 braids by simply reducing whenever the bead b has not yet reached the last crossing and a pair is found. Thus, reduce whenever $b < N$ and $t_b = -t_{b+1}$.

3.3.2 Level I Deception for Local Reference Braids

Local reference braids are anti-palindromes, such as (5 3 1 -1 -3 -5), possibly rearranged using the semi-commutative braid relators, such as (5 3 1 -1 -5 -3).

A Failed Representation

A possible alternative data structure was examined to attempt to reduce the complexity of computing braids. In this data structure crossings are represented by 2 unsigned integers instead of one signed integer. Thus when strand three crosses strand two this new structure would represent it as $[3, 2]$ instead of -2. This doubled the storage of the crossing which was a negative result. It seemed to make a poset algorithm more aesthetic in a few lines, but it took longer and took up more space, so that did not balance the doubling of the storage.

A Working Representation

A data structure that resolves Level I Deception braids in linear space and linear time is a stack per strand, tracking the unreduced crossings on each strand. Crossing are represented by their integer value, instead of by the pairs of integers above. Running such an algorithm against random identity braids immediately discovers that most braids are not pure or locally coupled.

3.3.3 Level II Deception for Semi-Global Reference

The twisting relators combine with the semi-commutative relators to make this level of deception. Twisting relators can be applied two symmetrical ways: as written, and in inverse. Thus, $cdc \rightarrow dcd$ can also be applied as $CDC \rightarrow CDC$. However, one cannot "mix case." That is, $cDC \not\rightarrow CDc$, $cDc \not\rightarrow DcD$, etc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$bcbC$	$dbcbC$ reduces, no interference
$bcbC$	$bdcbC$ reduces, interference commutes away since $db = bd$
$bcbC$	$bcdbC$ does NOT reduce
$bcbC$	$bcbdC$ does NOT reduce
$bcbC$	$bcbCd$ reduces, no interference

Table 3.8: Interference vs. Reduction

When using a double-linked list to track the next crossing on each strand, a relator like $bcb => cbc$ which implies $bcbC => cbcC => cb$ which becomes the reduction $bcbC => cb$ can be deceptive. It cannot always be applied (Table 3.8 because of non-commutative interference from neighboring strands. For example, where can a crossing d or D appear within the reduction, yet still permitting the reduction to take place? There are 5 combinatorial possibilities, of which 3 permit the reduction to take place, and 2 do not.

Such interference must be detected as a non-reducing blocking state by any successful braid algorithm. In the second line of the table, the braid $bdcbC$ reduces, even though the d appears in the middle of the reduction, because b and d are more than one apart, so they commute, as shown in Figure 3.11.

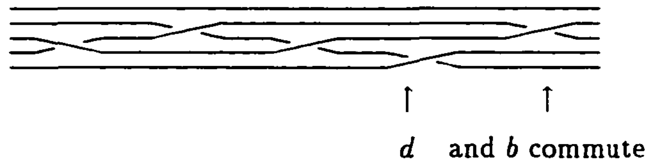


Figure 3.11: Braid bdcbC where c=3

The middle entry in the table cannot reduce because, d only commutes with b, which does not get it out of the way of c and C, as shown in Figure 3.12.

This leads to a modification of the data structure to permit tracking of these blockages or fences. Time and space could be traded off in implementing these.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

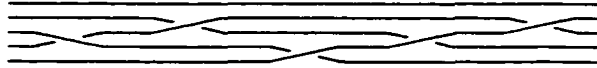


Figure 3.12: Braid bcdbC

Alternative 1

At a sacrifice of time, each element of each reduction could be cross-checked against the parallel reduction in the adjacent strands as in the Braid Lab Software function `testword_using_fence_list`. The fence-list cross-checking algorithm must go through each reduction and specifically check each possible fence interference. For example, in $bcbC \rightarrow cb$, an interfering d cannot occur between the internal b and its neighboring c and C. Similarly, an a could not occur in the between either of the bs and the internal c. Therefore, out of $bcbC$, the last three crossings must be checked for a d fence, and the first three crossings must be checked for an a fence. This fence-checking requirement will get more arduous as lengthier patterns are introduced, to search for and reduce.

Alternative 2

Although something like this is not ordinarily tried because of the exponential growth in the inputs, the following experiment was run, using Braid Lab Software function `tsallwords`. This function demonstrated that `meta_fsa` correctly stages through every braid of length 7 or less on 5 strands, and that every word that it identified as the identity actually is the identity according to `testword_using_fence_list`.

This did not result in any anomalies, but it did take a long time (a weekend). Much longer braids are not feasible to check like this. In addition, there is another problem. Checking two algorithms against each other on large numbers of braids gives evidence for the stability of the two algorithms, and for their equality on those braids tested, but it does not give evidence that either algorithm is correct on every single one of the braids tested, just that they both got the same result. Since this does not result in progress or counterexample generation, no further runs of this type were done.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Alternative 3

At a sacrifice of a constant amount of space, a FSA could encode all the reductions. Each reduction could be combined with sibling reductions in both of its adjacent strands. The FSA, in order to permit fences on either side of each strand (except on one side of the first and last strand), would have to contain states corresponding to every possible fence. That is, for the 4-length patterns, there would have to be states for possibly blocking on the occurrence of a fence before, after, or between each pair of crossings of a pattern. For example, *abcd* would have to check for fences at $FxFyFzFwF$ to catch every possible type of blockage.

3.3.4 Preliminary Upper Bound

A preliminary upper bound on the maximum number of states would be the number of words creatable from the alphabet $\{a, b, c, d, e, A, B, C, D, E, \varepsilon\}$ of up to 7 long, which is $11^7 = 19,487,171$. However, making use of symmetries and immediate blockages, not nearly all of these states are needed. In the reducible patterns, it would be possible to include only adjacent strands to the starting strand. That is, the x, y, z, and w can include only three strands, the current strand c or C, the prior strand b, B, or the next strand, d, D. The blocking strands could be a, A, b or B, if the reducing strand is d or D. On the other hand, the blocking strands could be d, D, e, or E if the reducing strand is b or B.

This means that the number of states is less than or equal to $6^9 = 10,077,696$. The number of these states can be further reduced by immediately blocking on an initial F, and stopping as soon as a reduction arises, so the final F is never condensed. This reduces the number of states to less than or equal to $6^7 = 279,936$, not counting repeated F's which to not increase the complexity of the data structure or the time it takes to reduce. Further reductions in the number of states can be realized because the fenced b and B are equivalent, and so are d and D, leaving $6^4 \times 3^3 = 1,296 \times 27 = 34992$ states.

We could search in parallel on all strands, but then the fences would have to be extended by the number of strands we are searching on, and would thus need extensive data structures to prevent the parallel processes from deadlocking each other with these additional blocking fences. Since it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

appears that the deadlock prevention would cost more execution time and memory than a non-parallel search, this kind of parallelism would only be worth it in low cohesion braids. While random braids would tend to lose cohesion as they are reduced, the kinds of maximally twisted braids useful for cryptography would tend to remain highly cohesive longer, making this approach less useful. The Braid Lab Software does not include implementation of this deadlock prevention, so the function `word_braid`, for example, is not thread-safe, nor can a skein easily be made thread-safe.

If examination starts only on c or C the number of states is further reduced by a factor of 3 to only 11,664 states, and if a meta-FSA makes use of the symmetry between c and C, there is now a maximum of 5,832 states. Of these, most are blocking. For example, anything starting with cc, regardless of followers or fencings, blocks immediately, eliminating one third of all the states. Other symmetries and pre-blocked states reduce the actual FSA used during the initial experiments to 52 active states plus 23 final states (`blocks` and `reduces`), for a total of 75 states.

This FSA finds all effects requiring fencing 2 strands away on either side of the current strand, which is why it starts at strand c or C, leaving room for fences at a, A, b, B, d, D, e, and E.

Trading Off the Alternatives

Either of these alternatives requires that each crossing be entered into the linked lists as five linked beads, representing the crossing itself, and two fence crossings on each of the adjacent strand lists. For example, the crossing -7 is entered on strand 7, coded as C, meaning the true crossing. It is accompanied by virtual, fence crossings of B on the linked list for strand 6, A on the strand 5 list, D on the strand 8 list, and E on the strand 9 list. Because the strands can be far from tracking each other evenly as reductions arise, the real and four virtual beads must be linked together. Thus the braid is covered in two dimensions by a mesh of horizontal (tracking the next real or virtual bead on each strand) and vertical (connecting the brother and sister bead and the real crossing) linked lists as shown in Figure 3.13. Every algorithm that correctly recognizes these patterns has an internal data structure that is equivalent to this in space or in time (see Section 3.2.5), in the sense that such an algorithm emulates the functionality of these linked lists.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 3.13: Vertical and Horizontal Links

3.3.5 Level III Deception

Upon programming `test_word_fsa` and `test_word_fence_list`, the random braid generator was then engaged. The research at this point becomes more difficult because almost all braids appear to be solvable using `fsa` or `fence_list`. The first braid not solvable by these took millions of iterations of the function `random_braid` to generate. The reason is that in identity braids, there are several ways to reduce the braid.

In reduced form, this counterexample at Level III is a braid requiring the following reduction cd^ncD, for example, cd^ncDFC^nd with fences between the braid cd^ncD and its inverse C^nd. There are several ways to defeat the fencing.

- Use `fsa`, and if that does not give sufficient look-ahead, build further levels on top of that. This is not satisfying, because the number of states T before reduction goes up rapidly with the number of fence levels before reducing by symmetries, and prior blockages. More importantly, if additional fence levels were added, it would be hard to know that further additions would not then still be needed. Therefore, it would be better to avoid using this data structure, if possible.
- Go back to the basic relator substitutions of inserting cC, Cc, $cbcBCB$, and other identity words at random spots in the word being solved. If a braid has K strands and N crossings, then at each crossing in the braid insert one of the random identity words $\{cC, Cc, cbcBCB, cdcDCD, CBCbcb, CDCcdc, \varepsilon\}$ corresponding to relators which apply at this crossing. This is a total of $7^{\#C}$ insertions, which is an exponential number of possible insertions to try. Therefore, it would be better to avoid doing business this way.
- Permit the finite state machine to repeat at the significant spots, as described in Section 3.3.5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cb	cd	$c\boxed{B}$	$c\boxed{D}$
$cb\boxed{c}$	$cd\boxed{c}$	cBC	cDc
$cbcB$	$cdcD$	$cBCB$	$cDCD$
$c\boxed{b}$	$C\boxed{d}$	CB	CD
Cbc	Cdc	$CB\boxed{C}$	$CD\boxed{C}$
$Cbcb$	$Cddc$	$CBCb$	$CDCd$

Figure 3.14: Two-Crossing Reductions

FSA II

The four states starting at the middle strand that do a trivial cancel are the following:

- cc (block)
- CC (block)
- cC (reduce)
- Cc (reduce)

The states which result in a two-crossing reduction from a four-crossing reduction are shown in Figure 3.14

The states with the squares around them represent repeated states, for example, $cdcD$ really means cdc^*D, with the second c repeated 0 or more times.

Counterexample to Algorithm II

A random identity braid of length 40 was spun up in Equation 3.2.

$$
\begin{array}{rllllllllll}
A40 := & (4, & 1, & -1, & 1, & 1, & 1, & 1, & -4, & 4, & 3, \\
& -1, & -1, & -1, & 1, & -2, & -1, & 1, & -1, & 1, & -2, \\
& -1, & -2, & -2, & -2, & -2, & -1, & -1, & -4, & -1, & -1, \\
& -1, & 4, & 2, & 1, & 1, & 2, & 1, & -3, & -1, & 4)
\end{array}
\tag{3.2}
$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The reductions of aA are taken out first, using permitted relations of the first kind ($ab = ba$, when $|a - b| > 1$).

$A40 \approx A16 = (4, 1, 1, 3, -2, -2, -1, -2, -2, -1, 2, 1, 1, 2, -3, -4)$

No relations of the second kind immediately reduce ($aBAB, BABa, Abab, orbabA$). However, there is a relation of the second kind which can be applied, and it leads to proof that $A40 \approx I$. First, applying that relation ($(-2, -1, -2) \approx (-1, -2, -1)$), another isotopic braid is produced, but still not one with an immediate reduction of the second kind.

$A40 \approx B16 = (4, 1, 1, 3, -2, -1, -2, -1, -2, -1, 2, 1, 1, 2, -3, 4)$

This releases two reductions $(1, 3, -2, -1, -2)$ and $(-1, -2, -1, 2)$ of the second kind, and serves as a counterexample to the efficacy of Algorithm II being a solution to the word problem.

It is interesting that more than a million random maximally–mixed braids of length 40 were spun up before this counterexample to Algorithm II was discovered. It is therefore interesting to know how dense these exceptions are, and specifically: is there a decipherable pattern to them which can be used to create Algorithm III which advancingly solves these (and possibly all) braid problems. Since 40 has a number of possible braids which is $\gg$ a million, it would be easier to work with the shortest non-advancing counterexample.

$C5 := (1, -2, -2, -1, -2)$

C5 requires a double application of the second kind of transformation ($abc \rightarrow bab$), when $|a-b| = 1$), and no braid shorter than it can require such an application, since this requirement would be vacuously reducible if it was of length 4, say $(1, -2, -1, 2)$.

This gives the ability to look ahead as many positions as needed to in these multiple application of the second kind of transformation. Multiple applications of ($aba \rightarrow bab$) look like this.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
&bA^0BA \rightarrow bBA \rightarrow A \\
&bA^1BA \rightarrow bABA \rightarrow bBAB \rightarrow AB \\
&bA^2BA - bA^2BAB^0 \rightarrow bA^1BAB^1 \rightarrow bA^0BAB^2 - bBAB^2 - AB^2 \\
&bA^nBA \rightarrow AB^n, n > 0, |A - B| = 1, \text{continuing the induction}
\end{aligned}
\tag{3.3}
$$

This final pattern generalizes the presentation of a braid group, B_1 from

$$B_1 \equiv < A \mid xy = yx, d(x,y) > 1; xyx = yxy, d(x,y) = 1 >$$

to

$$B_1 = B_2 \equiv < A \mid xy = yx, |d(x-y)| > 1; xY^nXY = YX^n, d(x,y) = 1 >$$

for all x and y in A and their inverses X and Y.

This reduces an $(N+3)$ character phrase to an $(N-1)$ character phrase. This can be thought of as a finite state automation (FSA) on each crossing of the braid. It can be programmed as an improvement to the FSA table shared by all crossings and described in the previous section. The crossings store their crossing number and the current FSA state which tells them what to look for next and when they are complete. The reason B_2 is a generalization of B_1, even though it generates the same groups, is that it leads to an advancing algorithm. To do this efficiently each crossing must be represented using a data structure which is efficient for triggering a reduction as soon as it is possible, without continually checking all the as-yet unreduced crossings. A linked list is needed, connecting each crossing to the crossing holding it up (fences blocking it, crossing of the wrong strand or in the wrong order, or zero representing crossings not yet queued on the multi-stack).

To search these linked lists faster, they can be stored both in their original form and also **inverted**, so they relate all crossings blocked by a given crossing to that given crossing. Then no search time be needed at all. Each time a crossing is reduced, and each time a fence is eliminated due to its crossing being eliminated, all of the FSAs on those crossings' inverted linked lists leave their state examined and possibly updated, possibly triggering their own collapses, and in turn updating their own inverted linked list of dependencies. Since the original form of the linked list is also stored,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

there is a fast way of removing a crossing eliminated by a reduction from all of the inverted linked lists. Since a crossing can only be reduced once, the only thing stopping this algorithm from being $O(N)$ queuing operations and $O(N \times K)$ storage is that these inverted linked lists can change as the braid is sequentially queued. In the worst case, a single potential reduction for a giving crossing can be blocked, then unblocked by each of the other crossings, implying $O(N)$ queuing operation. This worst case cannot hold true for every crossing, however.

Define braid coupling at a particular bead as the number of crossings and fences that can hold it up, that is the number of times a blockage is updated in the inverted linked list. The following examples give a feel for the coupling.

Example 1. (No extra examination required.)

$aBBBBB...BAB - aB^{1,000,000}AB - BA^{1,000,000}$

Since each of the million Bs is blocked by the crossing after it, and is cancelled at reduction but never updated, the coupling is $N = 1,000,003$, and if the above example were followed by $a^{1,000,000}b$, then no reexamination of the inverted linked lists are needed. The algorithm time in this case is $O(N)$.

Example 2. (Extra examination required.)

The aA adds coupling that affects the first B, causing it to update its linked list several times, pointing at a, then at the A, then at the Bs, then the A, then the final B. This can happen no more than one half of the crossings (since some are dependees and some are dependers).

Example 3. (It can happen a quarter of the time.)

$$\begin{aligned} &aaaBaBABBb \rightarrow \\ &aaaBBABb \rightarrow \\ &aaBAAb \rightarrow \\ &aBBa \end{aligned} \tag{3.4}$$

This reduction contains a Bb at the right. On the other hand, if the examination were advancing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the left, the Bb pair would not be seen until reaching the right. The point is, in three reductions, the links are rewritten three times completely, and so the algorithm examines a fraction of the other crossings for each of a fraction of the crossings. Even though these fractions may be relatively small, this implies the algorithm is doing $O(N^2)$ re-examinations, though possibly with a small coefficient (which explains experiments seemingly resulting in a linear amount of time for fairly large braids). In addition, averaging over all the cases might result in a shorter average execution, though this example establishes that the algorithm has $O(N^2)$ re-examinations in the worst case. Beads, of course, must be able to hold numbers at least as big as the number of crossings and as big as the number of strands, but this has nothing to do with the number of crossings visited, which is our measure of complexity of the algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example of a Cabling Algorithm

Example (a)

$$\begin{aligned} &aBAB \rightarrow \\ &aABA \rightarrow \\ &BA \end{aligned} \tag{3.5}$$

Example (b)

$$\begin{aligned} &aBAAB \rightarrow \\ &aBAABAa \rightarrow \\ &aBABABa \rightarrow \\ &aABAABa \rightarrow \\ &BAABa \end{aligned} \tag{3.6}$$

Note, as in Example (b) above, that there may be an extra A^{-1} resulting in a braid of length five "reducing" to a braid of length 5. However, if there was an A already visible, this would not happen.

Example (c)

$$\begin{aligned} &aBAABA \rightarrow \\ &aBABAB \rightarrow \\ &aABAAB \rightarrow \\ &BAAB \end{aligned} \tag{3.7}$$

Example (d)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\begin{aligned}&aBAAABA \rightarrow\\&aBAABAB \rightarrow\\&aBABABB \rightarrow\\&aABAABB \rightarrow\\&BAABB\end{aligned} \tag{3.8}$$

Example (e)

$$\begin{aligned}&aBAAAABA \rightarrow\\&aBAAABAB \rightarrow\\&aBAABABB \rightarrow\\&aBABABBB \rightarrow\\&aABAABBB \rightarrow\\&BAABBB\end{aligned} \tag{3.9}$$

Example (f)

$$\begin{aligned}&aBA^{n}BA \rightarrow\\&BAAB^{n-1}\end{aligned} \tag{3.10}$$

These result in a reduction in length of two crossings.

Although searching for this kind of cable may result in large reductions (or, rather, small reductions with a large scope), it is not an advancing algorithm, nor does it solve all braids without searching for other patterns in addition.

Like many algorithms, this cable search has end effects, which require special care when N=0. Equation 3.11 displays how it works for $N = 0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$aBA^0BA \rightarrow aBBBA \rightarrow BBabaBBA \rightarrow BAbabBBA \rightarrow BAbaBA \rightarrow$$
$$BAAabaBA \rightarrow BAAbabBA \rightarrow BAAbaA \rightarrow BAAb \rightarrow \quad (3.11)$$

Equation 3.11 shows that some crossings must be added in order to reduce the braid further, mitigating against being an advancing algorithm.

But the equivalence $aBBA \rightarrow BAAb$ is an interesting theorem in its own right, both as an example of the semi-commutativity of braid groups, and as a component in the table of conjugates, since isotopic braids are always in the same conjugacy class.

Algorithm III

Algorithm II queued a crossing and 4 fences for it, then repeatedly checked all active beads for cancellations as it continued to queue beads. This failed (that is, worked only with backtracking) on patterns of the type aB^nAB. Algorithm III adds the requirement to keep additional data structures: linked lists and inverted linked lists of all blockages.

When a cancellation occurs, Algorithm III uses these lists to reexamine all crossings that are blocked by the cancelled bead, updating their state, causing further cancellation when possible, recursively removing cancelled crossings as soon as they occur. If the bookkeeping is done properly, an already cancelled crossing will never appear on any blockage list.

It is expedient to give priority to removing cancelled crossings (and their fences) over re-examining crossings to update their state. Therefore, a scheduling structure is used (a linked list combined with a boolean flag per element to indicate whether it is already scheduled) to schedule re-examinations, while deletions are done immediately in Figure 3.15.

Old Algorithm (Inefficient Version)

The first optimization was to put in the fence beads so the pattern matches fail where blocked. The collapsible function need only check a hard coded pattern as in Figure 3.16. Additional complexity

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
For each letter $L in the braid
    Add L and its fences onto the strand skein
    For each of the affected strands, S
        For each earlier element E of the skein along strand S
            schedule E for examination
        end loop
        For each bead B scheduled for examination
            For each relation R
                For each symmetric variant V of this relation R
                    if collapsible (E, R, V) then
                        collapse (B)
                    end if
                end loop
            end loop
        end loop
    end loop
end loop
```

Figure 3.15: Old Algorithm (Inefficient Version)

in the collapsible function (such as making a portion of the hard pattern variable or limiting the search to a smaller number of possible lengths) can be traded for fewer iterations in three of the inner loops (i.e. for each earlier element, for each relation, and for each symmetric variant of the relation).

The collapse procedure includes a mega-collapse that is much simpler than for general groups. When a pattern, say $BABa \rightarrow ABAa \rightarrow AB$ is to be reduced a certain number (2) of the pattern elements are deleted, and a certain number of are to be kept and changed. In this version of the algorithm it happens that the ones kept never change, but that is coincidental and must change in the next improvement to the algorithm. Using the collapse/mega-collapse idiom, a template of actions is created, and the algorithm is improved, as follows. To collapse at pattern P of length $\#P$, with $\#R$ relation and $\#T$ letters to be kept, the algorithm in Figure 3.17 is carried out.

The template of actions is encoded as a FSA per strand and the algorithm becomes as in Figure 3.18.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
For each letter being cancelled
   Delete its upper and lower fence beads and its own bead:
       Mark the bead deleted
       Add the 4 predecessor beads onto the schedule
       Add the 1 successor   bead  onto the schedule
end loop

For each bead on the schedule
   For each relation R
       For each symmetric variant V of this relation R
           if collapsible (E, R, V) then
               collapse (B)
           end if
       end loop
   end loop
```

Figure 3.16: First Optimization

```
For each letter L being accumulated
    Add L and its fences as beads on the skein
    For each of the affected strands S
        For each earlier element E of the skein
          if it is within distance 3 along S
            schedule E
          end if
        end loop
        For each bead in the schedule B
            compute template substitution arguments TSAA (B,L)
            For each preprocessed action A
                if A matches TSAA(B,L) then
                    collapse B
                end if
            end loop
         end loop
     end loop
end loop
```

Figure 3.17: Old Algorithm (More Efficient Version)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
For each letter L being accumulated
    Add L and its 4 fences as beads onto the semi-stack skein
    For each of the affected strands S
        For each earlier element E along S
            schedule E
        end loop
        For each bead B in the schedule
            B.state := transition (B.state, L);
            if the new state is reducible then
                collapse B
            end if
        end loop
    end loop
end loop
```

Figure 3.18: Algorithm using FSAs to replace the inner loop

The inner loop(s) checking for patterns have been replaced by a FSA per bead, bumping a state according to a transition matrix and checking whether any blockages have been relieved. To get an FSA per strand, only one copy of the FSA is kept, and each bead tracks its current state, until it reaches a final state (`block` or `reduce`).

Unfortunately, this improvement forces all prior elements to be checked on the strands affected by the crossing and its fences. This excess checking can be dealt with by tracking the dependencies in two linked lists per bead, B. The first will list the prior beads that B is blocking. The second will list which beads are blocked by B.

That way, when a bead i added or reduced, it can know exactly who might be affected. This will save time over the full search because some of the patterns (e.g., a $BBBBBBB...BBBBAB$) contain a series of beads, each blocked by the next bead, except the initial a which is waiting for the final B, and so is the only one on the blocked-by list of the next bead available on the strand (or ∞, representing a bead not yet accumulated).

This improvement to the algorithm uses Blockage Lists to replace the loop For Each Element E along S, as in Figure 3.19.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
For each letter L being accumulated
    Add L and its fences as beads onto the skein
    For each strand S
        For each bead E on the strand S blocked by infinity
           schedule E
        end loop
    end loop
    For each bead B in the schedule
        B.state := Transition (B.state, L)
        if the new state is reducible then
            collapse B
        end if
    end loop
end loop
```

Figure 3.19: Fourth Improved Algorithm with Blockage Lists

Collapsing B using the FSA

If bead B goes into a reducible state, it is collapsed immediately. A final state represents some variant of a B^nAB, such as the degenerate aA, the singleton $aBAB$, or additional forward searches like $aBBBBAB$, its reverse $BABBBBa$, its inverse $Abbbbab$, or its reverse inverse $babbbbA$.

The degeneracy, aA, arises because this can be collapsed immediately rather than waiting for another state, such as B, as would be implied by literal restriction to the pattern $aB^0AB \rightarrow aAB \rightarrow B$. Special states are therefore added to the FSA for the cases bB, Bb, and similarly with the other fenced strands. The other final states are (cB^nCB, cD^nCD), their inverses (Cb^ncb, Cd^ncd), their reverses (BCB^nc, DCD^nc) which translate to (CDC^nd, CBC^nb), and their reverse inverses (cdc^nD, cbc^nB).

These all result in a stream of n (3 or more) beads collapsing to $n-2$ beads. It is important to note the inverse and reversal symmetries in the states, because it helps prevent errors in entering the FSA states into the computer, in writing test programs, and in simulating the states in cases where it is more efficient to store only a basis of the states, and have a meta-finite state automaton execute on the basis of states as if the whole set of states had been stored.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	1	2	3	4	5	6	7	8	9	10	11	12
Bead 3	b	U	a	V	B	W	B	X	A	Y	B	Z
Used by	0	0	0	0	4	4	6	6	8	8	10	not yet
State	-	-	b	b	bC	bC	bC	bC	bCB	bCB	bCBC	not yet

Figure 3.20: aB^nAB Arrivals

3.3.6 Collapsing B, using the Blockage Lists

The mechanism of the collapse ($aB^nAB \rightarrow BA^n$) could be theoretically implemented by removing the final A and B, then substituting $a \rightarrow B$, then changing the remaining $B^n - A^n$.

A sequence of blockage dependencies for X^n can each only depend on the next element. So the linked lists are unchanged in a B^nAB, except for the a, the final B in B^n, and the final A and B. Only these 4 linked lists need to be searched for other elements whose state might have changed.

If strands U, V, W, X, Y, and Z are separated from A and B and from each other by at least 2 strands, then aB^nAB might arrive as in Figure 3.20.

As B is accumulated onto the skein, bead 3 reaches a final state $bCBC$. The final 2 states BC (beads 9 and 11) are deleted along with their fences, and all dependent beads are scheduled for collapse. Then if there are any beads left (in this example, aBB), the first is inverted, and the rest are translated, making a $aBB - BAA$. Now this could have uncovered a prior (formerly impossible) collapse. For example, beads 1 and 3 now form a bB collapsible pair. To notice this, the algorithm follows the blockage list of a, which is bead 3, now B. Each element blocked by bead 3 (the list in this example contains only bead 1) is scheduled for having its state updated, which in this case triggers a further collapse.

It may be asked, why the FSA is permitted to access the skein database during a reduction. This extra memory access does not add to the time complexity level (for example, turning the FSA into a push-down automaton) of the FSA because of the following considerations.

- The FSA can be thought of as outputting a single word, the crossing which was reduced. This single word can be thought of as causing a separate process to do the following actions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

  - delete that crossing and its four fence crossings
  - to schedule the 10 crossings which precede or follow that crossing and its fences for further reductions

- The number of crossings in a braid diminishes with each reduction.
- Reexaminations are only scheduled upon accumulating a new input or a reduction.
- A crossing can only be reduced once.

However, if the data structure were unable to limit the transduction to outputting a word and required braid-long searches as part of the transduction, the complexity class would indeed be increased.

The Action Matrix

The 3 actions are (none, block, reduce). When a state transitions to a final state, the reduce action is called for. The actions for a reduce are:

1. Delete the last 2 beads. Recall that the act of deletion automatically deletes the fences and schedules all dependent beads to have their state reexamined.
2. If there are additional beads (that is, if the last two beads were anything more complicated than bB), reverse and translate them as necessary.
3. Repeatedly pick a bead scheduled for state update. Update its state based on the next node now on the strand. Deschedule it. If it results in a collapse, go to Step 1. If there are no more beads scheduled, then accumulate another bead. Otherwise, repeat this step.

Maybe the two extra linked lists for dependencies are not needed, because all dependencies are on the same strand which is already linked. That would simplify collapsing (just search the strand backwards for potential forward matches instead of updating 2 linked lists) at the expense of execution time for a linear search over all beads on that strand. One way to mitigate that is to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

note the double symmetry of the FSA (with respect to reversals and inverses). The FSA needs to be run backwards and forwards until there is a reduction or a block. The time wasted shows up in the worst-case example shown in Equation 3.12.

$$aBA^n(bB)^mB \tag{3.12}$$

where n is large, the entire reverse search over the BA^n must be repeated m times. On this braid (whose length is $3+n+2m$), the maximum number of searches $(2m+(n+2)m)$ will be at a point where m is approximately a quarter of the braid length, so that the bB pairs take up about half the braid. That maximum number of beads searched will be about one eighth the square of the length of the braid or $O(N^2)$.

To find this maximum, an approximate calculation is carried out as follows. Find the m that approximately minimizes N.

$$\begin{aligned} {}^{\#}S &= 2m+(n+2)m \approx 2m+nm \text{where} n = L-2m-3 \\ &= 2m+m(L-2m-3) \\ &= 2m+ml-2m^2-3m \\ &= m(L-1)-2m^2 \end{aligned} \tag{3.13}$$

Equation 3.13 is minimize when $L-1-4m=0$, which occurs when $\frac{L-1}{4}=m$ which is when $\frac{L}{4}\approx m$. At this value of m, $K=\frac{1}{8}(L^2-2L)$, which is approximately $\frac{1}{8}L^2$.

For example, for a braid of 1000 length, with $m=250$ pairs (bB) causing 250 re-searches of $N=1000-2(250)-3=497$ As and one B and one A, resulting in $2m+(N+2)m=500+(499)(250)=125,950$ re-searches. This is approximately $\frac{1000^2}{8}=125,000$. To eliminate this O(N^2) search, the linked lists are implemented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Level IV ($abAB$)

Level IV makes the fine distinction among those braids that require the $abAB$ type reduction versus those that can be solved using only $abaB$ type reductions, that is, semi-global versus global braids. The discussion of implementing this reduction is included above and just adds some extra reduce states to the FSA.

3.3.7 Conjugacy

In braid groups, all reductions eliminate two crossings. Two braids are not conjugate if one has an even length and the other an odd length. This observation solves the conjugate problem in 50 percent of all instances in a linear amount of time and in log space. However, the other 50 percent of the cases are more difficult.

Because of the categorization of conjugacy in terms of cosets, one could imagine a relationship between

- the equivalence relationship of the words in the cosets derived from a multiplication and
- the equivalence relationship of the words in a conjugacy class.

Most of the computations of conjugacy would immediately block, for example, speculating that two braids are **not** conjugate if they are of the following form: paq and pbq, where p and q are arbitrary braids (including the identity braid), and a and b are single crossings.

Unfortunately, a run of Braid Lab Software with the ends of the braids reflecting into each other (`linking=True`) came up with a braid that reduced to the following counterexample.

$$abb \sim ab\ abb\ BA$$
$$ababbBA \rightarrow ababA \rightarrow aabaA \rightarrow aab$$
$$abb \sim aab$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore we have what we shall call the "Horrible" Theorem that $abb \sim aab$. This makes conjugate determination difficult because there is no quick and easy way to know when it is time to terminate the search. Further investigation shows that at longer lengths, two braids still might be conjugate, even if they differ by only one crossing. This Horrible property effects both odd-numbered. even-numbered. long and short braids. The Braid Lab Software arrived at this counterexamples of length 4:

$abAa \rightarrow ab$

$abAb \rightarrow BbabAb \rightarrow BabaAb \rightarrow Babb \sim ab$

Therefore. $abAa \sim abAb$.

Conjugacy on the right ($Babb \sim ab$) can be checked in one of the following ways:

- Reflecting around the ends where the b and B "see" each other.
- Rotating the braid around. since all simple rotations are conjugate. For example. if a and b are simple crossings and x is an word. then the word axb is shifted one crossing to the right (end around) by conjugating $abx \sim A(axb)a \rightarrow xba$. The word axb is shifted one crossing to the left (end around) by conjugating $abx \sim b(axb)B \rightarrow baxbB \rightarrow bax$.
- Specifically conjugating $Babb \sim b(Babb)B \rightarrow ab$.
- Specifically conjugating $abAb \sim b(abAb)B \rightarrow babA \rightarrow abaA \rightarrow ab$.

In general. $ab^n \sim a^n b$. for all n (see Figure 3.21 at the end of this Section). So there is no choice, but to accept that differing by only one crossing does not categorize non-conjugacy at any length. This "Horrible" theorem stifled an application of the braid data structures developed for the word problem. If the horrible theorem had been false, and the initial intuition correct, then the braid word data structures would be directly useful in reducing conjugates, and in addition to all the existing blocking states, the work could stop immediately as soon as any words appeared of the form pbq that was different from paq.

This "horrible" theorem. has even worse consequences for efficiency, for it precludes any possibility

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of breaking a braid into smaller braids for the purpose of doing conjugacy reductions on the smaller pieces. Thus these data structures are not sufficient to solve the conjugacy problem by themselves.

There is, however, some good news, despite this theorem. The conjugate images of a braid have a relationship to the existing data structures, when the linked flag is set to reflect that the braid is to be treated as connected end to end. Each conjugate image of a braid is a combination of rotations, insertions of identity braids, and reductions. When the linked flag is set to True, the Level IV data structure can ignore rotations, because it can start anywhere in the "circle" of the braid's crossings. This data structure, finally, is a laboratory to play with the following conjecture.

Conjecture. The conjugate problem, whether there exists an a which makes $Axa = y$ for given braids x and y, can be solved by an algorithm on the linked skein data structure, by inserting y^I into the circle of x at the correct place and reducing it to ε.

The first experiment, Epstein's example, can be reducible this way. Epstein noted that not all conjugate braid images were simple rotations. For example, $bbccc \sim bbbcc$, a "horrible" example in Equation 3.14.

$$bbbcc \sim bc(bbbcc)CB \rightarrow cbcbbcB \rightarrow ccbcbcB \rightarrow ccbbcbB \rightarrow ccbbc \tag{3.14}$$

The first try at using this data structure would be to accumulate into the linked skein the braid $bbbcc$ and the braid $CBbc$ to get the conjugate image of the left braid, and then accumulated into the linked skein the braid $CCCBB$ which is the inverse of the right braid. This would reduce to $ccbbcCCCBB$, which reduces to $ccbbCCBB$, and no further, because it is not the identity. If $cc(CCCBB)CC \rightarrow CBBCC$ would have been added, however, it would have reduced to the identity. Unfortunately, taking the conjugate image of **both** sides has the undesirable side effect of implementing conjugates by conjugates, since adding a cc on the left and a CC on the right of $CCCBB$ is not the same as adding an identity. The left braid may be conjugated because it was being accumulated into a circle. Once the circle is populated, conjugating the right braid is equivalent to adding 3 braids, into the circle in this order: cc, $CCCBB$, and then CC, which is different from adding $CCCBB$ and an identity braids.

The second try will be to add identity braids which further conjugate the left braid before adding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the right. *ccbbc* must be conjugated by CC to get *bbccc*, the inverse of the right braid. There the following braids are accumulated into the linked skein.

1. the left braid *bbbcc*
2. the braid $CBccCCbc = \varepsilon$ (think of it as divided at its midpoint like $CC(bc(bbbcc)CB)cc$ in a circle)
3. the inverse of the right braid $CCCBB$, so the full circle would now be

$$CCbcbbbccCBccCCCBB$$

The reduction proceeds as follows.

1. $CCbcbbbccCBccCCCBB \rightarrow CCcbcbbccCBCBB \rightarrow CCcbcbbcBCBB$
2. $CCcbcbbcBCBB \rightarrow CbcbbcBCBB \rightarrow CcbcbcBCBB$
3. $CcbcbcBCBB \rightarrow bcbcBCBB \rightarrow bbcbBCBB$
4. $bbcbBCBB \rightarrow bbcCBB \rightarrow bbBB \rightarrow \varepsilon$

Is this an algorithm? No, guessing is required for the order in which to put things into the data structure and for which identity braids to insert, before reducing the circular identity.

Is this linked skein a useful data structure for exploring the braid conjugate problem? Yes, and it might eventually lead to an advancing algorithm for solving the conjugate problem. Such an algorithm would take a form something like the following. Compute certain metrics based on the form of the braids. Based on those metrics, insert a certain identity braid between them on the linked skein. If it reduces to the identity they are conjugate, else not.

Is there a problem with this algorithm? Not theoretically. However, practically, this research has not yet found a way to predict the identity braid needed to insert between the braids on the skein. Since the software is available on the Net, anyone may hack it and come up with an algorithm, but for now it is a tool for discovery, not an algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
& ab^n \sim a^n b \\
& (b)ab^n(B) = \\
& a^0 bab^{n-1} = \\
& a^1 bab^{n-2} = \\
& a^{n-1} bab^0 = \\
& a^{n-1} ba \sim \\
& (a)a^{n-1} ba(A) = \\
& a^n b
\end{aligned}
\tag{3.15}
$$

Figure 3.21: Proof of the Horrible Theorem

	a	b	c	d	e	A	B	C	D	E
c	-c	+cb	x	+cd	-c	-c	+cB	xx	+cD	-c
cb	x	+cbb	+cbc	x	-cb	x	$\boxed{PXX}$	+cbC	x	-cb
cd	-cd	x	+cdc	+cdd	x	-cd	x	+cdC	$\boxed{PXX}$	x
cB	x	$\boxed{PXX}$	x	x	-cB	x	x	+cBC	x	-cB
cD	-cD	x	x	$\boxed{PXX}$	x	-cD	x	+cDC	+cD	x
cbb	x	x	+cbc	-cbb	-cbb	x	$\boxed{PPXX}$	x	-cbb	-cbb
cbc	x	x	+cbc	-cbc	-cbc	x	$\boxed{XPPX_2}$	$\boxed{PPXX}$	-cbc	-cbc
cbC	x	x	$\boxed{PPXX}$	-cbC	-cbC	x	$\boxed{XMMX}$	x	-cbC	-cbC
cdc	-cdc	-cdc	+cdc	x	x	-cdc	-cdc	$\boxed{PPXX}$	$\boxed{XPPX_1}$	x
cdd	-cdd	-cdd	x	x	x	+cdd	+cdd	+cdC	$\boxed{PXX}$	x
cdC	-cdC	-cdC	$\boxed{PPXX}$	x	x	-cdC	-cdC	x	$\boxed{XMMX}$	x
cBC	x	x	$\boxed{PPXX}$	-cBC	-cBC	x	$\boxed{XPPX_2}$	x	-cBC	-cBC
cDC	-cDC	-cDC	$\boxed{PPXX}$	x	x	-c C	-cDC	x	$\boxed{XPPX_1}$	x

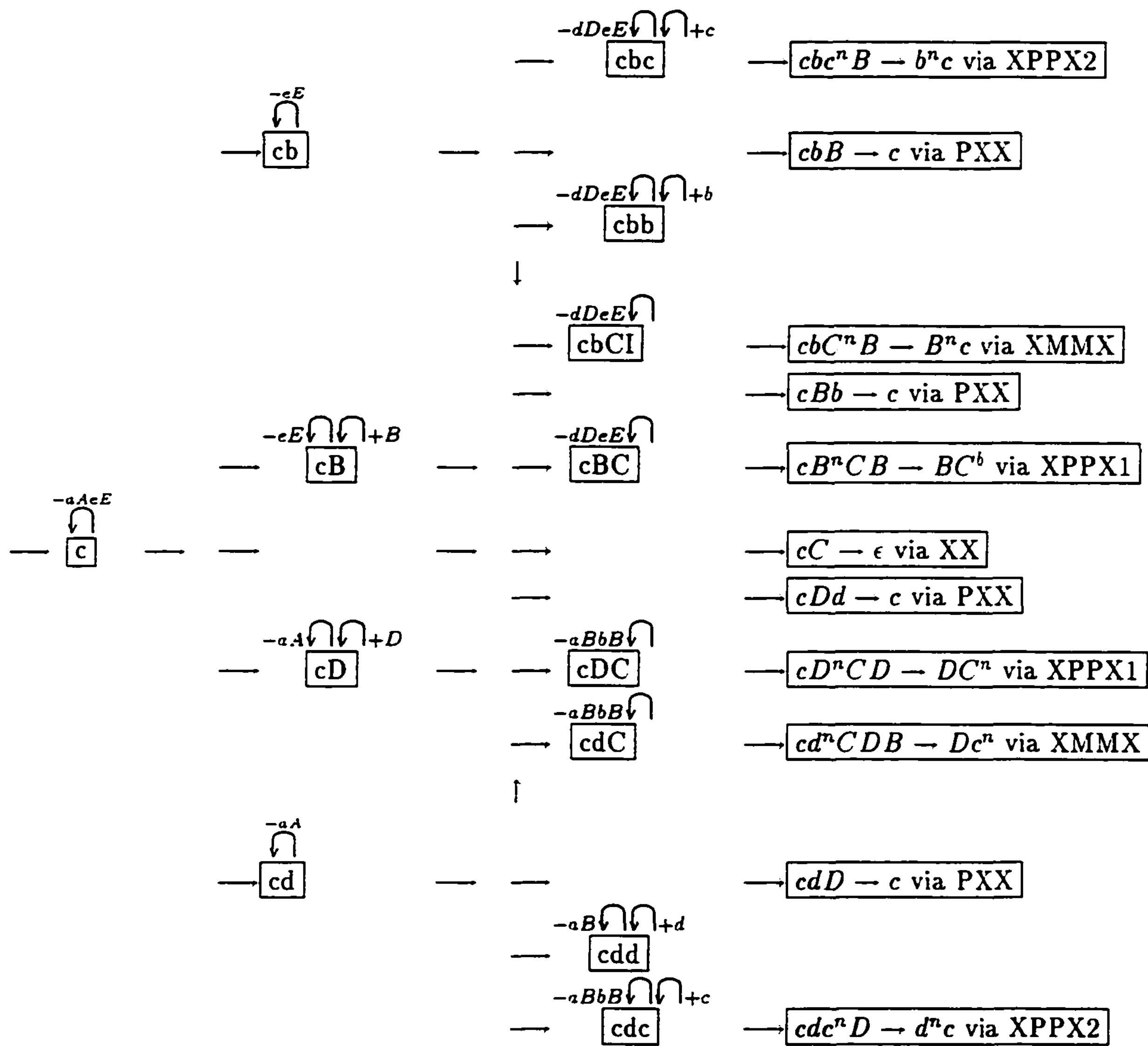
Table 3.9: Final FSA (Top Half Only)

3.3.8 FSA

The final FSA looks like Table 3.3.8, combined with mirror image states for C which is the inverse of c.

The top half of the FSA transition matrix displayed in Table 3.9. The symmetrical states for the C states, the inverse of c, constitute the bottom half. The transition codes (which could be implemented as a transition matrix combined with an action matrix, as a combined matrix, or directly in code) are explained in Table 3.10.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Action	**Non-Final States: change state and get next bead**
+state	Ignore the current bead during Recognition, Reduce during reduction
-state	Ignore the current bead during both Recognition and Reduction
$\boxed{PXX}$	Reduce patterns like $cBb \rightarrow c$ and keep going
$\times\times$	Reduce $cC \rightarrow \epsilon$ and keep going
$\boxed{PPXX}$	Reduce patterns like $abcC \rightarrow ab$
	Final States
$\times$	Block (stop immediately: no reduction possible)
$\boxed{XPPX_1}$	Reduce patterns like $cbc^nB \rightarrow b^nc$
$\boxed{XPPX_2}$	Reduce patterns like $cB^nCB \rightarrow BC^n$
$\boxed{XMMX}$	Reduce patterns like $cd^nCD \rightarrow Dc^n$

Table 3.10: Legend

Note that when Linking = False, the **home** code causes an automatic block, used to designate the end of the strand. When Linking = True, the **home** class is transparent, and a bead whose code is **home** causes the bead cursor to go one more bead and retrieve the Neighboring Bead. If the Neighboring Bead on the other side of the **home** bead is that same **home** bead, or if the Neighboring Bead is the same as the **start** bead, then **home** is returned and the search blocks immediately.

The FSA is sometimes run twice. First, each scheduled node that not already been reduce is run in Recognition Mode, to see whether it blocks. If it blocks, then no reduction is possible, but if not, then the FSA is run again in Reduction mode, using the code (such as $\boxed{XPPX_1}$) to guide the reduction process past the ambiguities. An example of an ambiguity is the pair of braids $cdAAAAAACD$ which reduces to $DAAAAAAc$ and $cdAAAAAAcD$ which reduces to $dAAAAAAc$, however, one does not know whether to change the d to a D or keep it d until after the unlimited sequence of A's is skipped. To avoid keeping a stack of pointers to the next bead before each sequence of skipped beads, just the reduction state is required. In this case, it is either $\boxed{XMMX}$ or $\boxed{XPPX_1}$. The plus ($\boxed{XPPX_1}$) keeps the d the same, and the minus ($\boxed{XMMX}$) inverts the d.

The Inductions for each of the reductions are given here.

$\boxed{XPPX_2}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$cbc^nB \rightarrow b^nc$ via $\boxed{XPPX_2}$ (symmetrical to cdc^nD)

b^0cbc^nB (start the induction)

$b^1cbc^{n-1}B$

$\vdots$

b^ncbc^0B

b^ncbB

b^nc

$\boxed{XPPX_1}$

$cB^nCB \rightarrow BC^n$ via $\boxed{XPPX_1}$ (symmetrical to cDC^nD)

cB^nCBC^0 (start the induction)

$cB^{n-1}CBC^1$

$\vdots$

cB^0CBC^n

$cCBC^n$

BC^n

$\boxed{XMMX}$

$cb^nCB \rightarrow Bc^n$ via $\boxed{XMMX}$ (symmetrical to $cd^nCD \rightarrow Dc^n$)

$Bbcb^nCB$ (add an identity, something undesirable while running the FSA, although it is okay to require it during the derivations)

Bc^0bcb^nCB (start the induction)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$Bc1bcbn - 1CB$

$Bc^n bcb^0 CB$

$Bc^n bcCB$

$Bc^n bB$

Bc^n

$\boxed{PPXX}$

$cbcC \rightarrow cb$ via $\boxed{PPXX}$

trivial derivation

$\boxed{PXX}$

$cbB \rightarrow c$ via $\boxed{PXX}$

trivial derivation

$\times\times$

$cC \rightarrow \epsilon$ via $\times\times$

trivial derivation

In reducing, it is easy to change, for example, d to D or vice versa, because just the codes get changed in existing beads that are thoroughly linked into the skein on five consecutive strands, but at widely varying bead locations along those 5 strands. However, at first glance, it seems to require an extra search of as much as $(N - 2)$ beads to find the spot on a given strand to change a bead to. For example, consider the bead $cb^{1,000,000}CB \rightarrow Bc^n$. To change the last $(n - 1)$ instances of crossing b to c, the fences of each of these beads must change. Previously these fences did not previously block strand e, but now they must block (insert a fence bead) strand e. If the data structure did not keep the state to guide the reduction, a search would be necessary for the next previous bead on strand e **prior to the current bead** to find where to insert the fence on e.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is a double linked list along strand e, but it is not indexed into the relative positions of each of the other strands. To do so would require a quadratic amount of storage (cross-referencing each bead on each strand to each bead on each other strand), which is undesirable. Therefore, a single word of storage (the FSA state) is used to guide the reduction without requiring any other cross reference information. The same problem seems to arise, if this is handled by inserting new beads, rather than changing existing beads. The extreme fences (a, A, e, and E) seem to require a search of the new strand. Since that would require quadratic space, it is would be easier to rebuild the entire skein for these cases, since that takes only linear time. However, there is a solution that keeps the skein intact. In all of the possible reductions, the extreme fence to be inserted is already present in either the first or second character that was condensed (had a + in the transition table) as part of the reduction. Thus, there is a place to put the extra fence at that horizontal (bead) position along the braid, without any searching at all. In conclusion then, the only memory required is the reduction state the FSA terminated in during the Recognition Phase.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Braid Complexity Metrics

4.1 The Metrics

Previously in this paper, a braid's coupling was discussed by counting the levels of collapse of its Cayley graph required to reduce it to the identity. This will now be enhanced to define a braid's cohesion and its complexification coupling in terms of the depth of the data structures required to solve it, and measures of the complexity of the crossings of the strands.

4.1.1 Heavy Cohesion and Cabling

Since the process of reading in the braid can detect all vertical and horizontal factors in linear time (advancing algorithm), one might hope to use this as input to parallel algorithms in linear space. However, the braid characteristics that would prevent an advancing one-pass algorithm would be the local semi-non-commutativity (the fences) which would be the same characteristic preventing independence between the parallel tasks, unless the braid were completely factored. A few quick experiments with Christmas tree lights leads to experiments to factor some random braids. A run of 1000 10-strand braids with 1000 crossings spun up no horizontally or vertically factorable braids. A similar run of maximally-knotted identity braids also spun up no horizontally or vertically factored braids, although in the process of reducing to the identity, they would lose cohesion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Example of the PR-Braid

A large example of full (heaviest) cabling occurs in [15] in the counterexample used in the proof that the set of minimal braids isotopic to a given braid is co-NP complete, if the number of crossings and the number of strands are permitted to vary. This counterexample used a single strand to build up three levels of cables of cables of cables, and then crossed them all with that single strand. Using these PR-counterexample braids as the initial random braids on which to build random identity braids did not result in any improvements required to the data structures used in this paper, despite the rapid, $O(N^5+\epsilon)$, growth in the size of the PR-braid with the number of strands per cable, and the complexity of each braid.

The ϵ is mentioned because at low number of strands per cable, N, below about 100, well within the area of applicability for practical cryptographic systems, the N^4 component of $2N^5+4N^4$ contributes a significant number (more than 1%) of the crossings to the PR-braid. This can be derived from the following PDL which simulates the number of crossings in the PR-braid in Figure 4.1.

Estimating the size of the PR-braid with n strands at each level of cabling can be done by calling

```
estimate (n, n, n, n, n, n);
```

This would result in the computation shown in Figures 4.2 and 4.1.

4.1.2 Braid Coupling

The coupling of a word in a group can be measured by the number of levels of mega-collapse, that is, the complexity of the data structure it requires to be recognized. The coupling of a word in a braid group can be measured similarly by the number of levels of reduction, that is the complexity of the data structure it requires to decipher its word problem (or, alternatively, its conjugate problem). For example, the coupling of the braids used as counterexamples to each data structure in Section 2 of this paper can be defined as the level of their data structure, as described in Table 4.2.

The higher order effects, which have not yet been observed in this research, would be an interference

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
function estimate (strands_per_cable: integer;
                   windings_around_each_cable: integer;
                   number_of_cables: integer;
                   repetitions_of_windings_per_cycle: integer;
                   hermetic_windings_per_cycle: integer;
                   cycles: natural) return integer;
begin
  -- The strands IN THE CABLES are the strands wound
  -- in the lowest level of cables.

  strands_in_cables := number_of_cables * strands_per_cable;

  -- Hermetic windings go around all cables at the next level.

  hermetics := hermetic_windings_per_cycle *
               strands_in_cables * 2;

  cable_windings := repetitions_of_windings_per_cycle *
                    strands_in_cables *
                    (2 * windings_around_each_cable + 2);

  return cycles * (hermetics + cable_windings);
end estimate;
```

Figure 4.1: Function Estimate

$SIC = n^2$
$H = 2n^3$
$cw = n * n^2 * (2 * n + 2)$
$x1 = n * (h + cw)$
$x2 = n * (2n^3 + n^3 * (2n + 2))$
$x3 = n^4 * (2 + 2n + 2)$
$x4 = 2n^4(2 + n)$
$x5 = 4n^4 + 2n^5$
return $x5$

Figure 4.2: Estimate of Growth of PR-Braid

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Strands Per Cable	Crossings in PR-Braid
0	0
1	6
2	128
3	810
4	3.072
5	8,750
6	20,736
7	43.218
8	81.920
9	144,342
10	240.000
11	380.666
12	580.608
100	1.04×10^{10}

Table 4.1: Growth of PR-Braid

Level	Sample Braid at This Level
bB	0
$(bB)^{100}$	0
$b^1 00B^{100}$	1
$abcCBA$	1
$abaB$	2
$abAB \rightarrow Ba$	3
aB^nAB	4
$aBA^n(bB)^mB$	4
higher order effects (not yet observed)	5

Table 4.2: Counterexamples

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

across multiple strands, such a cascade of fences, such as the crossing sequence *abcdefghijklm* interfering with a reduction involving *n*. Because of the look-ahead of the linked lists. all of the identity braids with cascading fences attempted with these data structures have been solved without taking the cascading into consideration, other than as the source of the required fences. This can mean any of several things.

- Possibly, there are no multiple-strand effects which prevent the FSA from solving the braid word problem, and the final FSA is powerful enough to solve the braid word problem.
- Possibly, there are multiple ($>$ 5-strand) effects which the random identity braid generator just did not happen to generate yet, for which the final FSA is not powerful enough to solve the braid word problem. One of the effects mitigating against 5-strand effects is the following interesting reduction.

  abcdedcba → *abcedecba* → *eabcdcbae* → *eabdcdbae* → *edacbcade* → *edcbabcde*

  This visually pleasing sequence inverts itself (Figure 4.3), yet remains equivalent.

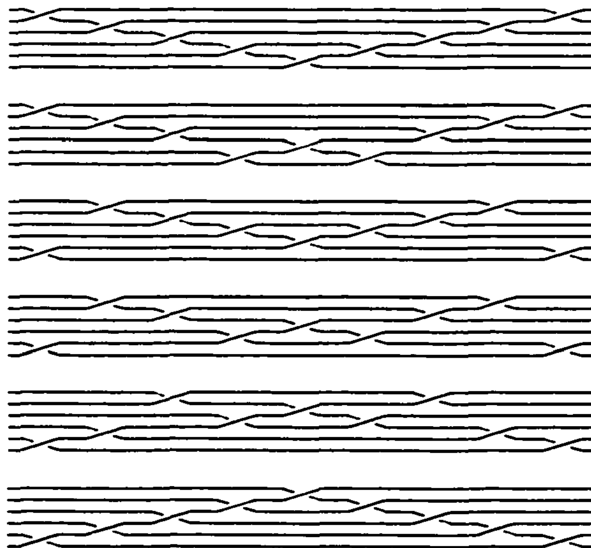


Figure 4.3: Mitigating Sequence

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Braid Name	**Braid**	Cohesion
Non-Identity Reduce	bcbC	1
yxiyixi Failure	BCbbDdBBCAaCCABbcb	2
Requires Respecting Fence	BcAbcbBCBaCb	2
Reliable	(-27 26 28 27 28 -28 -28 -28 -26 27)	2
PR-12 Need Virtual Memory	Patterson-Razborov Counterexample	2
NS=5 NK=48 Stack Dump	BADcBccdcabCDCCdcBACDbCdab	2
NS=100 NK=56 aB^nAB	dbCDDccdccdCDBDD	3

Table 4.3: Partial List of Braid Counterexamples

4.1.3 Braid Counterexamples

During the history of developing and maintaining the Braid Lab Software the braids in Table 4.3 caused jumps to be programming in the capability of the software. These jumps sometimes took the form of IF statements or loops, but more often took the form of changes in the skein architecture to accommodate the newly generated braid.

The last braid with a hundred strands looks like Figure 4.4 on top and has 94 extra strands beneath it with no crossings.

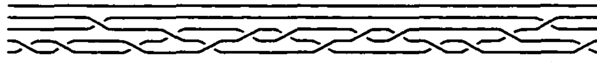


Figure 4.4: Braid dbCDDccdccdCDBDD

4.1.4 Group Counterexamples

Unlike the braid counterexamples which were spun up by machine, except the PR-braids which were seeded into the program, the group counterexamples in Table 4.4 are all taken from the group theory books and articles in [**16-21**].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Group/Word Name	Test	Cohesion Level
non-identity equality	AAAc=AAAc	1
with identity equality	$\epsilon CC\epsilon$	1
relator AAA	AAA=ϵ	1
cyclic	relator abAB implies abb=bab	1
commuter	relators ababb and babaa imply ab=ba	2
mega-collapse identity	relators ab and a imply abba=ε	3
ordering aAbBcCdD	random 2-gen 2-rel arb word	3
edge collapse	abaaaaaa, baaa imply baaabaaa=ε	4
Dyck(2.3.3)	bABa=ε implies bbbAABBBa^{5}4 = ε	4

Table 4.4: Partial List of Group Counterexamples

Note that more complicated data structures involving more node and edge collapses may be required for relatively **simpler** word problems that collapse to the identity in ways that are may be obvious to a human. Remember, these problems are being solve exclusively by building a Cayley graph out one level at a time until an identity is found. Other strategies could have made these examples quicker. but the intention of this research was to use the collapsing Cayley graphs as the **sole** driver of complexity in the data structures used to solve the word problem.

The last example the table, an experiment with a Dyck group, as set up in [**22**], is the group $< a, b, c \mid a^P, b^Q, (ab)^R >$. since two Dyck groups. Dyck(P.Q.R) and Dyck(R.Q.P) are isomorphic.

In this example the isomorphism test was run with parameters P=2. Q=3. and R=3. First the algorithm proved insufficient. then. after fixing the algorithm. the PC memory proved insufficient. and virtual memory was implemented. The Dyck group with parameters 2, 3, 3 required 60.485 nodes, with 62.495 adds and 1.945 mega-collapses. Groups are the essence of intertwinedness, so there is no quick way to predict the coupling, cohesion, number of nodes required to build the Cayley graph, number of adds, number of collapses, or number of mega-collapses, even for those groups that are solvable. In the general case, there is no way to guarantee solvability or even to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

predict whether a given word is solvable.

Historical

The following test code reflects the history of the group code development before it was realized how pervasive the word problem is in every method that was attempted for implementation. At that point, the decision was made to concentrate on braid groups.

```
-- The following worked with a single level of collapse:

((a+b)&(+(a*b*(-a)*(-b))),
       a*b*b, b*a*b, "cyclulative",4)
((a+b)&(a*a*a & a*b*(-a)*(-b)),
       b*b,     a*b*a*b*a, "T2")
((a+b)&(+(a*b*(-a)*(-b))),
       a*b*b*b, b*b*b*a, "T1")
((+a)&(+(a*a*a)),
       a*a*a*a*a*a,   +a, "T3")
((+a)&(+(a*a*a)),
       a*a*a*a*a,     +e, "triangle")
((a+b)&((a*a*a)&(b*b)),
       b*b*b*b*b*b,   +e, "twog")
((a+b)&((a*a*a)&(b*b)),
       b*b*b*a*a*a*b*b*b,   +e, "twog part 2")
((a+b)&(+(b*a*b)),
       b*a*b*b*a*b,   +e, "presix")
((a+b)&((b*a*b)&(a*a*a)),
       b*a*b*b*a*b,   +e, "sixer")
((a+b)&(+(a*b)),
       a*b*a*a*a*a*b*b*b, +a,"Line")
((a+b)&(+(a*a*b)),
       a*a, -b,"Doublet")
((a+b)&(+(a*a*b*a*b*b*b)),
       a*a*b*a*b*b*b,+e,"Half of seven")
((a+b)&(+(b*b*a*b*a*a*a)),
       b*b*a*b*a*a*a,+e,"Other half of seven")
((a+b)&((a*a*b*a)&(b*b*a*b)),
       b*b*a*b*a*a*b*a,   +e, "parts of 7")
((a+b)&((a*b)&(b*a*a*a)),
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
       b*a*a*a*b*a*a*a,   +e, "easy one")
((a+b)&((a*a*(-b)*(-b))&(a*b*a*(-b))),
       a*a*a*a,   +e, "a4")

-- The following required mega-collapse:

((a+b+c+d)&((a*b*(-c))&(b*c*(-d))&(c*d*(-a))&(d*a*(-b))),
       +c, a*b, "F(2,4)-1",5)
((a+b)&((a*a*b*a*b)&(b*b*a*b*a)),
       b*b*a*b*a,   +e, "tester2",5)
((a+b)&((a*a*b*a*b*b*b)&(b*b*a*b*a*a*a)),
       a*a*b*a*b*b*b, +e,"B1")
((a+b)&((a*a*b*a*b*b*b)&(b*b*a*b*a*a*a)),
       a*a*a*a*a*a*a, +e,"B2")
((a+b+c+d)&((a*c*(-a)*(-c))&(a*d*(-a)*(-d))&(b*c*(-b)*(-c))&
   (b*d*(-b)*(-d))),
       a*b*c*d,c*d*a*b, "quadder")
((a+b)&((a*a*b*(-a)*(-b)*(-b))&(a*a*a*a*a*a*a*a)),
       b*b*b*b*b*b*b*b, +e, "b8")
((a+b)&((a*b*a*b*b)&(b*a*b*a*a)),
       a*b, b*a, "commut",3)
((a+b)&((a*b*a*b*b)&(b*a*b*a*a)),
       a*a*a*a*a, +e, "a5",3)
((a+b)&((a*a*b*a*b*b)&(b*b*a*b*a*a)),
       b*b*a*b*a*a,   +e, "tester3",4)
((a+b+c+d)&((a*b*(-c))&(b*c*(-d))&(c*d*(-a))&(d*a*(-b))),
       +d, b*c, "F(2,4)-2",4)
((a+b+c+d)&((a*b*(-c))&(b*c*(-d))&(c*d*(-a))&(d*a*(-b))),
       +d, b*a*b, "F(2,4)-3",4)
((a+b+c+d)&((a*b*(-c))&(b*c*(-d))&(c*d*(-a))&(d*a*(-b))),
       +a, (-b)*(-b)*(-b), "F(2,4)-4",4)
((a+b+c+d)&((a*b*(-c))&(b*c*(-d))&(c*d*(-a))&(d*a*(-b))),
       b*b*b*b*b, +e, "F(2,4)-5",6)
((a+b)&((a*a*b*a*b*b*b)&(b*b*a*b*a*a*a)),
       b*b*a*b*a*a*a, +e, "tester4",9)
((a+b)&((a*b*a)&(b*a*a)),
       b*a*a,   +e, "testee2",6)
((a+b)&(+(b*a*a*a)),
       b*a*a*a,   +e, "testee3",6)

-- The following required identity-collapsing
-- in addition to mega-collapse
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
((a+b)&(a*b & (+a)),
      +b, +e, "Identity")
((a+b)&(a*b & (+a)),
      +b,      +e, "Id1")
((a+b)&(a*b & (+a)),
      +a,      +e, "Id2")
((a+b)&(a*b & (+a)),
      a*b*b*a, +e, "Id3")
((a+b)&((a*b)&(b*a*a)),
      b*a*a,   +e, "teste1",6)
((a+b)&((a*b*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "idnoteq",6)
((a+b)&((a*b*a*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "myniece",6)
((a+b)&((a*b*a*a*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "test10",12)

-- The following required schedule collapse (shuffles):

((a+b)&((a*b*a*a*a*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "test11",12)
((a+b)&((a*b*a*a*a*a*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "test12",12)
((a+b)&((a*b*a*a*a*a*a*a)&(b*a*a*a)),
      b*a*a*a*b*a*a*a,   +e, "test13",12)
```

4.2 Software Maintenance

The portion of software maintenance expense that pays for analysis of the impact of change (a small, yet critical portion of the total software maintenance budget) is driven by the system's cohesion and coupling.

Software maintenance estimation is more difficult than estimating software development because

- There are significant differences between development and maintenance
  - Maintainers need the preconditions and invariants left out by developers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Maintenance uses far more testing than development.
- Maintenance time is not correlated to SLOC.
- Multiple exits from loops and error exits are denigrated by structured programming enthusiasts who consequently eliminate from consideration useful error exit and error recovery strategies that would otherwise have been programmed.
- The second most time-consuming maintenance activity (after testing) is immeasurably small at development time - analyzing the impact of change.
- All complexity measures tend to be correlated to each other at maintenance time, and not highly correlated to cost (time).
- Most bugs are inserted by the maintenance programmers.
- Requirements creep is continuous at maintenance time with no respite for development releases.

- Projects tend to measure maintenance by the same easy-to-measure metrics they use to measure development by.
  - Idealized process descriptions which do not accurately reflect the processes they actually carry out.
  - SLOC
  - Structured programming (multiple exits)
  - Number of faces on the program flow chart
  - Number of nodes or edges on the data flow chart
- Projects do not tend to measure the hard-to-measure metrics.
  - Cost of opening a configuration (finding the code and tools, regenerating the documentation (understanding the code), etc.)
  - Requirements creep
  - Bugs inserted by programmers
  - Visibility of Global Variables, directly or aliased, or homonyms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

%	Br Gr	e.g.	Name	Local	SemiG	Global
5%	Type I	aB	Purely=No State	0	0	0
85%	Type II	$aBAB$	Local State	100	0	0
5%	Type III	$abAB$	Semi Global State	95	5	0
5%	Type IV	aBA^nB	Global State	90	$7\frac{1}{2}$	$2\frac{1}{2}$

Table 4.5: Braid Group Types

4.2.1 Coupling

There is an analogy between the locality of the braid reduction process and the locality of reference of variables in computer programs during software maintenance activities. The last metric, visibility of global variables, can be measured by analogy to the visibility of global reductions in groups. Certain braid groups tend to be distributed in nature (that is, show up randomly) accordingly to the matrix in Table 4.5, highly rounded.

This observation neither proves anything about braids nor about software maintenance. In particular, alone, it does not prove that developers should tend towards software whose data flow emulates that of the data structure required to solve simple groups, nor does it prove that maintainers will necessarily maintain simpler code cheaper.

However, it does present a useable natural classification for software source code complexity based on visibility of global variables. If a project uses this classification, the psychological law is invoked that states metrics start to improve as soon as they are posted on the manager's door.

In this discussion the phrase "global variable" means either an ordinary global variable or any kind of external interface, such as a port, an absolute memory address, an I/O operation or a connection to anything outside of the language the source code is written in.

To use this metric, the following counts are made.

- The count of the references to each global variable in each software component. The number of visible global variables in each component of the software, as well as semi global and global. Each homonym of a global variable counts as a global variale. Each mutable pointer counts

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as a global variable, even if it points to a local variable. Each alias of a global variable counts as a global variable, except that when a global variable is passed once as a parameter to a given component, it is promoted to being a local variable. Static variables whose scope is hidden count as semi global. Machine registers, absolute memory addresse. variables visible to more than one processor, and input-output ports all count as global variables.

- The count of the references to each semi-global variable in each software component. Each homonym of a semi-global variable and each alias of a semi-global variable (with the same exception as above), counts as a semi-global variable. Any variable dynamically allocated from a heap, which would otherwise have been considered to be a local variable. is counted as a semi-global variable. because the memory leakage is managed from a higher level than the component it is in. Interrupts are considered to be global variables. even if the handler is hidden locally.
- The count of the references to each local variable in each software component.
- The count of components containing one or more global variable references.
- The count of those components not containing no global variable references. but containing semi-global variable references.
- The count of those components not containing global or semi-global references, but containing local variable references.
- The count of those components that have no state at all (pure functions or pure packages). In most computer programming languages these correspond exactly to those components that do not have any variables which remain after a call. Languages which have ways of specifying machine registers. absolute addresses, methods of importing and exporting data to assembler code, methods of handling interrupts, parallel tasks, interprocessor calls, etc., may have ways of hiding the fact that a component contains state beyond what is represented by the variables in the component. In those languages, manual analysis might be required to ascertain whether a component has state.

Use the Metric Collection Form in Figure 4.6, replacing nnn with the numbers counted. If the system has no state there will be no visibilities, it is a pure system, data flow locality of reference

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	Components	#Sets	#Uses
# Pure	nnn	0	nnn
# Local	nnn	nnn	nnn
# Semi Global	nnn	nnn	nnn
# Global	nnn	nnn	nnn

Table 4.6: Metrics Collection Form

level zero. Otherwise it is an ordinary system. The percentage of local variable visibilities data flow locality of reference level (DFLRL).

Future Research

The next step in this research is

- measure the data flow locality of reference level metric on software maintenance projects for cost correlation.

4.2.2 Is there a Positive Return on Investment

The Project Return on Investment (ROI) an on-going organization which expends resources on that project is defined as the reduction in organizational costs achieved by the project, after taking into consideration all of the resources expended on the project (immediate and annual) and the decrease in the organization overhead and production annual costs (before and after the project). To engage in the project, the organization must be convinced that the ROI is not only positive (when computed according to the net current value of the savings to be received), and but that it is sufficiently positive to cover the risk of failure to deliver those savings.

Measuring the effects of the organization's productivity and overhead requires surveying and measuring the amount of time the people in the organization spend on productive and non-productive activities, defined as follows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Productive Activities (Tooth)
  - analyzing requirements from customer service requests
  - designing solutions
  - analyzing the impact of change
  - implementing changes
  - testing
- Non-Productive Activities (Tail)
  - managing facility, human, and electronic resources
  - managing the configuration of the baseline products (tracking the parts, changes to parts, and change requests)
  - building, packaging, delivering, and installing the baseline products comprising those parts
  - attending meetings of any kind in person or via e-mail, video teleconferencing, etc.
  - assuring quality
  - training and being trained
  - installing and learning how to use tools
  - analyzing and improving the organization's software maintenance processes
  - waiting for computer response on Net searches, electronic mail, database queries, tool searches, compiles, builds, tests, etc.
  - performing key and mouse strokes required because the maintainer's computer screen was not large enough to hold high resolution images of two full pages of information next to each other for comparison
  - any other automatable activities

It is necessary examine whether a positive ROI will be achieved by re-engineering a software system to have less coupling, that is, fewer instances of visibilities of global variables.

- Facts tending towards sufficient reduced costs to achieve a positive ROI

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Pure or local software is cheaper to analyze than semi-global or global. because there are no coupled semi-global and global relationships to check.
- Completely semi-global software is cheaper to analyze because it does not propagate bugs as far as global or interface data will. In particular, analysis of its bugs are limited to the object and its packagings, permitting the project to be divided for analysis purposes.
- Software with visibility of global variables is more expensive to analyze than the same software re-designed to have no visibility of global variables.

- Facts tending towards less reduction of costs

  At the higher levels of globality. current research is not sufficient to conclude that there will be a positive return on investment from making small changes in the software to lower the number of global references a little. Even though the simplified software will be faster to analyze, the overall cost of maintaining the system may remain the same because of the following.

  - Analysis costs are dwarfed by sunk investments like facility, configuration, tool license, fixed personnel and overhead costs, and will not show up if they are accounted for as part of these sunk costs.
  - The majority of the operational activities concentrate on testing. and the reduction in testing is only a secondary consequence of introducing fewer bugs. If the software maintenance facility has a backlog of customer service requests. unpaid overtime (never to be paid). analysis not done. testing deferred or not done, and schedule slips, then this backlog will eat up any saving in testing time. Until the backlog is gone. the savings from introducing locality of reference will not show up in the metrics. However, introducing locality of reference can have a measurable effect on these backlogs, which itself is a reduction in costs.
  - Typical maintenance facility do not measure separately the areas that are most affected by global visibility: costs of analyzing the impact of change, requirements creep, repeat fixes, split baselines, and bugs inserted by maintainers. Until they are measured, global visibility's cost impact will be invisible, though hefty.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To see this currently invisible effect, a project could

- publish global visibility metrics
- publish backlog metrics
  - backlog of customer service requests
  - cancelled customer service requests
  - customer service requests closed by the software maintenance facility without the customer's signature
  - unpaid overtime (never to be paid)
  - analysis not done
  - testing deferred
  - regression tests, stress tests, system tests not run
  - schedule slips
  - the organization's tooth-to-tail ratio
- independently publish metrics tallying the results of analyzing the following.
  - the effect of requirements creep
  - repeat bugs
  - bugs inserted by maintainers
- demand that 15 percent of all software maintenance time goes to reengineering such as the following.
  - reduce coupling accomplished through global variables
  - package methods into cohesive objects
  - reduce the usage of complex methods of coupling such as explicit pointers and multiple inheritance
  - eliminate comments in the code that are not true at the point in the code where they appear

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- add comments into the code that are true at that point in the code (preconditions, postconditions, and other invariants)
- measure the quarterly reduction in backlog in increase in the tooth-to-tail ratio of the organization achieved.

The data flow is the glue that relates the objects and algorithms. The metrics on the data flow of braid algorithms indicate that:

- The Braid Word Problem is too fast to solve on the average, to be useful in cryptographic encryptions.
- The Braid Conjugate Problem requires more research to determine how slow it is for the kinds of braids that would be used in cryptography.
- There is a strong analogy between braids and software maintenance that needs to be pursued measuring by the braid metrics given here.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Conclusion

Certain data structures reduce the time complexity of certain braid group algorithms, possibly as low as quadratic [**23**], at the expense of using a linear amount of space instead of a logarithmic amount of space.

Carefully, yet randomly generating some particularly twisted identity braid groups resulted in the discovery of counterexamples that can be used to test other group and braid algorithms. The classes of counterexamples were based on the levels of collapse in the Cayley graph, called levels of deception, because each additional level requires more complex software to decode it. These counterexamples imply that the braid word problem is too easy to be useful naively in cryptography [**24**]. However, additional research will be required on the conjugacy classes and the non-minimal braid representations in this area.

The cohesion and coupling of an identity braid determines the difficulty of reducing it. This difficulty is analogous to the difficulty of one particular activity during of ordinary software maintenance processes, namely the activity of analyzing the impact of change. This analogy covers: (a) order of code and order of fix: (b) metrics estimating the cost of analysis of the impact of change based on the cohesion and coupling of the source code; (c) correlation between cohesion and coupling on the one hand, and less predictive but easier to measure complexity metrics on the other hand; and (d) seeing the analysis of the impact of change as a permutation of the data flow diagram and program flow diagram.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Future research extending the skein structure presented here to solving the conjugacy problem in braids is outlined, as well as possible extensions to an expert system to automatically prove certain braid group theorems and to semi-automate the search for better braid group algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

References

1. J. T. Schwartz. *On Programming* (1974). Courant Institute of Mathematical Sciences. NYU.

2. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 195 and 50. Jones and Bartlett.

3. W. Magnus, A. Karrass. and D. Solitar. *Combinatorial Group Theory* (1977). Springer-Verlag.

4. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 183. Jones and Bartlett.

5. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 192-193. Jones and Bartlett.

6. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 198. Jones and Bartlett.

7. M. S. Patterson, and A. A. Razborov. The Set of Minimal Braids is Co-NP-Complete in *J Algor 12* (1991) 393-408. Academic Press.

8. M. S. Patterson, and A. A. Razborov. The Set of Minimal Braids is Co-NP-Complete in *J Algor 12* (1991) 394. Academic Press.

9. G. Baumslag, C. F. Miller III, and H. Short. Unsolvable Problems About Small Cancellation and Word Hyperbolic Groups. In *Bull Lon Math Soc Volume 26* (1995) 118.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

10. M. S. Patterson, and A. A. Razborov. The Set of Minimal Braids is Co-NP-Complete in *J Algor 12* (1991) 395. Academic Press.

11. H. Tietze. Über die Topologischen Invarienten mehrdimensionaler Mannigfaltigkeiten. in *Monatsh Math Phys* 19 (1908).

12. D. L. Johnson. *Presentation of Groups* in Lon Math Soc 15 (1900) 49. Cambridge Univ Press.

13. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 198. Jones and Bartlett.

14. G. Marsaglia, B. Narasimhan, and A. Zaman. A Random number generator for PCs in *Computer Physics Communications* 60 (Oct 1990) 345-349.

15. M. S. Patterson, and A. A. Razborov. The Set of Minimal Braids is Co-NP-Complete in *J Algor* 12 (1991) 397. Academic Press.

16. G. Baumslag, C. F. Miller III, and H. Short. Unsolvable Problems About Small Cancellation and Word Hyperbolic Groups in *Bull Lon Math Soc* volume 26 (1995) 97.

17. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups* (1992) 181-209. Jones and Bartlett.

18. M. Jantzen. *EATCS Monographs 14: Confluent String Rewriting* (1988). Springer-Verlag.

19. D. L. Johnson. Presentation of Groups in Lon Math Soc 15 (1990). Cambridge Univ Press.

20. W. Magnus, A. Karrass, and D. Solitar. *Combinatorial Group Theory* (1977). Springer-Verlag.

21. M. S. Patterson, and A. A. Razborov. The Set of Minimal Braids is Co-NP-Complete in *J Algor* 12 (1991) 397. Academic Press.

22. D. L. Johnson. Presentation of Groups in Lon Math Soc 15 (1990) 47-48. Cambridge Univ Press.

23. C. Papadimitriou. *Computational Complexity* (1994). Addison-Wesley.

24. B. Schneier. *Applied Cryptography* (1996). John Wiley.

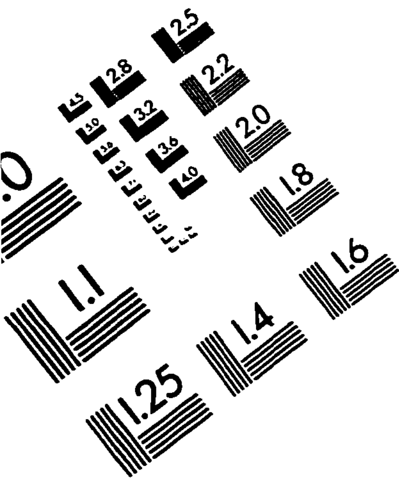
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

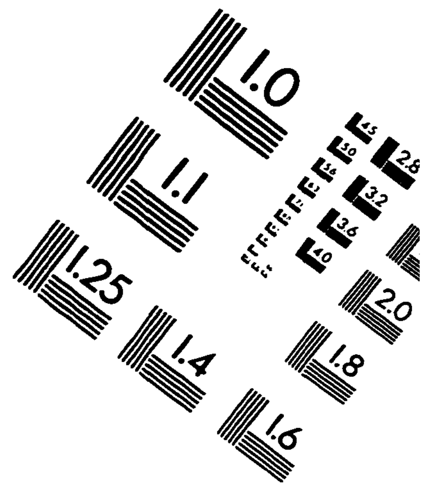
25. P. Dehornoy. A Fast Method for Comparing Braids in *Advances in Mathematics* 125 (1997) 200-233. Academic Press.

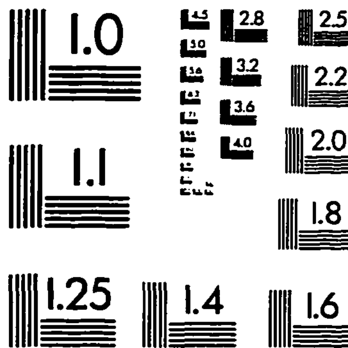
26. J. Hass, J. Lagarias, N. Pippinger, The Computational Complexity of Knot and Link Problems (preprint obtained from author J. Lagarias at jclresearch.att.com).

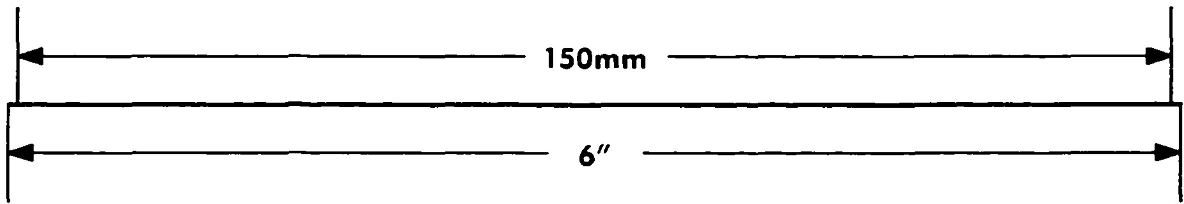
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

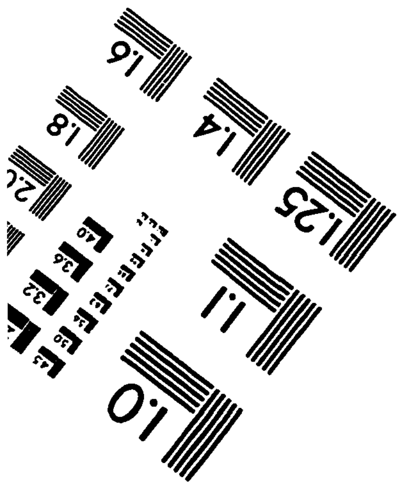
IMAGE EVALUATION TEST TARGET (QA-3)

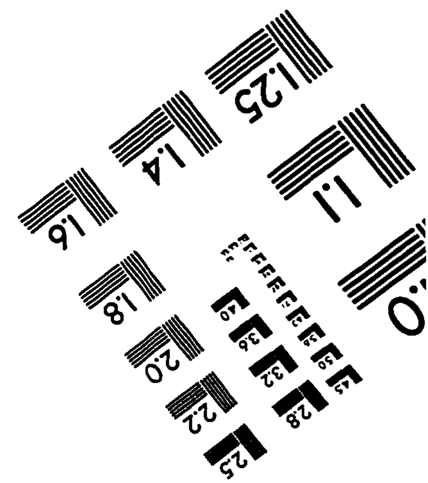












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.