# A PACKAGE OF ALGORITHMS FOR SOLVING HARD PROBLEMS IN GROUP THEORY

by

DMITRY BORMOTOV

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

2005

UMI Number: 3187464

UMI Microform 3187464

Copyright 2005 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

©2005

DMITRY BORMOTOV

All Rights Reserved

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.

05/06/06
Date

Alexei Miasnikov
Chair of Examining Committee

05/06/06
Date

Ted Brown
Executive Officer

Michael Anshel

Robert Gilman

Sergei Artemov

Vladimir Shpilrain
Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

# Abstract

A PACKAGE OF ALGORITHMS FOR SOLVING HARD PROBLEMS IN GROUP THEORY

by

Dmitry Bormotov

Advisor: Alexei Miasnikov

In this thesis we developed some methods of a new emerging area of mathematics, which we call experimental algebra. We designed new algorithms, both deterministic and heuristic, aimed at hard problems in algebra. Using these algorithms we solved some open problems in groups and semigroups. These algorithms are implemented and organized in a package. In addition to implementations of stand-alone algorithms for solving particular problems, the package contains a variety of reusable components - tools that can be used to modify the behavior of existing algorithms, to form alternative strategies and to build new algorithms out of provided components.

# Acknowledgments

I am grateful to my adviser Alexei Miasnikov for initiating and guiding this research, for providing ideas, support and inspiration without which this work would not be possible.

I would like to thank Michelle Baker, Ted Brown, Gilbert Baumslag and Alexei Miasnikov for providing financial support.

I wish to thank Kent Boklan for initiating the research on key hiding as well constantly challenging it with problems, ideas and concerns, which resulted in much better developed and efficient system. I would also like to thank Sean Zhang for integrating the system with 3DES implementation and it's testing.

For help with ideas, information, papers and other support I would like to thank Mike Anshel, Alexei D. Myasnikov, Vladimir Remeslennikov, Alexander Ushakov, Robert Haralick, Alexei Kvaschuk, Vladimir Shpilrain, Alexander Borovik, Denis Spellman, Gilbert Baumslag, Robert Gilman and Denis Serbin.

# Contents


I Introduction 1

1 Discussing the thesis 2

1.1 Brief description of thesis results . . . . . 2

1.2 Motivation . . . . . 4

1.2.1 Heuristic algorithms . . . . . 4

1.2.2 Braid group cryptography and Dehornoy forms . . . . . 6

2 Heuristic algorithms 8

2.1 Genetic algorithms . . . . . 8

2.2 Pattern recognition . . . . . 10

2.2.1 General remarks on Pattern Recognition tasks . . . . . 10

2.2.2 Feature Vectors . . . . . 12

2.2.3 Pattern Recognition Tools and Models . . . . . 14

II Solving equations 19

3 Equations in words 20

3.1 Introduction . . . . . 20

3.2 Solving equations in a free group . . . . . 22

3.2.1 Problem statement . . . . . 22

3.2.2 The algorithm . . . . . 22

3.2.3 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.3 Equations in a free semigroup. . . . . . . . . . . . . . . . . . . . . . . . 28

3.3.1 Solving equations in a free semigroup . . . . . . . . . . . . . . 28

3.3.2 The algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.3.3 Comparing genetic and direct algorithms . . . . . . . . . . . . 31

**4 Effective algorithm for finding all solutions of one-variable equations in free groups 33**

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.2 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.3 Pseudo-solutions of one-variable equations over free groups . . . . . . 35

4.3.1 Main results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

4.3.2 Pseudo-solutions of quadratic equations . . . . . . . . . . . . . 37

4.3.3 Reduction of cubics to quadrics . . . . . . . . . . . . . . . . . . 40

4.4 Description of solutions of one-variable equations over $F$ . . . . . . . 44

4.5 Description and Complexity of the Algorithm . . . . . . . . . . . . . 46

**5 Quadratic equations and the genus problem 54**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

5.2 Problem statement and known results . . . . . . . . . . . . . . . . . . 55

5.3 Reduction to free semigroups . . . . . . . . . . . . . . . . . . . . . . . 56

5.4 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

5.4.1 An example of genus 2 . . . . . . . . . . . . . . . . . . . . . . . 57

5.4.2 Another possible example of genus 2 . . . . . . . . . . . . . . . 58

**III Pattern recognition methods in groups and applications to cryptography 59**

**6 Pattern recognition methods and primitive elements in $F_2$ 60**

6.1 Introduction . . . . . 60

6.2 Pattern recognition system . . . . . 61

6.3 An iterative clustering procedure . . . . . 64

6.4 Classification by subwords . . . . . 66

6.5 Classification by exponents . . . . . 68

**7 Attacking Dehornoy forms** **74**

7.1 Introduction . . . . . 74

7.1.1 Definitions . . . . . 75

7.1.2 Generating test data . . . . . 77

7.2 Measuring data leakage . . . . . 79

7.2.1 Cyclical Hamming Distance . . . . . 79

7.2.2 Experimental data . . . . . 80

7.2.3 Levenstein distance . . . . . 83

7.2.4 Conclusions . . . . . 83

7.3 Partial attacks on AAG crypto-system . . . . . 84

7.3.1 Decreasing data leakage . . . . . 84

7.3.2 Genetic attack . . . . . 88

7.3.3 Search space reduction . . . . . 90

**8 Key hiding** **92**

8.1 Introduction . . . . . 92

8.2 The key hiding scheme . . . . . 93

8.2.1 Modifications in the 3DES algorithm . . . . . 93

8.2.2 Generating hiding functions . . . . . 95

8.2.3 Security of the key hiding method . . . . . 97

8.2.4 Improving performance . . . . . 98

**Bibliography** **99**

# Part I

# Introduction

# Chapter 1

# Discussing the thesis

## 1.1 Brief description of thesis results

In this thesis we developed some methods of a new emerging area of mathematics, which we call experimental algebra. We designed new algorithms, both deterministic and heuristic, aimed at hard problems in algebra. Using these algorithms we solved some open problems in groups and semigroups. These algorithms are implemented and organized in a package. In addition to implementations of stand-alone algorithms for solving particular problems, the package contains a variety of reusable components - tools that can be used to modify the behavior of existing algorithms, to form alternative strategies and to build new algorithms out of provided components.

In Chapter 3 we describe genetic algorithms for solving equations in free groups and free semigroups. This is the first time genetic algorithms have been used successfully in algebra. Notice that there is a famous Makanin's algorithm for solving equations. The best known upper bound for this problem is PSPACE and there are no known implementations for the algorithm, which are practical, not now and not likely to be in the future. Our algorithms, on the other hand, are capable of finding some (not necessarily all) solutions quickly.

In Chapter 4 we present an effective deterministic algorithm for finding all solutions of one-variable equations in free groups. It is known that all solutions of an one-variable equation can be described by a closed form - a finite union of parametric words, but there are no algorithms, which can produce it. Applying Makanin's algorithm would not be practical and also will not describe the solutions in the closed form. Our algorithm is the first one to do so and it has a polynomial-time complexity.

In Chapter 5 we demonstrate how genetic algorithms can be used in solving an open problem, namely the so-called *Genus problem*. We do so by reducing a quadratic equation for the Genus problem to a system of many equations in a free semigroup and solving the system by a specially designed genetic algorithm.

In Chapter 6 we apply regression models and other pattern recognition techniques to the task of classifying primitive elements of a free group. We suggest a general method, which allows one to uncover hidden mathematical structure for primitive elements only by looking at simple properties of words that represent them. We show that using these methods one can get some mathematical information about primitive elements without any prior algebraic knowledge about free groups. For example, using pattern recognition methods we have been able to recover Whitehead theorem and Nielsen criterion for free groups of rank 2.

In Chapter 6 we experiment with Dehornoy forms for words in a braid group. Our primary goal is to evaluate how well these forms perform when used for hiding information. We present a number of attacks against these forms as used in the known Anshel-Anshel-Goldfeld (AAG) crypto-system. Our best hybrid attack (genetic algorithm together with pattern recognition tools) recovers the key for a reduced key-size AAG - when the key length is 35% of the recommended length.

In Chapter 8 we introduce a key hiding method for 3DES, which is based on Dehornoy forms in braid groups. We suggest a way to use the Dehornoy algorithm,

which makes it difficult to recover the 3DES keys when either theoretical or practical (by a hacker) attack is mounted. The implementation of this method has been extensively tested for performance and is likely to become a part of a commercial software.

## 1.2 Motivation

### 1.2.1 Heuristic algorithms

Many problems in groups are undecidable. Even decidable problems are usually exponential in nature and cannot be approached by modern computers now or in the near future. The same is true for many polynomial algorithms since the power of the polynomial involved is to big for the problem to be completed in real time. While algorithms for solvable problems may be optimized and otherwise adapted for the computers, the unsolvable ones pose the real problem. Fortunately, many interesting problems are recursively enumerable. For example, the word problem in a finitely presented group. It easy to write a program which would list all words representing trivial elements in a given finitely presented group by enumerating all consequences of defining relations. So any trivial element will eventually appear on the list. The problem is, no matter how short your word is, there is no bound for the number of consequences we need to look through in order to find it. To simplify the discussion let us consider a particular method of enumerating trivial words. Assume that we have an infinite amount of memory and therefore can represent any group as a Cayley graph. First we choose the origin, say vertex $v$. Then we add every relator from a given presentation as a closed path from $v$ to $v$, labelling every edge of this path by the corresponding letter of the relator. After that we can continue building the graph by adding all conjugates of the relators in the way just described. It is clear that such a graph represents all trivial elements of the group. Given a word one

may try to start at $v$ and follow the edges corresponding to the letters of the word. If the word was read successfully in the graph, and the vertex we end up in is $v$, then the word is trivial. The problem is that for an infinite group the corresponding graph is infinite. However, we can already use it while it is being constructed, in other words we can see if the word is already on the trivial list. Given a group with five generators (ten, if counting inverses) the graph may contain $10^{10}$ words of length ten in the worst case. Well, given a good computer we can enumerate all these words. What about words of length 20? No computer can do $10^{20}$, not now, not in 10 years. What about 30,40,100? The fact is, we can enumerate all paths in the graph in a very small radius only.

The alternative to full enumeration is the random selection method. Just pick a random number of randomly chosen consequences of the given relators, freely reduce their product and compare the result with your word. The chance that you get a match is as negligible as before. The probability of finding the wanted word is $1/10^{30}$, $1/10^{40}$, $1/10^{100}$ etc.

The third approach for searching words or any other objects or solutions is thought of a "smart" search, which will not enumerate all solutions or choose them randomly but perform a search of the solution space based on the given information. The subject of this chapter is a genetic search or genetic algorithms for searching a space of solutions. It is a parallel beam search which incorporates techniques used by biological evolution.

We had much success in applying genetic algorithms for solving problems in combinatorial group theory. Their fast performance and advantages they had over many other approaches made them an important part of Magnus [1] - the software that contains a few hundreds algorithms for infinite groups. Magnus is being developed in the City College of New York, funded by National Science Foundation and freely available for everybody.

Another way to solve problem quicker is to use pattern recognition methods. Pattern recognition may provide valuable information about the properties of objects under consideration, may uncover some hidden structure, which would enable one to understand the problem better and then solve it more efficiently. Objects, such as words for example, can possibly be separated into classes, each satisfying a specific property, and then each class can be dealt with separately and more efficiently than trying to deal with all of them together. An uncovered information can also be used in heuristic algorithms to improve their performance. For example, one can use such information in a fitness function or genetic operators to reduce the search space of a genetic algorithm.

### 1.2.2 Braid group cryptography and Dehornoy forms

There are number of public encryption schemes based on difficulty of solving the word and conjugacy problems in braid groups [61, 62, 63]. The braid groups are of interest to cryptographic community because they can possibly offer:

- Immunity from quantum computer attacks.
- Protection from side-channel attacks.
- Faster speed for generating and transmitting keys.

In the mentioned above public key exchange protocols the communicating parties send messages to establish a common key. The cryptography in these schemes is based on the assumption that the conjugacy problem in braid groups is hard to solve, and so the parties exchange words conjugated by some other private words in a braid group. To prevent an attacker from simply reading private conjugators from the messages being exchanged, and therefore avoid the difficulty of solving the conjugacy problem, these conjugated words need to be shuffled, so that the conjugators are difficult to extract.

To hide the conjugators one can, for example, compute some known normal form for a word being sent and send it instead, or send an image of homomorphism into some other presentation for a braid group, such as Burau or colored Burau representation. Taking a word in its Dehornoy form is another possible choice for shuffling the word. For instance, it is used in [62] as the first of two steps to hide the conjugators. One reason for using Dehornoy forms is of course that it is very efficient to compute. On the other hand it seems that it mixes a given word very well. Here we would like to experiment with Dehornoy forms in order to understand how good this mixing really is.

# Chapter 2

# Heuristic algorithms

## 2.1 Genetic algorithms

The following is a very brief introduction into genetic algorithms. For more details see [2, 3].

Genetic algorithms (GAs) provide a learning method motivated by analogy to biological evolution. In the broadest sense, a GA creates a set of solutions that reproduce based on their fitness in a given environment. The process follows this pattern:

1. An initial population of random solutions is created.

The classical genetic algorithm operates with population of bit strings, hence genetic operators operate with bits too. Of course, often it is necessary to represent solutions differently and modify operators accordingly. Solutions or population members are initialized randomly at first and, if the algorithm stops, one or some of them will be real solutions to the problem. Solutions can be words, relations, group representation, sets or any other objects.

2. Each member of the population is assigned a fitness value based on its evaluation

against the current problem.

Fitness is a very important notion in GAs. Every potential solution is constantly evaluated, i.e. assigned a numerical value or a score, and like in nature the fittest will survive. It is always difficult to chose a fitness function which evaluates every potential solution or hypotheses accurately. The more accurate function we give the faster algorithm will work. At least we have to assure that the fitness of an optimal or real solution has the best fitness value possible.

3. Solutions with a higher fitness value are most likely to parent new solutions during reproduction.

Survival of the fittest is implemented by *fitness proportional selection* or *roulette wheel selection*. The probability $Pr(s_i)$ of selecting solution $s_i$ for reproduction is given by

$$Pr(s_i) = \frac{Fitness(s_i)}{\sum_{j=1}^{p} Fitness(s_j)},$$

where $p$ is the population size. This process of selecting population members for reproduction is called *selection*. Then we perform reproduction by selecting $p$ pairs $(s_1, s_2)$ according to the distribution of $Pr(s_i)$ and applying the *Crossover* operator. Corresponding to biological crossover, the software version combines a pair of parents by randomly selecting a point at which pieces of the parents' bit strings are swapped, i.e. we take a randomly chosen initial segment of the first parent and concatenate it with the terminal segment of the second parent, thereby producing an offspring. Then these offspring are exposed to *mutation* - a random change in a string, usually mean inverting one or more bits.

4. The new solution set replaces the old, a generation is complete, and the process continues at step 2.

The final step is called *replacement*. Here the new offspring produced by *crossover*

and affected by *mutation* replace the old population and the process continues from step 2. It is important to notice that we may want to replace only part of the old population by new members. Usually it helps to keep a few members with the best fitness values in the population. This process is called *elitist selection* .

## 2.2 Pattern recognition

Here we discuss all basic definitions, notions, and notations concerning pattern recognition (**PR**) systems. For more details see [52].

### 2.2.1 General remarks on Pattern Recognition tasks

One of the main applications of Pattern Recognition (**PR**) techniques is classification of a variety of given objects into categories. Usually classification algorithms or *classifiers* try to find a set of measurements (properties, characteristics) of objects, called *features*, which gives a descriptive representation for the objects.

Generally, pattern recognition techniques can be divided in two principal types:

- *supervised learning;*
- *unsupervised learning* (*clustering*).

In supervised learning the decision algorithms are "trained" on a prearranged dataset, called *training* dataset in which each pattern is labelled with its true class label. If such information is not available, one can use clustering. In this case clustering algorithms try to find *clusters* or "natural groupings" of the given objects. In this paper we use supervised learning pattern recognition algorithms.

Every pattern recognition task of the supervised learning type has to face all of the following issues:

1. **Obtaining the data.** The *training datasets* can be obtained from the real world or generated by reliable procedures, which provide independent and representative sampled data.

2. **Feature extraction.** The task of feature extraction is problem specific and requires knowledge of the problem domain. If such knowledge is limited then one may consider as many features as possible and then try to extract the most "significant" ones using statistical methods.

3. **Model selection.** The model is the theoretical basis for the classifier. A particular choice of the model determines the basic design of the classifier (though there might be some variations in the implementation). Model selection is one of the most active areas of research in pattern recognition. Usually model selection is closely related to the feature extraction. In practice, one may try several standard models starting with the simplest ones or more economic ones.

4. **Evaluation.** Evaluation of the performance of a particular **PR** system is an important component of the pattern recognition task. It answers the question whether or not the given system performs as required. To evaluate a system one can use various accuracy measures, for example, percentage of correct answers. To get reliable estimates other sets of data that are independent from the training sets must be used. Such sets are called *test datasets*.

   Typically we view a **PR** system as consisting of components 1)-4).

5. **Analysis of the system.** Careful analysis of performance of a particular classifier may improve feature extraction and model selection. For example, one can look for an optimal set of features or for a more effective model. Moreover, through analysis of the most significant (insignificant) features one may gain a new knowledge about the original objects.

### 2.2.2 Feature Vectors

A feature vector is just a vector of properties about the object of interest, in our case about the mathematical object of interest. The properties are chosen to be ones thought to be relevant to the problem. We will illustrate the selection of features for words in a free group $F = F(X)$ with basis $X$.

Let $w$ be a reduced word in the alphabet $\in X^{\pm 1}$. Below we describe features of $w$ which characterize a certain placement of specific words from $F(X)$ in $w$.

Let $K \in \mathbb{N}$ be a natural number, $v_1, \ldots, v_K \in F(X)$ be words from $F(X)$, and $U_1, \ldots, U_{K+1} \subseteq F(X)$ be subsets of $F(X)$. Denote by

$$C(w, U_1 v_1 \ldots v_K U_{K+1})$$

the number of subwords of the type

$$u_1 v_1 u_2 \ldots v_K u_{K+1},$$

where $u_j \in U_j$, which occur in $w$. For fixed $K, v_1, \ldots, v_K, U_1, \ldots, U_{K+1}$ we obtain a *counting function*

$$w \in F \longrightarrow C(w, U_1 v_1 \ldots v_K U_{K+1}) \in \mathbb{N} \tag{2.1}$$

The normalized value

$$\frac{1}{|w|} C(w, U_1 v_1 \ldots v_K U_{K+1})$$

is called a *feature* of $w$ and the function

$$w \in F \longrightarrow \frac{1}{|w|} C(w, U_1 v_1 \ldots v_K U_{K+1}) \in \mathbb{R}$$

is called a *feature function* on $F$. Usually we omit $U_i$ in our notations if $U_i = \emptyset$. If $\bar{C} = (C_1(w), \ldots, C_N(w))$ is a sequence of counting functions like (2.1) one can

associate with $w$ a vector of real numbers:

$$f_{\bar{C}}(w) = \frac{1}{|w|} < C_1(w), \ldots, C_N(w) > \in \mathbb{R}^N$$

which is called a *feature vector*. Every choice of the sequence $\bar{C}$ gives a vector $f_{\bar{C}}(w)$ which reflects the structure of $w$.

For example, if $a \in X^{\pm 1}$ then $C(w, a)$ counts the number of occurrences of the letter $a$ in $w$. The feature vector (where for simplicity we assume that the components are written in some order which we do not specify)

$$f_0(w) = \frac{1}{|w|} < C(w, a) \mid a \in X^{\pm 1} >$$

shows the frequencies of occurrences of letters from $X^{\pm 1}$ in $w$. The feature vector

$$f_1(w) = \frac{1}{|w|} < C(w, v) \mid |v| = 2 >$$

shows the numbers of occurrences of words of length two in $w$ relative to the length of $w$.

To visualize some structures described by the counting functions above we associate with a given word $w \in F(X)$ a weighted labelled directed graph $\Gamma(w)$. Put $V(\Gamma(w)) = X^{\pm 1}$. For given $x, y \in X^{\pm 1}$ and $v \in F(X)$ we connect the vertex $x$ to the vertex $y$ by an edge with a label $v$ and weight $C(w, xvy)$. Now, with every edge from $x$ to $y$ with label $xvy$ one can associate a counting function $C(w, xvy)$, and vice versa. It follows that every subgraph $\Gamma$ of $\Gamma(w)$ gives rise to a particular set of counting functions $\bar{C}_\Gamma$ of the type $C(w, xvy)$, and conversely, every set $\bar{C}$ of counting functions of the type $C(w, xvy)$ determines a subgraph $\Gamma_{\bar{C}}$ of $\Gamma(w)$. For instance, the feature mapping $f_1$ corresponds to the subgraph $\Gamma_1(w)$ of $\Gamma(w)$ which is in a sense a directed version of the so-called *Whitehead graph* of $w$.

Let $U_n$ be the set of all words in $F$ that are length $n$. Let $W_n$ be the set of all words in $F$ that are of length $n$ or less. Other relevant features can be defined as follows. Each corresponds to various subgraphs of the graph $\Gamma(w)$:

$$f_2(w) = \frac{1}{|w|} < C(w, x_1 U_1 x_2) \mid x_1, x_2 \in X^{\pm 1} >;$$

$$f_3(w) = \frac{1}{|w|} < C(w, x_1 U_2 x_2) \mid x_1, x_2 \in X^{\pm 1} >;$$

$$f_4(w) = \frac{1}{|w|} < C(w, x_1 U_3 x_2) \mid x_1, x_2 \in X^{\pm 1} >;$$

$$f_5(w) = \frac{1}{|w|} < C(w, x_1 W_1 x_2) \mid x_1, x_2 \in X^{\pm 1} >;$$

$$f_6(w) = \frac{1}{|w|} < C(w, x_1 W_3 x_2) \mid x_1, x_2 \in X^{\pm 1} > .$$

### 2.2.3 Pattern Recognition Tools and Models

There is a variety of pattern recognition tools that are useful for determining a way of making a distinction given a set of feature vectors from one class and a set of feature vectors from another class. For each given class of objects, we sample objects from the class and construct the set of corresponding feature vectors. The pattern recognition technology provides a way of determining a best or near best boundary in the feature space that distinguishes the one class from the other. In this section we will review some of the basic techniques. The reader interested in a fuller discussion may consult general references [58],[5][6],[7].

#### Principal Components

There are occasions when the feature vectors coming from a class either all lie in a small dimensional flat or most of them lie in a small dimensional flat. Principal component analysis can determine this.

Let $x_1, \ldots, x_N$ be the set of feature vectors sampled from a given class. Define their sample mean $\mu$ by

$$\mu = \frac{1}{N} \sum_{n=1}^{N} x_n$$

Define their sample covariance $C$ by

$$C = \frac{1}{N} \sum_{n=1}^{N} (x - \mu)(x - \mu)'$$

Let $t_1, \ldots, t_N$ be the eigenvectors of $C$ with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_N$. Should the feature vectors indeed lie in a small dimensional flat, then there will be a $K < N$ such that $\lambda_k = 0,\ k = K + 1, \ldots, N$.

In this case, feature vectors $x$ coming from objects that are in the class can be recognized by testing whether or not

$$||T(x - \mu)|| = 0$$

where $T$ is a $N \times (N - K)$ matrix whose columns are eigenvectors $t_{K+1}, \ldots, t_N$.

Those in the class will have $||T(x-\mu)|| = 0$. $||T(x-\mu)|| > 0||$ is a sure indication that $x$ comes from an object out of the class, but there may be some objects out of the class for which $||T(x - \mu)|| = 0$.

In the case of two classes, we form the matrix $T_1$ from the zero eigenvalue eigenvectors of the covariance matrix from the class one feature vectors and the matrix $T_2$ from the zero eigenvalue eigenvectors of the covariance matrix from the class two feature vectors.

Now if $||T_1(x - \mu_1)|| = 0$ and $||T_2(x - \mu_2)|| > 0$, we assign vector $x$ to class one. If $||T_1(x - \mu_1)|| > 0$ and $||T_2(x - \mu_2)|| = 0$, we assign vector $x$ to class two. If $||T_1(x - \mu_1)|| = 0$ and $||T_2(x - \mu_2)|| = 0$, feature vector $x$ comes from an object that is both class 1 and class 2. If $||T_1(x - \mu_1)|| > 0$ and $||T_2(x - \mu_2)|| > 0$ feature vector

$x$ comes from an object that is neither class 1 nor class 2.

**Classifying by Distance**

Let $T_1$ and $T_2$ be defined as before. We form the discriminant function

$$f(x) = ||T_1(x - \mu_1)|| - ||T_2(x - \mu_2)||$$

which measures the difference between feature vector $x$ and the flat associated with class 1 and the flat associated with class 2. The decision rule is to assign vector $x$ to class 1 if $f(x) > \theta$, otherwise assign to class 2. Here, after the discriminant function is defined we determine the value of $\theta$ that minimizes the error.

Classifying by distance can also be done with respect to the class means. Here the discriminant function is defined by

$$f(x) = (x - \mu_1)' C_1^{-1} (x - \mu_1) - (x - \mu_2)' C_2^{-1} (x - \mu_2)$$

As before, the decision rule is to assign vector $x$ to class 1 if $f(x) > \theta$, otherwise assign to class 2. Here also after the discriminant function is defined we determine the value of $\theta$ that minimizes the error.

**Linear Classifiers**

Classifying may be done by a linear decision rule. Here the discriminant function is given by

$$f(x) = v'x$$

where vector $v$ is the weight vector and is the normal to the hyperplane separating the feature space into two parts.

If $f(x) < \theta$ the decision rule is to assign the vector $x$ to class 1 otherwise to class

2.There are a variety of ways to construct the weight vector $v$. One method is by regression.

**Regression Classifier**

In the regression classifier, we form a matrix $A$ whose rows are the feature vectors. We form a vector $b$ whose $kth$ component is 0 if the $kth$ feature vector comes from class one and whose $kth$ component is 1 is the $kth$ feature vector comes from class two. We determine the weight vector $v$ as that vector that minimizes $||Av - b||$. The minimizing vector $v$ is given by the normal equation

$$v = (A'A)^{-1}A'b$$

The discriminant function is defined by

$$f(x) = v'x$$

We assign a vector to class one if $f(x) < \theta$ and to class two otherwise. $\theta$ is chosen to minimize the error of the assignment.

**Quantizing**

Let $f$ be a discriminant function. We evaluate $f(x)$ over all the sampled vectors $x$ from class one and from class two to determine the range. We divide the range in a fixed number $M$ of quantizing intervals. The simplest way is called equal interval quantizing. Here the range is divided up into $M$ equal intervals. In each interval the number of sampled vectors coming from class one and coming from class two is determined. The interval is labelled by the class of the majority of the vectors in it.

A vector $x$ having discriminant value $f(x)$ which falls into the $mth$ quantizing interval is assigned to the class that labels the quantizing interval.

Another simple alternative quantizing scheme is to divide the range into intervals each of which have the same number of sampled discriminant values. This is called equal probability quantizing.

A more complex scheme is to divide the discriminant range into $M$ intervals in such a way that the classification error is minimized.

# Part II

# Solving equations

# Chapter 3

# Equations in words

## 3.1 Introduction

Solving equations in equationally defined free algebras (Unification) is a widely used technique in Computer Science, see e.g. [8]. In particular, solving equations in free semigroups, i.e. word equations, is of great interest in e.g. associative rewriting and completion, string unification in PROLOG-3, extensions of string rewrite systems, unification in some theories with associative non-commutative operators, and in symbolic mathematical packages.

The problem of solving word equations was considered at least since the late fifties by A. Markov (see [9]). Partial solutions were known long ago: in the late sixties Hmelevskiĭ [9] solved the problem for equations in three variables, Matiyasevich [10] solved it for the case in which each variable occurs at most twice, and in the seventies Lentin [11], Plotkin [12], and Siekmann [13] gave semi-decision procedures.

In 1976 Makanin [14] solved the problem in its complete generality giving us an algorithm to decide if arbitrary systems of word equations have solutions (the case of systems of equations reduces easily to the case of only one equation). Some year later, in 1982, again Makanin proved that solving equations in free groups

was a decidable problem [15]. Both Makanin's algorithms have received very much attention. The enumeration of all unifiers (which gives all possible solutions to an equation as well) was done by Jaffar for semigroups [16] and by Razborov for groups [17]. In the meantime, there has been some work simplifying various aspects of the algorithm and even some implementations [18, 19, 20, 21]. Also, Schulz [20] generalized the result for the case of variables with regular constraints.

Jaffar in [16] calculated an upper bound for the running time of Makanin's algorithm which was four times exponential in the length of the equation. Later Kościelski and Pacholski [22] improved it to non-deterministic triple exponential time. More detailed analysis by Gutiérrez [24] and Diekert [26] showed an upper bound of double exponential space-complexity (i.e. no more than triple exponential time-complexity). The best upper-bound so far is EXPSPACE [25]. Plandowski, without using Makanin's algorithm , presented an upper-bound of PSPACE for the problem of satisfiability of equations in a free semigroup [27]. On the other hand, the analysis of the complexity of Makanin's algorithm for groups was done by Kościelski and Pacholski [23], who showed that it is not primitive recursive. With respect to lower bounds, the only known lower bound for both problems is NP-hard, which seems to be weak for the case of free groups.

In this chapter we describe genetic algorithms for solving equations in free groups and free semigroups. This is the first time genetic algorithms have been used successfully in algebra. Our algorithms are capable of finding some (not necessarily all) solutions quickly.

## 3.2 Solving equations in a free group

### 3.2.1 Problem statement

Given a free group $F$ we define an equation in variables $x_1, x_2, ..., x_n$ over $F$ as formal equation of the form

$$g_1 x_{i_1}^{\epsilon_1} g_2 x_{i_2}^{\epsilon_2} ... x_{i_n}^{\epsilon_n} g_{n+1} = 1, \tag{3.1}$$

where $g_i \in F, \epsilon_i = \pm 1$. The problem is to find a solution, i.e. a substitution for the variables $x_i$ such that the equation becomes identity in $F$.

### 3.2.2 The algorithm

#### Population

Since we are looking for a solution for a given equation, it is natural to work with a population of solutions, i.e. population consisting of a finite number of substitutions for the variables of the equation. The algorithm uses population of 50 tuples of words. Each tuple has the same number of elements as the number of variables and thereby represent one possible substitution for the variables. At the beginning every substitution for every variable for all 50 solutions is generated randomly. Experiments showed that the lengths of the random words does not make a significant difference, since even when starting with very short words and given a small probability for increasing the length, the algorithm was able to adapt and found a solution for the equation which had only lengthy solutions.

#### Fitness function

The fitness function has to measure how close a potential solution is to a real solution. Here we defined it in the following way. A member of the population, say $(w_1, ..., w_n)$, defines a substitution for the variables of equation (3.1): $x_i \to w_i$. Then

the length of the substituted image

$$f(w_1, ..., w_n) = |g_1 w_{i_1}^{\epsilon_1} g_2 w_{i_2}^{\epsilon_2} ... w_{i_n}^{\epsilon_n} g_{n+1}|$$

is the fitness value of $(w_1, ..., w_n)$. If $(w_1, ..., w_n)$ is a real solution to the equation (3.1), its fitness value is 0. Otherwise it is strictly greater than 0. Moreover, the longer the substituted image we get the further from 0 is the corresponding fitness value. So, the algorithm's goal will be to minimize members' fitness values, with 0 as the optimum value.

One might say that such fitness function is not adequate, since for example, it assigns a better value to an image of length 1 than to an image of length 5, while it might happen than 1 is a local minima, a misleading way, and 5 is one step apart from a solution. It may happen, of course. However, in addition to the fitness function, a genetic algorithm has powerful operators like crossover, mutation and others. The search performed is much more complicated than a simple one-directional search. In such an environment GA is a capable of considering a great variety of solutions no matter how limiting the fitness function is. And because of the nature of GA, reaching a local minima does not pose such a big problem as in many other methods, like gradient search in neural networks, for example.

The fact is, a simple function like this performs exceptionally well. One might think more and design a more accurate fitness function, if it seems possible, therefore helping the algorithm. One must remember, however, that more sophisticated function will probably take more time to compute, therefore slowing down the algorithm. And it is not always clear who is right, our intuition or evolution? Remember, that we started to apply genetic algorithms since we couldn't solve the problem in the first place. Our approach is the following. Save your time and use the simplest function possible, the one which is naturally adequate. Then, when the algorithm is

finished, invent and play with other functions, so that there's something to compare them with.

In order to simplify the use of roulette wheel selection and to keep analogy with biology - the fitter the better - the fitness values are changed in the following way. Every fitness value $v$ is assigned the value of difference between the maximal fitness value and $v$. Therefore the best solution now has the largest value and we are maximizing it. In addition, to speed up the evolution, *fitness scaling* was applied. In this case every fitness value was replaced by its square.

### Crossover

Crossover between two tuples of words is done coordinate-wise, i.e. the first coordinate of the first tuple crosses over with the first coordinate of the second tuple, the second coordinate of the first tuple crosses over with the second coordinate of the second tuple and so on. Therefore we need to define a crossover on words. The algorithm uses a counterpart of the classic one-point crossover for bit strings. In other words, we take a random initial segment of the first word and concatenate it with a random terminal segment of the second word.

### Mutation

Mutation on tuples can again be defined through a mutation on their components - words. A mutation is random change in a bit string or in a word, and requires us to invert one or more bits or one or more generators in a word. The following three mutations were used in the algorithm:

Insert a new letter in a randomly chosen position - 10% chance

Delete one randomly chosen letter(generator) from the word - 10% chance

Replace one randomly chosen letter by a different one - 80% chance

It might be a good idea to replace these fixed probabilities to variables, values of

which are dependent on the length of the mutated word.

**Replacement**

The replacement works almost as usual. All members of the population get replaced by new members except the first one, which is always replaced by the best representative of the old population. This is a simple form of the elitist selection.

### 3.2.3 Experimental results

**Experiment 1**

Let us consider a simple experiment where equations are of the form

$$a^N x\, b^N = 1$$

where $x$ is a variable, $a$ and $b$ are generators of a free group, $N$ is a positive integer. Though solutions are obvious in this case, the genetic algorithm does not know that and therefore, the experiment is sound. Given a simple equation like this we can control the length of the minimal solution for each $N$ and estimate the growth of time needed for different lengths. The table below shows the results of this experiment for minimal solutions of lengths 10,20,30,40,50,60,100 and 200. For each length it contains the average number of generations, the average time of one generation and the average time needed to find the solution. All averages were computed based on 20 consecutive runs of the algorithm. All time values are in seconds and were taken on PentiumII-400MHz machine.

| Length | Avg number of generations | Avg time per generation | Avg time |
|---|---|---|---|
| 10 | 13 | 0.00810277 | 0.105336 |
| 20 | 38.7 | 0.00878626 | 0.340028 |
| 30 | 175.3 | 0.0107125 | 1.8779 |
| 40 | 194.7 | 0.0133946 | 2.60792 |
| 50 | 215.3 | 0.0152716 | 3.28797 |
| 60 | 412.7 | 0.0168407 | 6.95016 |
| 100 | 899.5 | 0.026206 | 23.5723 |
| 200 | 2013 | 0.0473142 | 95.2434 |

As can be seen from the description of the algorithm and as confirmed by the table above, the average time per generation is growing linearly with the length of the equation. The number of generations can also be bounded by a linear function, and therefore the average time can be bounded by a quadratic polynomial. Though a more thorough analysis is needed, the experiment has already shown dramatic improvement over exponential (at least) algorithm discussed above.

**Experiment 2**

As experiments have shown, though the genetic algorithm does not use the fact that the given equation has a particular form, still it much easier to evolve words, pieces of which are of the form $a^N$. Here we consider a more difficult for GA equation:

$$w_1 x w_2 = 1$$

where $x$ is a variable, $w_1$ and $w_2$ are randomly generated words in the free group with the condition that there is no cancellation occurs in the word $w_1^{-1} w_2^{-1}$. This way we still control the length of the minimal solution. The results are given in the

table in the same fashion as above.

| Length | Avg number of generations | Avg time per generation | Avg time |
|---|---|---|---|
| 10 | 111.2 | 0.00581994 | 0.647177 |
| 20 | 1298 | 0.00791865 | 10.2784 |
| 30 | 62214.3 | 0.010184 | 633.589 |

The genetic algorithm was still capable to find a solution in every randomly generated case in a feasible time. It is important to notice that the exponential algorithms cannot be used to find solutions for length 20 or 30. Also, it is interesting to see the difference in amounts of time it takes the program to solve virtually the same problem. As an example, the table below shows time measurements for 10 consecutive runs of the genetic algorithm.

| Experiment number | Number of generations | Time (in seconds) |
|---|---|---|
| 1 | 8966 | 109.559 |
| 2 | 184698 | 1904.29 |
| 3 | 217421 | 2159.09 |
| 4 | 46386 | 476.261 |
| 5 | 60479 | 618.543 |
| 6 | 560 | 5.89761 |
| 7 | 39147 | 412.737 |
| 8 | 50516 | 506.43 |
| 9 | 12995 | 132.838 |
| 10 | 975 | 10.2399 |

#### Other experiments

The equations considered above had only one variable. The purpose of this was the desire to control the length of the minimal solution which is difficult when given many variables. In Magnus we tried and were successful to solve equations with five and more variables. There is a discussion group on equations on the Magnus home page [1] where anyone can suggest an equation to solve. The reason for the group is to test GA against the problems interested to people and to improve the existing implementation.

## 3.3 Equations in a free semigroup.

### 3.3.1 Solving equations in a free semigroup

The case of a free semigroup (free monoid) G is somewhat simpler than solving equations in a free group, since we don't have cancellations. There exists very simple algorithm to solve equations of the type

$$g_1 x_{i_1} g_2 x_{i_2} ... x_{i_n} = C, \tag{3.2}$$

in a free semigroup. Here $x_1, x_2, ..., x_n$ are variables, $C$ and $g_i$ are constants in G. Obviously, the length of each variable is bounded by $|C|$. So full enumeration works (in theory!). But again it takes exponentially many steps. As before genetic algorithm performs much better.

### 3.3.2 The algorithm

#### Population

As in the case of equations in a free group, the genetic algorithm for solving a "graphic" equation works with a population of tuples, where each component of

each tuple is a word which substitutes one variable. The population size is again 50, since this number has found to be efficient in many experiments. In the beginning every word generated randomly. For the genus problem (see Section 5) we know that the length of the constant is 28 and we have 9 variables. Therefore, there is no need to generate random words longer than $28/9 \approx 3$.

**Fitness function**

After applying a substitution to the left part of the equation (3.2) the algorithm needs to compare the result with the given constant. In general, we have two words which we want to compare. And the closer they are to each other, the bigger the fitness function should be. Again, keeping in mind the importance of simplicity, I chose the Hamming distance to be a fitness function in this case. First the shorter word gets extended by appending an unused before character, $ for example. Then both words are compared coordinate-wise. The fitness value is initialized to zero and gets increased by one for every mismatch between the characters. Therefore, fitness is equal to zero when two words are identical and we want to minimize the fitness function once again. As before we change it to the equivalent maximization task and scale the fitness values.

Often when comparing two words, it is preferable to compute the Hamming distance between the first word and all cyclical permutations of the second and then choose the least distance as a fitness value. For example, the words *abcd* and *bcd* will be different in every coordinate and therefore have fitness four. On the other hand they are similar, since they have a big common piece *bcd*. The new fitness function will notice this similarity and assign a better value one, which should speed up the algorithm. For the genus problem (see Section 5) we must compare any substitution image with every cyclical permutation of the given constant. So such a procedure is not only desirable but a must do. Putting it into the fitness function kills two birds

with one stone.

### Crossover

Crossover between two tuples of words is done coordinate-wise, i.e. the first coordinate of the first tuple crosses over with the first coordinate of the second tuple, the second coordinate of the first tuple crosses over with the second coordinate of the second tuple and so on. Therefore we need to define a crossover on words. The algorithm uses a counterpart of the classic one-point crossover for bit strings. In other words, we take a random initial segment of the first word and concatenate it with a random terminal segment of the second word.

Different from the algorithm for the equations in a free group, crossover was applied not to all pairs but to 95% only. This is best experimentally found chance of crossover for this task.

### Mutation

Mutation on tuples can again be defined through a mutation on their components - words. A mutation is random change in a bit string or in a word, and requires us to invert one or more bits or one or more letters in a word. The following three mutations were used in the algorithm:

Insert a new letter in a randomly chosen position - 10% chance

Delete one randomly chosen letter(generator) from the word - 10% chance

Replace one randomly chosen letter by a different one - 80% chance

Different from the algorithm for the equations in a free group, mutation was applied not to all pairs but to 95% only. This is best experimentally found chance of mutation for this task.

**Replacement**

After a number of experiments the best way to perform replacement was the strongest form of elitist selection [4] : a member of the new population will replace a member of the old population only if it has better fitness value. In other words, the population on every generation consists only of the best solutions found so far.

### 3.3.3 Comparing genetic and direct algorithms

Though a simple enumeration of all substitutions on variables of a given equations is futile, in some special cases it is possible to limit the amount of enumeration and hence, speed up the algorithm. For example, when considering nine forms for the genus problem (see Section 5) one notices that these equations are quadratic with nine variables, each variable occurs exactly twice and they are placed in such a way that being substituted they seriously limit possibilities for other variables. Therefore a “smart” direct algorithm can be written, which would takes these into account and cut the enumeration dramatically. The table below is the comparison between genetic algorithm described in this chapter and the best direct algorithm we could provide in a reasonable time. The first column contains the number of an equation from the theorem. The second and the third - the time needed to obtain a solution by genetic and direct approach correspondingly. All time values are given in seconds and taken on PentiumII-400MHz machine. The average time spent by genetic algorithm is 98.13012. The average time spent by direct algorithm is 251.11076. So, despite all the efforts, the genetic algorithm still outperforms on average. It is important to notice that the direct algorithm was designed to suit this particular case while the genetic algorithm works for every equation. Also, it took about ten times more time to implement the direct approach.

| Equation number | Genetic algorithm | Direct algorithm |
| --- | --- | --- |
| (5.2) | 32.5003 | 427.094 |
| (5.3) | 88.0291 | 87.2012 |
| (5.6) | 129.897 | 416.294 |
| (5.8) | 160.607 | 250.337 |
| (5.9) | 79.6172 | 74.6276 |

# Chapter 4

# Effective algorithm for finding all solutions of one-variable equations in free groups

## 4.1 Introduction

In 1960 Lyndon proved that one needs only finitely many *parametric words* to describe solutions of one-variable equations over $F$ [46]. Further progress in this direction was made by Appel [37] and Lorents [45], who gave the exact form of the required parametric words. Namely, it turned out that the solution set of $E(x)$ in $F$ is contained in a finite union of sets of the type

$$\{p^n q \mid p, q \in F, n \in \mathbb{Z}\}. \tag{4.1}$$

Their argument was rather technical; they used direct computations in free groups and the Nielsen cancellation method. Recently, Chiswell and Remeslennikov gave an alternative proof of this result [39], which is based on ultrapowers and algebraic geometry over groups. They showed also that the solution set of $E(x)$ is precisely

a finite union of sets of the type (4.1). Another recent result is due to Gilman and and Myasnikov [36], who used context free languages to show that solutions of $E(x)$ are a finite union of sets of the form

$$\{p_1 p_2^i p_3^j p_4 p_5^k p_6^l p_7 \mid (i, j, k, l) \in S\}$$

where the $p_i$'s are words over $\Sigma^*$, and $S$ is a semilinear subset of $N^4$.

Our result is an effective deterministic algorithm for finding all solutions of one-variable equations in free groups. This is the first algorithm that solves the problem and describes all solutions in the form (4.1). We also show that it has a polynomial-time complexity.

## 4.2 Definitions

Let $F$ be a free group with a basis $X$. Let $\Sigma = X \cup X^{-1}$. For a word $w \in \Sigma^*$ we denote by $\bar{w}$ the reduced form of $w$. Observe, that there are usually several different ways to obtain $\bar{w}$ from $w$. We will assume that elements from $F$ are presented by reduced words in $\Sigma$. For words $u, v \in \Sigma^*$ we write $u \circ v$ in the case that there is no cancellation in $uv$, i.e., $|uv| = |u| + |v|$.

Let

$$E(x) = u_1 x^{\epsilon_1} u_2 x^{\epsilon_2} \ldots u_d x^{\epsilon_d}$$

be an element of a free product of an infinite cyclic group generated by $x$ and the free group $F$, i.e., $E(X)$ is a word in one variable $x$ with $\epsilon_i \in \{1, -1\}$ and with constants $u_i \in F$. By $E(v)$ we denote the word obtained by replacing $x^{\epsilon_i}$ with $v^{\epsilon_i}$. The subword $v^{\epsilon_i}$ of $E(v)$ is called the *$i$-s occurrence of $v$ in $E(v)$*. Formal expressions of the form $E(x) = 1$ are called *one-variable* equations over $F$. An element $v \in F$ is a *solution* of $E(x) = 1$ if $E(v)$ is freely equal to 1 in $F$.

**Definition 1** *We say that the i-s occurrence of $v$* cancels out *in $E(v)$ if there exists a way to reduce the word $E(v)$ such that all letters from $v^{\epsilon_i}$ cancel out during this reduction process.*

**Definition 2** *We say that an element $v \in F$ is a pseudo-solution of $E(x) = 1$ if some occurrence of $v$ cancels out in $E(v)$.*

**Remark.** Obviously every solution of $E(x) = 1$ is also a pseudo-solution of $E(x) = 1$.

## 4.3 Pseudo-solutions of one-variable equations over free groups

### 4.3.1 Main results

The following result shows that solutions of arbitrary one-variable equations over $F$ are pseudo-solutions of cubic ones. It has been stated in [36] in a slightly different manner, but the same argument works here as well.

**Theorem 1** *[Reduction to the cubic equations] Let $d \geq 3$ and let*

$$E(x) = u_1 x^{\epsilon_1} u_2 x^{\epsilon_2} \dots u_d x^{\epsilon_d} = 1$$

*be a one-variable equation over a free group $F$. If $v$ is a pseudo-solution of $E(x) = 1$ in $F$ then $v$ is a pseudo-solution of an equation of the type*

$$x^{\epsilon_{i-1}} u_i x^{\epsilon_i} u_{i+1} x^{\epsilon_{i+1}} = 1 \tag{4.2}$$

*for some $i = 1, \dots, d-1$ (where $\epsilon_0 = 0$).*

In section 4.3.3 we show that pseudo-solutions of cubic equations are, in fact, pseudo-solutions of some particular quadratic equations which one can find effectively.

**Theorem 2** *[Reduction to quadratic equations] Let*

$$E(x) = x^{\epsilon_1} u_1 x^{\epsilon_2} u_2 x^{\epsilon_3} = 1$$

*be a cubic one-variable equation over $F$. Then one can effectively find a finite set $Q(E)$ of quadratic one-variable equations over $F$ such that if $v$ is a pseudo-solution of $E(x) = 1$ in $F$ then there exist elements $v_1, v_2 \in F$ and equations $Q_1, Q_2 \in Q(E)$ such that $v = v_1 \circ v_2$ and $v_i$ is a pseudo-solution of $Q_i$, $i = 1, 2$.*

**Remark 1** *In fact, one may assume that quadratic equations from the set $Q(E)$ are of a particular form. Namely, one can effectively find a finite set $Q'(E)$ of quadratic one-variable equations over $F$ of the type*

$$Q'(E) = \{x^{\delta_1} w_i x^{\delta_2} \mid i \in I\}, \tag{4.3}$$

*and a finite set of elements $C$ of $F$ such that if $v$ is a pseudo-solution of $E(x) = 1$ in $F$ then there exist equations $Q_1, Q_2 \in Q'(E)$ and elements $c_i \in C$ such that $v = v_1 \circ v_2$ and $v_i = c_i \circ t_i$ where $t_i$ is a pseudo-solution of $Q_i$, $i = 1, 2$.*

In section 4.3.2 we give a precise description of pseudo-solutions of quadratic one-variable equations over $F$.

**Theorem 3 (Pseudo-solutions of quadratic equations)** *Let*

$$E(x) = u_1 x^{\epsilon_1} u_2 x^{\epsilon_2} u_3 = 1$$

*be a quadratic equation over $F$. Then there are finitely many pairs $(a_i, b_i)$ of elements*

*from $F$ such that every pseudo-solution of $E(x) = 1$ in $F$ takes the form $a_i b_i^n$ for suitable $i$ and $n \in \mathbb{Z}$. Moreover, one can find these pairs $(a_i, b_i)$ effectively.*

### 4.3.2 Pseudo-solutions of quadratic equations

There are two principle cases, which we consider in separate lemmas.

**Lemma 1** *Let $u, v \in F$. The left occurrence of $v$ cancels out in $vuv$ if and only if one of the following cases holds:*

*(i) $v = u_1^{-1}$ where $u = u_1 \circ u_2$, or*

*(ii) $v = u_2^{-1} u_1^{-1}$ where $u = u_1 \circ u_2^2$.*

**Proof.** Suppose that $v$ doesn't cancel out in $vu$, then $v = v_1 \circ u_1^{-1}$ and $u = u_1 \circ u_2$ such that $vu = v_1 \circ u_2$ and $v_1 \neq 1$. We have $vuv = v_1 \circ u_2 \cdot v_1 \circ u_1^{-1}$, therefore $u_2$ cancels out in $u_2(v_1 \circ u_1^{-1})$.

**Case 1.** Let $|u_2| \leq |v_1|$. Then $v_1 = u_2^{-1} v_{11}$ and

$$vuv = v_1(v_{11} \circ u_1^{-1}) = u_2^{-1} \circ v_{11} \cdot (v_{11} \circ u_1^{-1}).$$

Hence $v_{11} v_{11} = 1$ and $v_{11} = 1$, so $vuv = u_2^{-1} \circ u_1^{-1}$ and $u_2 = 1$. Therefore $v_1 = 1$ - contradiction.

**Case 2.** Let $|u_2| > |v_1|$. Then $u_2 = u_{21} \circ v_1^{-1} = u_{21} \circ u_{22}$ and

$$vuv = (v_1 \circ u_{21}) u_1^{-1} = (u_{22}^{-1} \circ u_{21}) u_1^{-1}.$$

Hence $u_1^{-1} = u_{21}^{-1} \circ u_{22} \circ u_{11}^{-1}$,

$$v = v_1 \circ u_1^{-1} = u_{22}^{-1} \circ u_1^{-1} = u_{22}^{-1} \circ u_{21}^{-1} \circ u_{22} \circ u_{11}^{-1},$$

and $u = u_1 \circ u_2 = u_{11} \circ u_{22}^{-1} \circ u_{21}^2 \circ u_{22}$. This finishes the proof.

**Remark.** It may seem that $(ii)$ implies $(i)$. However the following example shows that this is not the case. Let $w \in F$, consider the following product $w^{-1} \cdot w^2$. Then $w^{-1}$ *does not* necessarily cancels out in $w^2$. Indeed, let $w = w_1^{-1} \circ \bar{w} \circ w_1$ where $w_1 \neq 1$, then $w^{-1} \cdot w^2 = w_1^{-1} \circ \bar{w} \circ w_1$ where $w_1^{-1}$ "belongs" to $w^{-1}$.

**Lemma 2** *Let $u, v \in F$. The left occurrence of $v$ cancels out in $vuv^{-1}$ if and only if $v$ takes one of the following forms:*

*(i)* $v = u_1^{-1}$ *where* $u = u_1 \circ u_2$*;*

*(ii)* $v = s_1^{-1} r u^{-n}$ *where* $u = r^{-1} \circ s \circ r$ *and* $s = s_1 \circ s_2$*-cyclically reduced.*

**Proof.** Suppose that $v$ doesn't cancel out in $vu$. Consider two cases.

**Case 1.** There is no cancellation in $uv^{-1}$. Then $v = v_1 \circ u^{-1}$, and $vuv^{-1} = v_1 \cdot u \circ v_1^{-1}$. By induction we get $v = w \circ u^{-n}$ and the left occurrence of $w$ cancels out in $wuw^{-1}$. So $u = u_1 \circ u_2$, $w = u_1^{-1}$ and $v = u_1^{-1} \circ u^{-n}$.

**Case 2.** Suppose that there is some cancellation in $uv^{-1}$. Then $u = u_1 \circ r$, $v^{-1} = r^{-1} v_2^{-1}$ such that $uv^{-1} = u_1 \circ v_2^{-1}$ and $r \neq 1$. Then $vuv^{-1} = (v_2 \circ r) \cdot (u_1 \circ v_2^{-1})$. Since $v_2 \circ r$ cancels out we have $|u_1| \geq |r|$. Therefore $u_1 = r^{-1} \circ s$ and $u = u_1 \circ r = r^{-1} \circ s \circ r$ and $vuv^{-1} = v_2(s \circ v_2^{-1})$. By the previous case $v_2 = s_1^{-1} \circ s^{-n}$ where $s = s_1 \circ s_2$, so $v = s_1^{-1} \circ s^{-n} \circ r$ and $u = r^{-1} \circ s \circ r$.

Notice that in case 1.) element $u$ is cyclically reduced and therefore case 1.) follows from case 2.) by taking $r = 1$.

**Lemma 3** *Let $c_1, c_2, c_3 \in F$, $\epsilon_1, \epsilon_2 \in \{1, -1\}$. If $v$ is a solution of a quadratic equation*

$$E(x) = c_1 x^{\epsilon_1} c_2 x^{\epsilon_2} c_3 = 1,$$

*then $v$ takes the following form:*

$$v = c \circ t,$$

*where $c \in F$ and $t$ is a pseudo-solution of a quadratic equation of type:*

$$E'(x) = x^{\delta_1} w x^{\delta_2},$$

*where $w \in F$, $\delta_1, \delta_2 \in \{1, -1\}$.*

**Proof** Let $v$ be a solution of $c_1 x^{\epsilon_1} c_2 x^{\epsilon_2} c_3 = 1$. Then $v$ is also a solution of $c_3 c_1 x^{\epsilon_1} c_2 x^{\epsilon_2} = 1$, which we rewrite as $c_4 x^{\epsilon_1} c_2 x^{\epsilon_2} = 1$, where $c_4 = c_3 c_1$. $v$ being a solution means that all occurrences of $v$ cancel out in $c_4 v^{\epsilon_1} c_2 v^{\epsilon_2}$. If $v^{\epsilon_1}$ cancels out, then $v^{\epsilon_1} = v_1 \circ v_2$, $v_1$ cancels out in $c_4 v_1$ and $v_2$ cancels out in $v_2 c_2 v^{\epsilon_2}$. Then $c_4 = c_{41} \circ c_{42}$, $v_1 = c_{42}^{-1}$. We have two possibilities for $\epsilon_2$:

a) $\epsilon_2 = \epsilon_1$, then $v_2$ cancels out in $v_2 c_2 c_{42}^{-1} \circ v_2$, or $v_2 c_5 v_2$, where $c_5 = c_2 c_{42}^{-1}$.

b) $\epsilon_2 = -\epsilon_1$, then $v_2$ cancels out in $v_2 c_2 v_2^{-1} \circ c_{42}$. Then either $v_2$ cancels out in $v_2 c_2 v_2^{-1}$ or $v_2^{-1}$ cancels out in $v_2 c_2 v_2^{-1}$.

□

**Proof of Theorem 3.** To prove the theorem we will consider all possible cases that $v$ can cancel out in $x^{\epsilon_1} u x^{\epsilon_2}, \epsilon_i \in \{1, -1\}$:

**Case 1.** The left occurrence of $v$ cancels out in $vuv$. Then $v$ takes one of the forms from lemma 1.

**Case 2.** The left occurrence of $v$ cancels out in $vuv^{-1}$. Then $v$ takes one of the forms from lemma 2.

**Case 3.** The left occurrence of $v$ cancels out in $v^{-1}uv^{-1}$. By replacing each occurrence of $v$ by $v^{-1}$ and applying lemma 1 we can show that the forms in this case are inverses of the forms from lemma 1.

**Case 4.** The left occurrence of $v$ cancels out in $v^{-1}uv$. Similarly, by replacing $v$ by $v^{-1}$, the forms are inverses of the forms from lemma 2.

**Case 5.** The right occurrence of $v$ cancels out in $v^{-1}uv^{-1}$. The forms can be obtained by replacing $u$ by $u^{-1}$ in lemma 1.

**Case 6.** The right occurrence of $v$ cancels out in $vuv$. By taking inverse of $vuv$ we can see that the forms can be obtained by replacing $u$ by $u^{-1}$ in case 3.

**Case 7.** The right occurrence of $v$ cancels out in $vuv^{-1}$. Replace $u$ by $u^{-1}$ in lemma 2.

**Case 8.** The right occurrence of $v$ cancels out in $v^{-1}uv$. Replace $u$ by $u^{-1}$ in case 4. □

### 4.3.3 Reduction of cubics to quadrics

**Proof of Theorem 2.** To prove the theorem we consider several cases. Cases 1 through 3 are the principal ones and the rest follow from them.

Observe first, that the result is obvious if the middle occurrence of $v$ does not cancel out in $E(v)$. Indeed, in this event either $v^{\epsilon_1}$ cancels out in $v^{\epsilon_1}u_1v^{\epsilon_2}$ or $v^{\epsilon_3}$ cancels out in $v^{\epsilon_2}u_2v^{\epsilon_3}$, and the result follows. So we may assume now that the middle occurrence of $v$ cancels out in $E(v)$.

**Case 1.** Assume $\epsilon_1 = -1, \epsilon_2 = 1, \epsilon_3 = -1$. Thus the middle occurrence of $v$ cancels out in $v^{-1}u_1vu_2v^{-1}$.

To prove this case it is enough to notice that the middle occurrence of $v$ cancels out in $v^{-1}u_1vu_2v^{-1}$ if and only if $v = v_1 \circ v_2$ such that $v_1$ cancels out in

$$v_2^{-1} \circ v_1^{-1} \cdot u_1v_1 \tag{4.4}$$

and $v_2$ cancels out in

$$v_2u_2(v_2^{-1} \circ v_1^{-1}). \tag{4.5}$$

If $v_1$ cancels out in (4.4) then one of the following holds:

a) $v_1$ cancels out in $v_1^{-1}u_1v_1$.

b) $v_1^{-1}$ cancels out in $v_1^{-1}u_1v_1$, since there is no cancellation between $v_2^{-1}$ and $v_1^{-1}$ in (4.4) and so in order for $v_1$ to cancel out, $v_1^{-1}$ must cancel out first.

Similarly if $v_2$ cancels out in (4.5) then either

a) $v_2$ cancels out in $v_2u_2v_2^{-1}$, or

b) $v_2^{-1}$ cancels out in $v_2u_2v_2^{-1}$.

**Case 2.** Assume $\epsilon_1 = 1, \epsilon_2 = 1, \epsilon_3 = 1$. Thus the middle occurrence of $v$ cancels out in $vu_1vu_2v$.

Then $v = v_1 \circ v_2$ such that the right occurrence of $v_1$ cancels out in $v_1 \circ v_2 \cdot u_1v_1$ and the left occurrence of $v_2$ cancels out in $v_2u_2 \cdot v_1 \circ v_2$.
Suppose that $v_1$ cancels out in $u_2 \cdot v_1$. Then $v_1 = u_{22}^{-1}$ and $u_2 = u_{21} \circ u_{22}$. Therefore the left occurrence of $v_2$ cancels out in $v_2u_{21}v_2$.

Similarly we consider the case when $v_2$ cancels out in $v_2u_1$.

So the left occurrence of $v_2$ cancels out in $v_2u_2 \cdot v_1 \circ v_2$ and $v_1$ doesn't cancel out in $u_2v_1$. Suppose that $v_2$ does not cancel out in $u_2v_1$. Then $v_2 = r \circ s$, $u_2v_1 = s^{-1} \circ t$, $r \neq 1$ and $v_2u_2 \cdot v_1 \circ v_2 = r \circ t \circ v_2$. Since $r \neq 1$ must cancel out this case is impossible.

Suppose that $u_2v_1$ cancels out in the left occurrence of $v_2$. Then $v_2 = r \circ s$, $u_2v_1 = s^{-1}$ and $v_2u_2 \cdot v_1 \circ v_2 = r \cdot r \circ s$. Then $r = 1$, which means this case is possible only if $v_2$ cancels out in $v_2u_2v_1$ - the only case left to consider. Now we have the following possibilities:

a) $v_2$ cancels out in $v_2u_2$. Then $v_2 = u_{21}^{-1}$ where $u_2 = u_{21} \circ u_{22}$. Therefore the right occurrence of $v_1$ cancels out in $v_1 \circ u_{21}^{-1} \cdot u_1 \cdot v_1$.

b) Suppose that $v_2$ does not cancel out in $v_2u_2$. Since $v_2$ cancels out in $v_2u_2v_1$ we have $u_2 = r \circ s$, $v_2 = y \circ r^{-1}$, $y \neq 1$ and $v_1 = s^{-1} \circ z$. Then $y$ cancels out in $v_2u_2v_1 = yz$ and $z = y^{-1} \circ z_1$. Since the right occurrence of $v_1$ cancels out in $v_1 \circ v_2 \cdot u_1v_1$ and $v_2$ doesn't cancels out in $v_2u_1$ (we already considered this

case ) we may assume that $v_1$ cancels out in $v_2u_1v_1$. We have now $v_2u_1v_1 = y \circ r^{-1} \cdot u_1 \cdot s^{-1} \circ y^{-1} \circ z_1$ where $s^{-1} \circ y^{-1} \circ z_1$ cancels out. Therefore $y^{-1}$ cancels out in $y \circ r^{-1} \cdot u_1 \cdot s^{-1} \circ y^{-1}$ taking the inverse we get $y \cdot t \cdot y^{-1}$ where $t = su_1^{-1}r$ and the left occurrence of $y$ cancels out. Notice that $r^{-1}u_1s^{-1} \neq 1$, indeed otherwise $z_1 = 1$ and $v = v_1 \circ v_2 = s^{-1} \circ y^{-1} \circ y \circ r^{-1}$, therefore $y = 1$ - contradiction.

So $v_1$ and $v_2$ take the form $v_i = c_1yc_2$, where $c_1, c_2 \in F$ and $y$ cancels out in $y \cdot t \cdot y^{-1}$. By applying lemma 2 we can see that each $v_i$ can be written in the form $ab^nc$, $a, b, c \in F$. Then $v_i$ is a solution of quadratic equation $[a^{-1}xc^{-1}, b] = 1$. Applying lemma 3 finishes the proof of b).

**Case 3.** Assume $\epsilon_1 = -1, \epsilon_2 = 1, \epsilon_3 = 1$. Thus the middle occurrence of $v$ cancels out in $v^{-1}u_1vu_2v$.

Let $v = v_1 \circ v_2$. Then the right occurrence of $v_1$ cancels out in

$$v_2^{-1} \circ v_1^{-1} \cdot u_1v_1 \tag{4.6}$$

and the left occurrence of $v_2$ cancels out in

$$v_2u_2 \cdot v_1 \circ v_2. \tag{4.7}$$

As in case 1., if $v_1$ cancels out in (4.6) then one of the following holds:

1) $v_1$ cancels out in $v_1^{-1}u_1v_1$, or

2) $v_1^{-1}$ cancels out in $v_1^{-1}u_1v_1$.

We consider 1) and 2) separately:

**1)** Notice that this case is the symmetrical (reading from right to left) version of lemma 2. So, by repeating proof of lemma 2 we have that $v_1$ takes one of the following forms:

a) $v_1 = u_{12}^{-1}$ where $u_1 = u_{11} \circ u_{12}$;

b) $v_1 = u_1^n r^{-1} s_2^{-1}$ where $u = r^{-1} \circ s \circ r$ and $s = s_1 \circ s_2$-cyclically reduced. Or we can write $v_1$ in a form without cancellation: $v_1 = r^{-1} \circ s^n \circ s_2^{-1}$ - also found in the proof of lemma 2.

In case a) the left occurrence of $v_2$ cancels out in $v_2 u_2 u_{12}^{-1} \circ v_2$. Let $u_3 = u_2 u_{12}^{-1}$, then $v_2$ cancels out in $v_2 u_3 v_2$.

In case b) the left occurrence of $v_2$ cancels out in $v_2 u_2 r^{-1} \circ s^n \circ s_2^{-1} \circ v_2$.

Let us assume that $r^{-1} \circ s^n \circ s_2^{-1}$ cancels out in $u_2 r^{-1} \circ s^n \circ s_2^{-1}$. Then $u_2 = u_{21} \circ u_{22}$, $r^{-1} \circ s^n \circ s_2^{-1} = u_{22}^{-1}$ and $v_2 u_2 r^{-1} \circ s^n \circ s_2^{-1} \circ v_2 = v_2 u_{21} v_2$. So, $v_2$ cancels out in $v_2 u_{21} v_2$.

If $r^{-1} \circ s^n \circ s_2^{-1}$ does not cancel out in $u_2 \cdot r^{-1} \circ s^n \circ s_2^{-1}$, then we consider two possibilities:

a) Let $v_2$ cancel out in $u_2 r^{-1} \circ s^n \circ s_2^{-1}$. Then $u_2 r^{-1} \circ s^n \circ s_2^{-1} = t_1 \circ t_2$ and $v_2 = t_1^{-1}$.

So, $v_2$ is in the form, which is a particular case of $ab^n c$, where $a, b, c \in F$. Hence, $v_2$ is a solution of quadratic equation $[a^{-1} x c^{-1}, b] = 1$. Applying lemma 3 finishes the proof of a).

b) Suppose $v_2$ does not cancel out in $u_2 r^{-1} \circ s^n \circ s_2^{-1}$. Then $u_2 r^{-1} \circ s^n \circ s_2^{-1} = u_{21} \circ u_{22}$, $v_2 = v_{21} \circ u_{21}^{-1}$, $v_{21} \neq 1$ and $v_2 u_2 r^{-1} \circ s^n \circ s_2^{-1} \circ v_2 = v_{21} \circ u_{22} \circ v_2$. But $v_{21}$ must cancel out, so we have contradiction.

**2)** By letting $v_1' = v_1^{-1}$ we again have a case of lemma 2. Hence we can prove case 2) by following the proof of case 1) step by step.

**Case 4.** The middle occurrence of $v$ cancels out in $vu_1vu_2v^{-1}$.

It is clear that $v$ cancels out in a quasi-equation if and only if $v$ cancels out in its inverse. The inverse of $vu_1vu_2v^{-1}$ is $vu_2^{-1}v^{-1}u_1^{-1}v^{-1}$. Let $z = v^{-1}$ and replace each occurrence of $v$ by $z$. We will have $z^{-1}u_2^{-1}zu_1^{-1}z$. Now we can apply case 3 and find

quadratic equations for $z$. Since $z = v^{-1}$, inverting this equations will give us the quadratic equations for $v$.

**Case 5.** The middle occurrence of $v$ cancels out in $v^{-1}u_1v^{-1}u_2v^{-1}$.

Similarly, taking the inverse of $v^{-1}u_1v^{-1}u_2v^{-1}$ we have $vu_2^{-}1vu_1^{-1}v$. Now the set of quadratic equations can be obtained by applying case 2.

**Case 6.** The middle occurrence of $v$ cancels out in $v^{-1}u_1v^{-1}u_2v$.

The inverse of $v^{-1}u_1v^{-1}u_2v$ is $v^{-1}u_2^{-1}vu_1^{-1}v$. Apply case 3.

**Case 7.** The middle occurrence of $v$ cancels out in $vu_1v^{-1}u_2v^{-1}$.

Let $z = v^{-1}$. Replacing $v$ by $z$ results in $z^{-1}u_1zu_2z$. Now the equations are the inverses of the equations from case 3.

**Case 8.** The middle occurrence of $v$ cancels out in $vu_1v^{-1}u_2v$.

As in case 7, let $z = v^{-1}$. Replacing $v$ by $z$ results in $z^{-1}u_1zu_2z^{-1}$. The equations are the inverses of the equations from case 1.

This finishes the proof.

## 4.4 Description of solutions of one-variable equations over $F$

Appel [37] and Lorents [45] gave the exact form of the parametric words, which describe solutions of one-variable equations over $F$.

Now we show how one can derive their description from the results above. For the sake of completeness we provide a few definitions from language theory and a simple lemma from [36].

A language is bounded if it is a subset of $w_1^* \cdots w_n^*$ for some $w_i \in \Sigma^*$. Lemma 4 shows that bounded context–free languages are related to semi–linear subsets of $N^n$, the set of $n$–tuples of non–negative integers. A semi–linear set is a finite union of linear sets, and a linear set is one of the form $\vec{m} + M$ for $\vec{m} \in N^n$ and $M$ a finitely

generated submonoid of $N^n$. Semi–linear sets are closed under intersection, monoid homomorphism from $N^n$ to $N^m$, and Cartesian product [43]. The last part means that if $T \subset N^n$ and $T' \subset N^m$ are semi–linear, so is $T \times T' \subset N^{n+m}$.

**Lemma 4** *If $L \subset \Sigma^*$ is context–free and $\{u_i, v_i \; 1 \leq i \leq n\} \subset \Sigma^*$, then $\mathcal{J} = \{(j_1, \ldots, j_n) \mid u_1 v_1^{j_1} \cdots u_n v_n^{j_n} \in L\}$ is semi–linear.*

*Proof.* $L' = L \cap u_1^* v_1^* \cdots u_n^* v_n^*$ is the intersection of a context–free language with a regular one and so is context–free. Define

$$\mathcal{I} = \{(i_1, j_1, \ldots, i_n, j_n) \mid u_1^{i_1} v_1^{j_1} \cdots u_n^{i_n} v_n^{j_n} \in L'\}.$$

It suffices to show that $\mathcal{I}$ is semi–linear as $\mathcal{J}$ can be recovered from $\mathcal{I}$ by operations which preserve semi–linearity.

Let $\Delta = \{b_i, c_i \mid 1 \leq i \leq n\}$ be a new alphabet. By [43, Lemma 2.6] $L'' = \{b_1^{i_1} c_1^{i_1} \cdots b_n^{i_n} c_n^{j_n} \mid (i_1, j_1, \ldots, i_n, j_n) \in \mathcal{I}\}$ is context–free, and Parikh's Theorem [38, Section 2.4] applied to $L''$ says that $\mathcal{I}$ is semi–linear. □

**Corollary 1** *Let*

$$E(x) = u_1 x^{\epsilon_1} \cdots u_d x^{\epsilon_d} = 1$$

*be a reduced one-variable equation over $F$. Then the solution set of $E(x) = 1$ in $F$ is a finite union of sets of the form*

$$\{p_1^i p_2^j q \mid (i, j) \in S\}$$

*where the $p_1, p_2, q \in F$ and $S$ is a semilinear subset of $N^2$.*

## 4.5 Description and Complexity of the Algorithm

Let

$$E(x) = u_1 x^{\epsilon_1} u_2 x^{\epsilon_2} \dots u_d x^{\epsilon_d} = 1 \tag{4.8}$$

be an one-variable equation *of degree* $d$ over a free group $F$. As usual we assume that all constants $u_i \in F$ are freely reduced. Clearly, up to conjugation every one-variable equation over $F$ can be written in the form (4.8). The *length* of the equation $E(x)$ is defined by:

$$l = |E(x)| = \sum_{i=1}^{d} |u_i| + d.$$

To find all solutions of $E(x) = 1$ we perform the following steps:

**Algorithm 1** *[Finding all solutions of $E(x)$]*

**1)** *Enumerate all subwords of $E(x)$ of the type*

$$x^{\epsilon_{i-1}} u_i x^{\epsilon_i} u_{i+1} x^{\epsilon_{i+1}} \tag{4.9}$$

*including also subwords $x^{\epsilon_d} u_1 x^{\epsilon_1} u_2 x^{\epsilon_2}$ and $x^{\epsilon_{d-1}} u_d x^{\epsilon_d} u_1 x^{\epsilon_1}$.*

**2)** *For every cubic equation (4.9) find the quadratic equations*

$$x^{\delta_{i-1}} w_i x^{\delta_i} \tag{4.10}$$

*as described in Theorem 2.*

**3)** *For every quadratic equation (4.10) find all its pseudo-solutions in the form*

$$ab^n c, \tag{4.11}$$

*as described in Theorem 3.*

**4)** *Use pseudo-solutions from (4.11) to describe all pseudo-solutions of the cubic equations (4.9) in the form*

$$ab^{n_1}cd^{n_2}e, \tag{4.12}$$

*where* $a, b, c, d, e \in F, n_1, n_2 \in \mathbb{N}$.

**5)** *Determine (as in Lemma 9 below) for which values* $n_1$, $n_2$ *the pseudo-solutions (4.12) are solutions of the equation* $E(x) = 1$.

**Remark.** We assume above that $d \geq 3$. Indeed, the case $d = 1$ is trivial. If $d = 2$ then Lemma 3 reduces the situation to the quadratic equations 4.11, in which case one can proceed directly to the step 3) of the algorithm.

**Theorem 4** *[Correctness and time-complexity of Algorithm 1] Algorithm 1 computes all solutions of the one-variable equation (4.8) in* $O(d^5 \cdot l^4)$ *time.*

We start the proof of the theorem with the following lemmas.

**Lemma 5** *Let the left occurrence of* $v$ *cancels out in* $vuv$*, where* $|u| \leq l$*, then the number of possible forms* $ab^nc$ *of type (4.11) that* $v$ *takes is* $O(l)$*, for each such form* $|a|, |b|, |c| \leq l$*, and they can be computed in* $O(l^2)$ *time.*

**Proof.** According to lemma 1 $v$ is one of the following:

(i) $v = u_1^{-1}$ where $u = u_1 \circ u_2$, or

(ii) $v = u_2^{-1}u_1^{-1}$ where $u = u_1 \circ u_2^2$.

In case (i) the number of initial segments of $u$ is $O(l)$. Each such initial segment has length $O(l)$, so to write down all of them would take $O(l^2)$ operations.

In case (ii) we have $O(l)$ possibilities for $u_1$. For each such possibility we need to check if the rest of $u$ is a square. This can also be done in linear time. Indeed, let $w$ be a square, then w in a freely reduce form will look like $w_1 \circ w_2 \circ w_2 \circ w_1^{-1}$.

Then we can find $w_1$ in linear time by repeatedly comparing and removing letters from both ends of $w$. What's left is $w' = w_2 \circ w_2$, which can be checked by putting two pointers: one at the beginning of $w'$, the other at its half-length, and then comparing the corresponding letters and simultaneously moving both pointers to the right. This also takes $O(l)$ time. So, we do $O(l)$ operations for $O(l)$ possibilities for $u_1$. Then the total work is $O(l^2)$.

**Lemma 6** *Let the left occurrence of $v$ cancels out in $vuv^{-1}$, where $|u| \leq l$, then the number of possible forms $ab^nc$ of type (4.11) that $v$ takes is $O(l)$, for each such form $|a|, |b|, |c| \leq l$, and they can be computed in $O(l^2)$ time.*

**Proof.** According to lemma 2 $v$ is one of the following:

(i) $v = u_1^{-1}$ where $u = u_1 \circ u_2$;

(ii) $v = s_1^{-1} r u^{-n}$ where $u = r^{-1} \circ s \circ r$ and $s = s_1 \circ s_2$-cyclically reduced.

In case (i) the number of initial segments of $u$ is $O(l)$. Each such initial segment has length $O(l)$, so to write down all of them would take $O(l^2)$ operations.

In case (ii) it takes $O(l)$ time to find $r$. Then we have $O(l)$ possible choices for $s_1$. For each choice of $s_1$ we write down $v$, which again takes $O(l)$ time, so the total work is $O(l^2)$.

The following lemma is a known result for free groups and included here without proof:

**Lemma 7** *Let $G$ be a free group. Then for any sequence of elements $u_1, \ldots, u_k, g \in G$ if the equality*

$$u_1^n u_2^n \ldots u_k^n = g \tag{4.13}$$

*holds for infinitely many integers $n$ then:*

*a)* $g = 1$*;*

*b) there exist* $i, j$*, where* $1 \leq i < j \leq k$*, such that*

$$u_i \dots u_j = 1 \quad and \quad [u_s, u_t] = 1 \; for \; any \; s, t \in \{i, \dots, j\}. \tag{4.14}$$

*(we call such* $u_i \dots u_j$ *a* cancellation segment*);*

*c) the equality (4.13) holds for all integers* $n$*.*

**Lemma 8** *Let* $|a|, |b|, |c| \leq l$*. Then:*

*1) One can effectively decide in* $O(d^3 l)$ *steps whether* $E(ab^n c) = 1$ *for infinitely many* $n \in \mathbb{Z}$ *or not.*

*2) If* $E(ab^n c) = 1$ *only for finitely many* $n$ *then all such* $n$ *are bounded by* $K|E(x)|$*, where* $K \leq 3ld$*.*

*Proof.* Substitute $ab^n c$ into $E(x) = 1$ and rewrite $E(ab^n c)$ in the conjugacy-power form:

$$\tilde{S}(x) = (b^{\bar{u}_1^{-1}})^{\varepsilon_1 n} (b^{\bar{u}_2^{-1} \bar{u}_1^{-1}})^{\varepsilon_2 n} \dots (b^{\bar{u}_d^{-1} \dots \bar{u}_1^{-1}})^{\varepsilon_d n} \bar{u}_1 \dots \bar{u}_d = 1. \tag{4.15}$$

where $\bar{u}_i$ are obtained from $u_i$ and $a, b$ in rewriting $E(ab^n c)$. This takes $O(ld) + O(l) = O(ld)$ steps. By Lemma 7 it must be a cancellation segment in (4.15). If here are no such segment then $ab^n c$ is not a solution for infinitely many $n$. Otherwise, delete the cancellation segment from (4.15) and repeat. This allows one to decide 1).

According to Lemma 7 cancellation segment $u_i \dots u_j$ satisfies

$$u_i \dots u_j = 1 \quad and \quad [u_s, u_t] = 1 \; for \; any \; s, t \in \{i, \dots, j\}.$$

To find a cancellation segment we start with $u_1$, multiply it on the right by $u_2$ and check for identity, then multiply by $u_3$ and so on. We do this $O(d)$ times while

cancelling no more than $O(l)$ letters on each step. Then repeat this process starting with $u_2$, then $u_3$, and so on, $O(d)$ times in total. The complexity of this process is $O(d^2l)$. All possible commutators can be precomputed beforehand, which is $O(d^2)$ commutators in $O(d^2l)$ steps. Then it is enough to look up $O(d)$ commutators on each step: $[u_1, u_2], [u_2, u_3], \ldots, [u_{d-1}, u_d]$. Therefore, finding a cancellation segment takes $O(d^2l)+O(d^2(l+d)) = O(d^2l)$ steps. We can remove at most $O(d)$ cancellation segments, so 1) takes $O(d^3l)$ steps.

To prove 2) notice that if $ab^nc$ is not a solution of $E(x) = 1$ for infinitely many $n$ then in the process of deleting the cancellation segments described in 1) one arrives to the point where there are no cancellation segments in (4.15). In this event one may assume that in (4.15) all the neighbors

$$(b^{\bar{u}_i^{-1}\ldots\bar{u}_1^{-1}})^{\varepsilon_i n}, \quad (b^{\bar{u}_{i+1}^{-1}\ldots\bar{u}_1^{-1}})^{\varepsilon_{i+1} n}$$

do not commute. Notice, that one can also assume that $b$ is cyclically reduced (otherwise one can change $a, b, c$ accordingly). Observe that in the product

$$(b^{\bar{u}_i^{-1}\ldots\bar{u}_1^{-1}})^{\varepsilon_i n}(b^{\bar{u}_{i+1}^{-1}\ldots\bar{u}_1^{-1}})^{\varepsilon_{i+1} n} \tag{4.16}$$

the cancellation cannot be longer than $|\bar{u}_{i+1}| + 2|b|$. Indeed, in this case $b$ would commute with $\bar{u}_{i+1}$. So if $n > 2(|\bar{u}_{i+1}| + 2|b|)$ then the powers $b^n$ do not cancel completely in (4.16). Therefore, if

$$n|b| > \max\{2(|\bar{u}_{i+1}| + 2|b|) \mid i = 1, \ldots, d\} + |\bar{u}_1 \ldots \bar{u}_d| \tag{4.17}$$

then $E(ab^nc) \neq 1$.

Notice that by Lemmas 6 and 5 one has $|\bar{u}_i| \leq 3l$ and also $|\bar{u}_1 \ldots \bar{u}_d| \leq 3ld$. So if $n \geq 3ld$ then $E(ab^nc) \neq 1$. For better performance the algorithm computes estimate

(4.17) for each case of $ab^n c$.

**Lemma 9** *Let $|a|, |b|, |c|, |d|, |e| \leq l$. Then:*

*1) One can effectively find in $O(d^4 l^2)$ steps all integers $n_1$ for which $E(ab^{n_1}cd^{n_2}e) = 1$ for infinitely many $n_2 \in \mathbb{Z}$.*

*1) One can effectively find in $O(d^4 l^2)$ steps all integers $n_2$ for which $E(ab^{n_1}cd^{n_2}e) = 1$ for infinitely many $n_1 \in \mathbb{Z}$.*

*3) Let $n_1$ be an integer such that $E(ab^{n_1}cd^{n_2}e) = 1$ only for finitely many $n_2 \in \mathbb{Z}$ and $n_2$ be an integer such that $E(ab^{n_1}cd^{n_2}e) = 1$ only for finitely many $n_1 \in \mathbb{Z}$. If $E(ab^{n_1}cd^{n_2}e) = 1$ then $|n_1|, |n_2| \leq O(|E(x)|)$ and this bound can be effectively found.*

*Proof.* To prove 1) we repeat the argument from Lemma 8. To this end rewrite $ab^{n_1}cd^{n_2}e$ in the form $a_1 d^{n_2} e$ where $a_1 = ab^{n_1}c$. Then one can write $E(a_1 d^{n_2} e)$ in the power-conjugacy form as in (4.15) and repeat the cancellation process described in Lemma 8. In this case to find a cancellation segment one has to solve the corresponding commutation equation which involves the exponent $n_1$. Namely,

$$[(b^{\bar{u}_i^{-1} \ldots \bar{u}_1^{-1}})^{\varepsilon_i n}, \quad (b^{\bar{u}_{i+1}^{-1} \ldots \bar{u}_1^{-1}})^{\varepsilon_{i+1} n}] = 1, \tag{4.18}$$

where $a$ is replaced by $a_1$ (and $b$ by $d$, and $c$ by $e$). This implies

$$[b, \bar{u}_{i+1}] = 1 \tag{4.19}$$

so by Lemma 8 either $n_1 \leq O(l)$ or commutation holds for arbitrary $n_1$. In the former case we got a bound on $n_1$ and in the latter one we can continue and look for another commutation segment. Notice, that this process will stop and at some point we will get that some equation of the type (4.19) have only finitely many

solutions for $n_1$, otherwise $E(x) = 1$ would have solutions $ab^{n_1}cd^{n_2}e$ for all $n_2$ and $n_1$ - impossible.

The process is organized as follows. We enumerate $d-1$ neighboring commutators of type (4.18). For each one we solve the corresponding equation of type (4.19) in $O(d^3 l)$ steps by applying Lemma 8. In case that $n_1 \leq O(l)$ we enumerate all possible values for $n_1$, substitute each of them into the power-conjugacy form of type (4.15) and continue working with the resulting forms as in Lemma 8. The total work is $(O(d^3 l) + O(d^3 l)O(l))(d-1) = O(d^4 l^2)$.

2) is similar to 1) and 3) is similar to Lemma 8.

**Proof of Theorem 4.**

Step 1. of the algorithm enumerates $d$ cyclical subwords of type (4.9). According to theorem 1 we can find all solutions of $E(x) = 1$ by enumerating all pseudo-solutions of these subwords. Notice that for each occurrence of $x$ in $E(x) = 1$ there is a subword of type (4.9), which contains that $x$ in the middle. Then it is enough to consider only the cases where the middle occurrence of $x$ cancels out.

Step 2. According to theorem 2 if $v$ is a pseudo-solution of one of the subwords of type (4.9), then $v = v_1 \circ v_2$ and each $v_i$ can we written as $c \circ t$, where $t$ is a pseudo-solution of a quadratic equation of type (4.10). Notice that in the proof of the theorem there are only finite number of quadratic equations of type (4.10) that need to be considered both for $v_1$ and $v_2$. Also notice that $|c| \leq l$ and $|w_i| \leq l$.

On step 3. we find the pseudo-solutions for $v_1$ and $v_2$ in the form (4.11). We can do this since the proof of theorem 3 reduces all cases to two principal ones, which we considered in lemmas 5 and 6. According to these lemmas there are $O(l)$ possibilities for each $v_1$ and $v_2$ and to compute those we need $O(l^2)$ operations.

On step 4. we concatenate forms for $v_1$ and $v_2$ and construct all possible forms for $v = v_1 \circ v_2$ of type (4.12). There are $O(l)$ possibilities for each $v_1$ and $v_2$, so the number of forms for $v$ is $O(l^2)$. The length of each form is $O(l)$, so we need $O(l^3)$

operations to write them down. The total work consists of computing forms for $v_1$, $v_2$, and $v$ and is $O(l^2) + O(l^2) + O(l^3) = O(l^3)$ operations.

Step 5. For each form of type $ab^{n_1}cd^{n_2}e$ we apply the procedure described in Lemma 9 in the case that both $n_1$ and $n_2$ are not equal to zero. If one of them is zero then we can save some work by applying Lemma 8. In any case the total work for each form is no more than $O(d^4l^2)$ operations. We have $O(l^2)$ such forms to check for each of $d$ cyclical subwords, so the total complexity of the algorithm is $O(d^4l^2) \cdot O(l^2) \cdot d = O(d^5 \cdot l^4)$.

# Chapter 5

# Quadratic equations and the genus problem

## 5.1 Introduction

Quadratic equations, i.e. equations where every variable occurs at most twice, deserve a special interest. Comerford and Edmunds [31, 30] found an algorithm for solving quadratic equations and described all solutions. Their algorithm is at least exponential but is much better than the general Makanin's algorithm. It is also known that the quadratic case is the principal one: Kharlampovich and Myasnikov [33] showed that the quadratic case is the principal one for equations over free groups, i.e. the problem of solving an arbitrary equation over a free group can be reduced to solving a finitely many quadratic equations.

In this chapter we demonstrate how genetic algorithms can be used in solving an open problem, namely the so-called *Genus problem*. We do so by reducing a quadratic equation for the Genus problem to a system of many equations in a free semigroup and solving the system by a specially designed genetic algorithm.

## 5.2 Problem statement and known results

The genus problem is important because of its applications in topology and logic. The problem can be stated as follows. Let $f$ be an element from the derived subgroup of a free group $F$. Genus of $f$ is the minimal number of commutators, say $n$, such that $f$ can be expressed as a product of $n$ commutators. Let us consider the quadratic equation

$$x_1^2 x_2^2 x_3^2 x_4^2 = 1$$

where $x_1, x_2, x_3, x_4$ are variables. *Genus* of a solution

$$x_1 = u_1, x_2 = u_2, x_3 = u_3, x_4 = u_4 \quad (u_i \in F)$$

is the genus of the product $u_1 u_2 u_3 u_4$. Genus of the equation is the supremum of the genuses of all its solutions. What is the genus of the equation?

For a long time, the only known example were ones of genus 1. J. Comerford and Y. Lee [34] were the first to find a solution of genus 2 and prove it. D. Spellman came up with a possible example of the equation of genus 2. The program described here has been used to check this example and confirmed his hypothesis. Until now it was very difficult or sometimes impossible to construct or check a possible solution of genus 2. Genetic algorithm provided a quick and simple (in real time) way to express a given word as a product of two commutators, if it can be expressed. There's also a direct non-genetic algorithm which is much slower but can say "no" when the word cannot be expressed.

## 5.3 Reduction to free semigroups

So we need to solve the equation

$$[y1, y2][y3, y4] = C,$$

where $C$ has form $x_1x_2x_3x_4$ as discussed above. Since the word $C$ can sometimes be very long and in order to speed up further tests for solving problems for genus 3,4,etc, it preferable to reduce solving the equation in a free group to solving a number of equations over a free semigroup. This is possible because of the result obtained by J.Comerford, L.Comerford and C.Edmunds [35]:

**Theorem** The equation $[y1, y2][y3, y4] = C$ has a solution if and only if there is a cancellation-free solution to an equation $W'' = C_0$ where $C_0$ is a cyclically reduced conjugate of $C$ and $W''$ is one of the following:

$$tpqr^{-1}p^{-1}sq^{-1}rs^{-1}t^{-1}uxy^{-1}u^{-1}zx^{-1}yz^{-1}, \tag{5.1}$$

$$pqr^{-1}p^{-1}uxy^{-1}u^{-1}tsq^{-1}rs^{-1}zx^{-1}yz^{-1}t^{-1}, \tag{5.2}$$

$$pqr^{-1}p^{-1}uxy^{-1}u^{-1}tzx^{-1}yz^{-1}sq^{-1}rs^{-1}t^{-1}, \tag{5.3}$$

$$pqr^{-1}p^{-1}tysq^{-1}rs^{-1}z^{-1}xt^{-1}uzy^{-1}x^{-1}u^{-1}, \tag{5.4}$$

$$pqr^{-1}p^{-1}tyz^{-1}xt^{-1}uzsq^{-1}rs^{-1}y^{-1}x^{-1}u^{-1}, \tag{5.5}$$

$$sxyzs^{-1}ptx^{-1}z^{-1}u^{-1}q^{-1}p^{-1}r^{-1}uy^{-1}t^{-1}qr, \tag{5.6}$$

$$sxyzs^{-1}pquy^{-1}t^{-1}p^{-1}r^{-1}q^{-1}tx^{-1}z^{-1}u^{-1}r, \tag{5.7}$$

$$sxyzs^{-1}r^{-1}uy^{-1}t^{-1}qrptx^{-1}z^{-1}u^{-1}q^{-1}p^{-1}, \tag{5.8}$$

$$pqrx^{-1}u^{-1}z^{-1}q^{-1}ys^{-1}r^{-1}zt^{-1}y^{-1}p^{-1}xstu, \tag{5.9}$$

At this point in order to try resolving the genus problem one need an efficient

algorithm for solving equations in a free semigroup. We use the algorithm described in Section 3.3.

## 5.4 Experimental results

### 5.4.1 An example of genus 2

This example comes from J. Comerford and Y. Lee [34]. Let

$x_1 = aBA$,

$x_2 = abAABAbaaBA$,

$x_3 = abABA$,

$x_4 = aaabA$

Then $x_1^2 x_2^2 x_3^2 x_4^2 = 1$ and $C = x_1 x_2 x_3 x_4$ is not a commutator.
Performing the substitution will give us

$$C = aBAabAABAbaaBAabABAaaabA$$

As computed by the genetic algorithm it can be expressed as a product of two commutators:

$$[ABabaBaba, ABabABAba]\,[ABAba, AB]$$

The commutators computed by genetic algorithm are different from those found by J. Comerford and Y. Lee. In fact, it is important to notice that because of the non-deterministic nature of the genetic algorithm one may get different solutions each time the algorithm is run. Since the constant $C$ above is not very long, the algorithm was able to find a solution in about 1 second on a PentiumII machine.

### 5.4.2 Another possible example of genus 2

In the example produced by D. Spellman

$x_1 = a,$

$x_2 = Bab,$

$x_3 = aaBaabbAABAAbAA,$

$x_4 = aaBaabaaBaaBAAbAABAAbAA$

Then $x_1^2 x_2^2 x_3^2 x_4^2 = 1$ and $C = x_1 x_2 x_3 x_4$ is not a commutator.

Performing the substitution will give us

$$C = ab^{-1}aba^2b^{-1}a^2b^2a^{-2}b^{-1}a^{-2}ba^{-2}a^2b^{-1}a^2ba^2b^{-1}a^2b^{-1}a^{-2}ba^{-2}b^{-1}a^{-2}ba^{-2}$$

As computed by the genetic algorithm it can be expressed as a product of two commutators:

$$[a^3b^{-1}a^2ba^2b^{-1}aba^{-1}, \quad a^3b^{-1}a^2ba^2b^{-1}a^{-1}ba^{-2}b^{-1}a^{-2}ba^{-3}]$$

and

$$[a^{-1}, aba^{-2}b^{-1}a^{-2}ba^{-2}]$$

There is more than one solution for this example. The genetic algorithm needs about 5-7 minutes to produce one of them.

# Part III

# Pattern recognition methods in groups and applications to cryptography

# Chapter 6

# Pattern recognition methods and primitive elements in $F_2$

## 6.1 Introduction

Let $F_n = F(x_1, ..., x_n)$ be a free group of rank $n$ with basis $\{x_1, ..., x_n\}$. An element $w \in F_n$ is called *primitive* if it is a member of some free basis of $F_n$, i.e., if there exists an automorphism of $F_n$ that takes $w$ to $x_1$.

Much is known about primitive elements in $F_2$. There are various necessary conditions on primitivity, which tell one on how primitive elements in $F_2$ should look like [49]. However, these conditions are far from being sufficient.

The classical way to determine whether an element $w \in F_n$ is primitive or not is to run the Whitehead's algorithm on $w$ [50]. This algorithm has quadratic time complexity in the length of the input word $w$. However, it depends exponentially on the rank $n$ of the free group $F_n$. For $F_n$ with $n > 5$ the algorithm becomes unfeasible. An alternative approach is to run a genetic version $GW$ of the Whitehead's algorithm [51]. The algorithm $GW$ is very efficient in practice even for large ranks, but it has its own drawback of being an "one-way" algorithm: given a primitive element $w$

$GW$ often terminates quickly providing a proof that $w$ is, indeed, primitive. On the other hand, if $GW$ does not terminate quickly, there is no guarantee that the element $w$ is not primitive.

In this chapter we describe a pattern recognition system that recognizes primitive elements in $F_2$. The corresponding classification algorithm (the *classifier*) has linear time complexity and reasonably high accuracy (88%). However, our purpose to design this system was quite different. We use the system as a research tool in experimenting with primitive elements in $F_2$, trying to find some hidden patterns in the reduced words representing these elements. It turns out that this approach provides valuable insights on the structure of primitive and non-primitive elements.

In Section 6.2 we describe the basic pattern recognition system. In Section 6.3 an *iterative clustering procedure* is introduced which allows one to uncover various clusters and relate them to the corresponding properties (*features*) of the elements. The following two sections demonstrate how this method can be applied to the set of primitive elements. In Section 6.4 we evaluate performance of the classifier and in the last Section 6.5 we show that analysis of the most significant features immediately leads one to the known results on free groups: Whitehead's cut-point theorem and the Nielsen's test on primitivity.

## 6.2 Pattern recognition system

We refer to Section 2.2 for all basic definitions, notions, and notations concerning pattern recognition (**PR**) systems.

Our basic (**PR**) system is a natural adjustment (to the case of primitive elements) of the **PR** system described in Section 2.2. More precisely, the

**Features.** We use the feature vectors of the type defined in Section 2.2.2. For

example, the feature vector

$$f(w) = \frac{1}{|w|} < C(w,v) \mid |v| = 2 >$$

shows the numbers of occurrences of words of length two in $w$ relative to the length of $w$. We also use other counting functions in our feature vectors, such as C(w,y*z), C(w,y**z), C(w,y***z), where $y$ and $z$ range over all possible letters from $X^{\pm 1} = \{x_1, ..., x_n, x_1^{-1}, ..., x_n^{-1}\}$ and * stands for an arbitrary letter from $X^{\pm 1}$.

**Model.** Our model is based on the linear regression classifier described in Section 2.2.3. Namely, for a word $w$ with a feature vector $f(w)$ we compute first the discriminant function

$$\hat{P}(w) = \beta f(w)$$

where $\beta$ is the vector of regression coefficients obtained from the training data set. The decision rule is based on the equal interval quantizing method described in Section 2.2.3.

We denote the obtained **PR** system by $P_f$ and the corresponding classifier by $R_{f,\beta}$.

Sometimes we modify the system $P_f$, to accommodate the quadratic regression model, as follows. For a given feature vector-function $f =< f_1, \ldots, f_n >$ define a new feature vector-function $f'$ by adding to $f$ all pair-wise products of components of $f$:

$$f' =< f_1, \ldots, f_n, f_1 f_1, f_1 f_2, \ldots, f_n f_n > .$$

Then the linear regression **PR** system $P_{f'}$ is equivalent to the system based on the corresponding quadratic regression model.

**Evaluation.** To evaluate the performance of the the classifier $R_{f,\beta}$ we use the accuracy measure described in Section 2.2.3.

**Training data set** We generate the training data set $D$ for the system $P_f$ as follows.

Roughly speaking, we start with a set of short words, primitive and non-primitive, and apply to them a sequence of Whitehead automorphisms chosen at random until the prescribed length is achieved. We also cyclically reduce the words after each application of an automorphism. More precisely, our training set $D$ consists of 10000 elements, where one half is primitive and the other is not primitive. The element's length varies from 5 to 1005.

To assure that all lengths are represented we divide $D$ into 50 bins $B_1 \dots B_{50}$, each containing 100 primitive elements and 100 non-primitive elements, such that $B_1$ contains only elements of length from 5 to 24, $B_2$ - from 25 to 44, and so on.

Here are the algorithms for generating $D$:

**Algorithm 1. Generate primitive part.**

`Choose a letter from the basis at random and assign it to` $w$.

`Repeat until all bins are full:`

1. `Apply a randomly chosen Whitehead automorphism to` $w$.
2. `Cyclically reduce the result.`
3. `Check the length of` $w$.

   `If the corresponding bin` $B_i$ `has space put` $w$ `into it.`

   `Otherwise, go to step 1.`

**Algorithm 2. Generate non-primitive part.**

`Choose a word of length 10 at random and assign it to` $w$.

`If` $w$ `is primitive (Whitehead algorithm) repeat the previous step.`

`Repeat until all bins are full:`

1. `Apply a randomly chosen Whitehead automorphism to` $w$.

2. `Cyclically reduce the result.`

3. `Check the length of` $w$.

   `If the corresponding bin` $B_i$ `has space put` $w$ `into it.`

   `Otherwise, go to step 1.`

We would like to note that choosing random elements of length 10 at the beginning does not affect the representativeness of $D$. By applying randomly chosen Whitehead automorphisms we are able to construct elements of various *Whitehead minimal lengths* ( defined in [55]). We chose 10 for computational reasons only.

## 6.3 An iterative clustering procedure

In this section we suggest a general method that can be used to uncover a hidden structure in a given set of data.

Let $D$ be a given set with a unary predicate $P(x)$. Put

$$D_0 = \{x \in D \mid P(x) = 0\}, \quad D_1 = \{x \in D \mid P(x) = 1\}.$$

Consider the following task: given sets $D_0$ and $D_1$ (or some sample subsets of them) find some "interesting" properties of elements from $D_0$ and $D_1$. To put it more precisely, let $f = (f_1, \ldots, f_n)$ be a feature vector-function on $D$. One may view features $f_i$ (and their combinations) as particular properties of elements from $D$. In this setting the task above amounts to finding some combinations of the features $f_i$ which describe reasonably big clusters of elements in $D_0$ or in $D_1$.

To solve the task we use the following *iterative clustering procedure* **ICP**. Let $P_f$ be a particular **PR** system with the feature vector-function $f$. For certainty, assume

that $P_f$ is the **PR** system based on linear regression model with the equal interval quantizing decision rule.

**Step 1. Searching for a cluster.**

Take the set $D$ as a training data set for $P_f$ and find a corresponding vector of regression coefficients $\beta$. Denote by $d(x) = \beta f(x)$ the corresponding discriminant function and by $R_{f,\beta}$ the corresponding classifier. If there exists a quantizing interval $I$ (or a sequence of consecutive intervals) such that most of elements $x$ with $d(x) \in I$ are from the subset $D_i$ $(i = 1, 2)$ then we say that $R_{f,\beta}$ *found a cluster*.

In the case that no clusters were found, one may look for new features replacing the feature vector-function $f$ by a new one.

**Step 2. Finding significant features.**

Let $C_1$ be a cluster found on Step 1. We used the feature vector-function $f$ to find $C_1$. It may happen that some of the features in $f$ are essential (*significant*) for finding $C_1$ and some of them are not. There are several standard methods to find significant features for the cluster $C_1$ (see [58] for standard tests on significance).

Intuitively, the set of significant features for $C_1$ gives the combined set of properties that separates $C_1$ in $D$. This, in a sense, gives a particular solution to the task above.

**Step 3. Iteration.**

Remove the cluster $C_1$ found on Step 1 from the set $D$ and denote $D' = D - C_1$. Run **ICP** on the set $D'$.

After $n$ iterations of **ICP** one has intervals $I_1, \ldots, I_n$, clusters $C_1, \ldots, C_n$, sets of the most significant features $g_1, \ldots, g_n$, vectors of regression coefficients $\beta_1, \ldots, \beta_n$, and the corresponding discriminant functions $d_1, \ldots, d_n$. One can construct now a new "combined" classifier $R$ as follows. Given $x \in D$ compute $d_1(x)$. If $d_1(x) \in I_1$ then classify $x$ with respect to $R_{g_1,\beta_1}$. If $x \notin I_1$ then compute $d_2(x)$ and repeat the

procedure.

**Step 4. Analysis of clusters and features.**

Given a cluster $C_i$ as above and the corresponding vector of significant features $g_i$ one may try to analyze $g_i$ and extract the "combined meaning" of the features from $g_i$. This may provide an algebraic hypothesis on the structure of elements from $C_i$, thus resulting in a sufficient condition for an element $x \in D$ to belong to the set $D_0$ (if $C_i \subset D_0$) or to the set $D_1$ (if $C_i \subset D_1$).

Sometimes, such an analysis may provide a good new feature vector, hence a better classifier.

**Step 5 Preprocessing: partitioning the data set $D$.**

It happens sometimes that **ICP** fails on Step 1 because the linear regression does not work on the whole set $D$, but the set $D$ is the union of subsets $B_1, \ldots, B_n$, such that the system $P_f$ works quite efficiently on every subset $B_i$ producing a regression coefficient vector $\gamma_i$ with $\gamma_i \neq \gamma_j$. In other words, there is no unique vector of regression coefficients that works efficiently for all $B_i$'s. In this event one may try to partition first the set $D$ using some known clustering algorithms and then apply **ICP** to each of the obtained clusters.

## 6.4 Classification by subwords

In this section we illustrate the performance of the iterative clustering procedure on the **PR** system $P_f$ defined before.

The accuracy of the classifier $R_f$ based on the system $P_f$ with the linear regression model is only 55%. The classifier $R_{f'}$ based on the quadratic regression model has accuracy around 72%. The cubic regression model gives about the same accuracy measure as the quadratic one. Now we may apply the next step of the

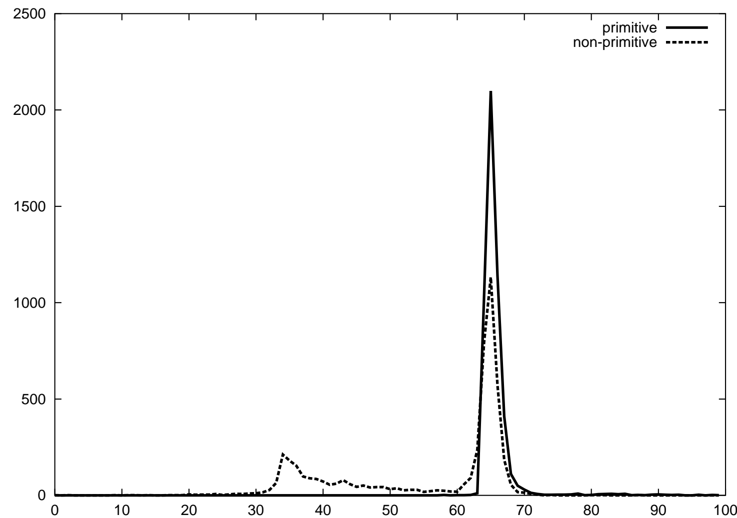


Figure 6.1: Classification by subwords

iterative clustering procedure.

Figure 6.1 shows the corresponding quantizing intervals and the number of primitive and non-primitive elements in each interval. One can notice a subclass of non-primitive elements in the left part of the figure (before the interval 64), which is separated perfectly from the rest. This means that the procedure found a first cluster and Step 1 of the procedure is finished.

On Step 2 the procedure did not reduce the number of features, i.e., all the features turned out to be equally significant. On Step 3 the procedure removed these perfectly classified elements from the data set, which resulted in a new data set. At this point the procedure repeated Step 1 and found another cluster.

The resulting two-step classifier has an improved accuracy of 85%. It happened that we could repeat this process one more time and the resulting three-step classifier has accuracy of 88%. Afterward no more clusters of significant size could be found and the improvement of the accuracy became negligible.

In the Table 6.1 we collected results of experiments with the iterative clustering procedure on other feature vectors.

| Features | After Step 1 | After Step 3 |
|---|---|---|
| C(w,v) | 72% | 88% |
| C(w,y*z) | 72% | 80% |
| C(w,y**z) | 75% | 85% |
| C(w,y***z) | 71% | 81% |
| C(w,y****z) | 75% | 85% |
| C(w,y**z) and C(w,y****z) | 75% | 85% |

Table 6.1: Accuracy of classifiers for various subword features.

## 6.5 Classification by exponents

In this section we demonstrate how **ICP** may lead one to purely algebraic conjectures. First we introduce new features of the type we have not considered before. Any word $w$ in $F_2$ can be written in one of the following two forms:

$$w = a^{n_1} b^{m_1} a^{n_2} b^{m_2} \dots a^{n_k} b^{m_k} \dots,$$

or

$$w = b^{m_1} a^{n_1} b^{m_2} a^{n_2} \dots b^{m_k} a^{n_k} \dots,$$

where $i$, $k$, $n_i$, and $m_i$ are non-zero integers. Since a cyclic permutation does not change the primitiveness of a word, we can always consider that all words in $D$ are written in the first form. Let us denote a vector of exponents in $w$ by $e(w)$:

$$e(w) = < n_1, m_1, \dots, n_k, m_k, \dots >$$

and denote by $|e(w)|$ the length of vector $e(w)$.

$D$ is a finite set and so $n_i$'s can only have a finite number of different values in

$D$. Let us denote these values by $u_1, ..., u_p$:

$$n_i \in \{u_1, ..., u_p\}.$$

$m_i$'s can also have only a finite number of different values in $D$. Let us denote these by $v_1, ..., v_q$:

$$m_i \in \{v_1, ..., v_q\}.$$

Then we define our new features as follows. Feature $g_i$ for a word $w$ is the number of times $a$ occurs with exponent $u_i$ in $w$, where $1 \leq i \leq p$. Feature $h_j$ for a word $w$ is the number of times $b$ occurs with exponent $v_j$ in $w$, where $1 \leq j \leq q$. These new features form a new feature vector $f''$:

$$f''(w) = \frac{1}{|e(w)|} < g_1, ..., g_p, h_1, ..., h_q >$$

.

Applying Step 1 through 4 of **ICP** using the feature vector $f''$ we can find new clusters and build a new classifier $R_{f''}$. However, $R_{f''}$ does not have a better accuracy than the classifiers from Section 6.4, and so we move to Step 5 of **ICP** - partitioning the data set. As mentioned before, in order to do that we can try to run known clustering algorithms. Alternatively, we can partition $D$ using some simple criteria. For example, let us forget for a moment whether or not elements in $D$ are primitive, let $n$ be the number of times letter $a$ occurs in word $w$ divided by the length of $w$, let $\mu$ be the mean value of $n$ computed on the data set $D$ and let $Q(w)$ be the following:

$$Q(w) = \begin{cases} 1, & n < \mu; \\ 0, & \text{otherwise.} \end{cases}$$

Using $Q(w)$ we can partition $D$ into two subsets. Now we can apply Step 1 of

the **ICP** to both subsets separately. Experiments show that we can find big clusters in both partitions. In other words, we achieve much better results when compared to applying Step 1 to the whole data set $D$. Testing features on significance shows that in the first partition the most significant features depend only on $b$, while in the second partition they depend only on $a$. Moreover, these features in both partitions are of the same type and the only difference between them is the letter that they depend on. At this point we can continue to work with both partitions separately. However, since automorphism

$$\phi = \begin{cases} a \longmapsto b \\ b \longmapsto a \end{cases}$$

does not change whether or not an element is primitive, we can apply it to all words in $D$, join both partitions together and continue working with a single data set.

Applying $\phi$ proved to be very useful. Clearly we can further modify $D$ by applying other automorphisms in the hope of achieving better results. We then further *normalize* the data set by applying

$$\phi' = \begin{cases} a \longmapsto a^{-1} \\ b \longmapsto b^{-1} \end{cases}$$

and cyclic permutations, so that our words will always start with $a$ and end with $b$. We can do this since $\phi'$ or cyclic permutations do not change whether or not a word is primitive. We will call the normalized data set $D'$.

Applying **IPC** to $D'$ using the feature vector $f''$ showed that there are two most significant features in our vector. Let us call them $f_1$ and $f_2$:

1. $f_1$ - the number of times $a$ occurs with exponent 1

2. $f_2$ - the number of times $a$ occurs with exponent -1

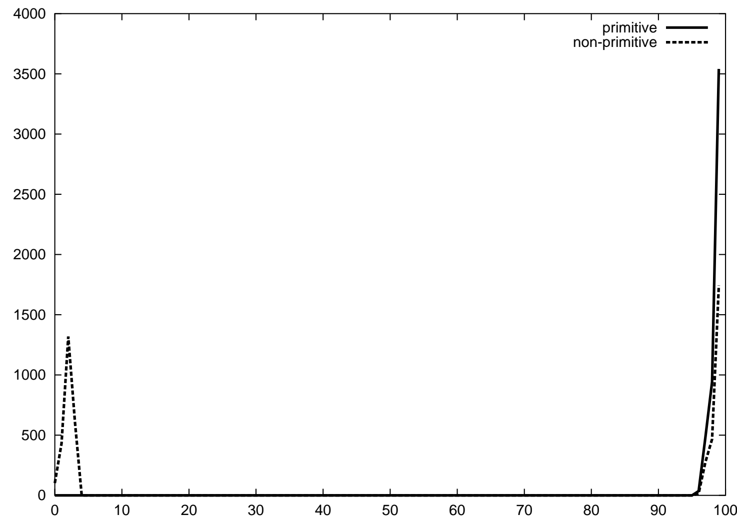


Figure 6.2: Classification by using $f_1$ and $f_2$

Classifier $R_{f''}$ separated a large subset of non-primitive elements from the rest of $D'$ (see the left part of Figure 6.2 ). Let us call this separated part by class $C_1$ and the rest of $D'$ by class $C_2$.

By plotting the values of $f_1$ against $f_2$ in both classes (see Figures 6.3, 6.4 ) we can clearly see that the picture for $C_1$ is very different from the picture for $C_2$. The points of $C_2$ lie exactly on one of the axis or extremely close to it. Being on one of the axis implies that either $f_1 = 0$ or $f_2 = 0$. In other words, the exponents have the same sign. On the other hand, the picture for $C_1$ shows that exponents in $C_1$ do not have the same sign.

Clearly, we found a new piece of information, which we now incorporate into a new feature:

$$f_3 = \begin{cases} 1, & if \text{ a } has\ the\ same\ sign; \\ 0, & otherwise. \end{cases}$$

Experiments show that a single feature $f_3$ performs as well as $f_1$ and $f_2$ combined. This demonstrates the process of constructing new features out of the discovered ones as described in Step 4 of **ICP**.

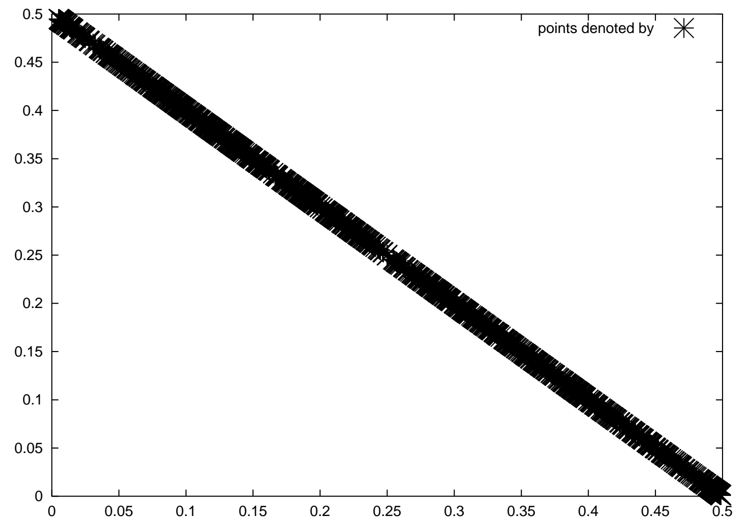


Figure 6.3: Class $C_1$

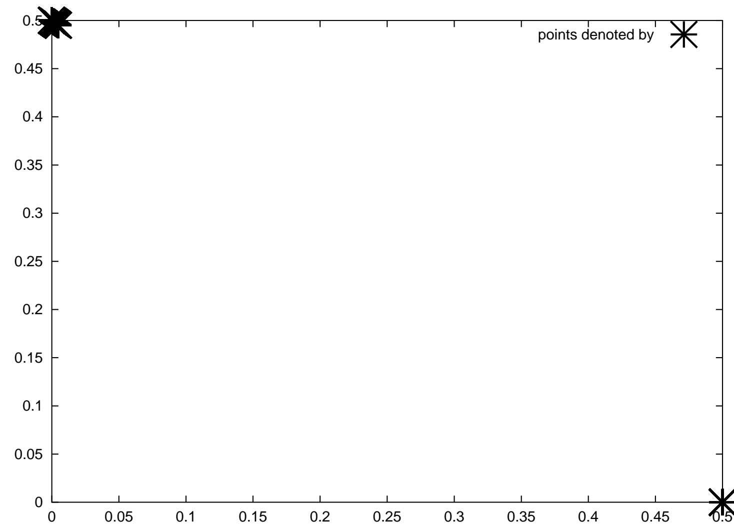


Figure 6.4: Class $C_2$

We will also have the same results if we replace $a$ by $b$ in the definition of features $f_1, f_2, f_3$. Interestingly enough, we actually found a theorem proved by Nielsen [57]:

**Theorem (Nielsen criterion).** Let's consider word $w$ in $F_2$, which starts with $a$'s and ends with $b$'s but the exponents for a given generator do not all have the same sign. Then $w$ is not primitive.

Another way to construct new features is to use other representation for words, for instance the Whitehead graph. The Whitehead graph $Wh(w)$ for word $w \in F(x_1, \dots, x_n)$ is constructed as follows. The vertices of $Wh(w)$ correspond to generators $x_i$ and their inverses. For every subword $x_i x_j$ in $w$, $Wh(w)$ contains an edge connecting vertex $x_i$ to $x_j^{-1}$, for every subword $x_i x_j^{-1}$ in $w$, $Wh(w)$ contains an edge connecting vertex $x_i$ to $x_j$, and so on.

We can add all properties of the Whitehead graph to our feature vector. Testing our best features against the new ones will give us another interesting fact. $f_3$ separates a subset of non-primitive elements, which contains more than 99% of all elements, whose Whitehead graph does not have a cut point. This leads us to another well known theorem:

**Theorem (Whitehead).** Let $w$ be a word in $F_2$. If the Whitehead graph of $w$ does not have a cut point, $w$ cannot be primitive.

# Chapter 7

# Attacking Dehornoy forms

## 7.1 Introduction

In this chapter we experiment with Dehornoy forms for words in a braid group. Our primary goal is to evaluate how well these forms perform when used for information hiding.

Section 6.1 gives the necessary background material on Dehornoy forms and algorithms involved. It also discusses how we generate test data for our experiments, so that the set of all possible inputs is adequately represented.

Section 7.2 describes our experiments on data leakage. It discussed various metrics used and concludes on how much information is leaked when Dehornoy algorithm is used for data hiding.

Section 7.3 contains partial attacks on the first of the two encryption steps of the AAG crypto-system [62]. This is the step where Dehornoy algorithm is used. Different attacks are described, each of which pinpointing a possible vulnerability and suggesting an adjustment of parameters to make the system stronger.

### 7.1.1 Definitions

In this section we briefly review the necessary material on Dehornoy forms and specify the forms and algorithms being used in our experiments. For more details on Dehornoy forms see [59].

`Definition.` *Artin's braid group* $B_n$ is the group generated by $\sigma_1, ..., \sigma_{n-1}$ submitted to the relations

1. $\sigma_i \sigma_j = \sigma_j \sigma_i$ where $|i - j| \geq 2$
2. $\sigma_i \sigma_{i+1} \sigma_i = \sigma_{i+1} \sigma_i \sigma_{i+1}$ for all $i = 1, ..., n - 2$

`Definition.` The braid word $w$ is *reduced* (or in *Dehornoy form*) either if $w$ is the nullstring, or if the *main generator* of $w$, defined as the generator with lower index occurring in $w$, occurs only positively or only negatively.

As showed by Patrick Dehornoy in 1994 [59, 60], computing the Dehornoy form for braid word $w$ automatically solves the word problem for $w$:

`Proposition.` A nonempty reduced braid word cannot be equivalent to the nullstring.

The process of computing a Dehornoy form for $w$ consists of finding special subwords in $w$ called *handles* and deleting them in a systematic way.

`Definition.` A $\sigma_j$*-handle* is a braid word of the form $\sigma_j^e v \sigma_j^{-e}$, where $e$ is $+1$ or $-1$ and the word $v$ contains only generators $\sigma_k^{\pm 1}$ with $k < j - 1$ or $k > j$. A *main* handle of $w$ is a subword of $w$ that is a $\sigma_i$-handle, where $\sigma_i$ is the main generator of $w$.

To delete handle $\sigma_j^e v \sigma_j^{-e}$ one can use a so-called *local reduction*, which is to apply *in the handle* the alphabetical homomorphism $\phi_{j,e}$ defined by

$$\phi_{j,e} : \begin{cases} \sigma_j^{\pm 1} \longmapsto \varepsilon \\ \sigma_{j+1}^{\pm 1} \longmapsto \sigma_{j+1}^{-\varepsilon} \sigma_j^{\pm 1} \sigma_{j+1}^{\varepsilon} \\ \sigma_k^{\pm 1} \longmapsto \sigma_k^{\pm 1} fork \neq j, j+1 \end{cases}$$

It is clear that applying local reduction will remove one handle. One can also see that it might create one or more other handles instead. The authors in [59] define so-called *permissible* handles and argue that if only these types of handles are removed than the process of finding reduced forms will terminate in finitely many steps.

There are many ways and strategies of computing reduced forms. Obviously, if one is only interested in solving the word problem for a given word, then it is enough only to remove the main handles, which lessens the time of the computation. In this chapter we will use so-called *fully reduced forms*, i.e. the forms obtained after all handles have been eliminated. They are more of interest for data hiding, since

- the change to the original word is much greater;
- there are more choices of a paths the computation might follow;
- on average, the resulting word is shorter than the original word;
- these forms are used for data hiding in the AAG crypto-scheme [62].

To compute fully reduced forms we will use a simple, though not necessarily most efficient, *"FullHRed"* algorithm from [59].

`Algorithm.` Repeat the following steps until word $w$ is fully reduced:

1. Remove the leftmost handle of $w$ by applying local reduction.
2. Freely reduce the result.

### 7.1.2 Generating test data

The main question here is how to generate test data, which adequately represents the whole set of possible inputs. Since braid groups are infinite, their elements are not uniformly distributed. The same problem remains if we replace infinite group $B_n$ by a ball with a radius $k$, where k is a big enough number, but not too big to make practical computations impossible. For details refer to [64] and [65].

In our case the problem is somewhat simpler, since we generate random words, not elements. In the crypto-schemes [61, 62, 63] the authors randomly generate words to build public and private keys and therefore our experimenting with random words is appropriate.

To further simplify the problem, since we are dealing with real life crypto-schemes, we do know the parameters being used and so, we do not need to deal with randomly chosen infinite words, but with randomly generated words of particular lengths, or particular range of lengths.

In order to generate a random word of length $l$ in a group with $n$ generators, we use an uniform pseudo-random generator to choose the first letter with a probability $1/2n$ and all the remaining ones with probability $1/(2n-1)$.

To ensure representativeness of the test data set we use the approach suggested in [51]. Typically we construct data set $S_{test}$ as a union:

$$S_{test} = \bigcup_{l=1}^{L} \bigcup_{i=1}^{K} w_{i,l} \quad (w_{i,l} \in B_n),$$

where $L$ is the maximum length considered and $K$ is the number of words generated for each length.

As mentioned above we know what $L$ is for our experiments. Also, it is meaningless to start from length 1. A handle may only occur in a word of a length at least 3. Since we are interested in data hiding, we would like to apply to a word

more then just a few local reductions, which is impossible for very short words. So, in our experiments we consider words of length at least 20. The justification for this is that in real crypto-systems, the Dehornoy algorithm is applied to much longer words - a few thousand letters long or more.

As of the choice for $K$, the bigger $K$ we take, the more precision we will have for the result. On the other hand, we would like $K$ to be as small as possible, so that the experiments consume less time and therefore allow further bigger scale experiments to finish in a reasonable time. Typically we are interested in some numerical characteristic of the set $S_{test}$, which represents a feature either of this set or the algorithm. Let us call it $\chi(S_{test})$. For instance, in the next section we are interested in numerical characteristics, such as the average length of fully reduced words for $S_{test}$ or the Hamming distance between words from $S_{test}$ and their reduced forms. We then fix a small real number $\varepsilon > 0$ and pick $K$ as small as possible, such that any increase of $K$ will imply a change in $\chi(S_{test})$ within $\varepsilon$.

All our experiments show that $K = 200$ gives us the exact value of $\chi(S_{test})$ that we consider up to 3 or 4 digits after the decimal point. The further increase of $K$ results in stabilizing the further digits, but since we consider this precision sufficient, we choose $K = 200$ for all experiments. So, finally, we define $S_{test}$ to be

$$S_{test} = \bigcup_{l=L1}^{L2} \bigcup_{i=1}^{200} w_{i,l} \quad (w_{i,l} \in B_n),$$

where $[L1, L2]$ is the range of length that we are interested in.

## 7.2 Measuring data leakage

### 7.2.1 Cyclical Hamming Distance

To analyze how well the Dehornoy form performs when used for data hiding, we will work with the set of pairs

$$S_n = \{(w, \overline{w})\},$$

where $w \in S_{test}$, $\overline{w}$ is the fully reduced form of $w$, and $n$ is the number of generators of braid group $B_{n+1}$.Here we would like to find out how well the information $w$ is hidden in its Dehornoy form $\overline{w}$, or, in other words, how different is $\overline{w}$ from $w$. To measure this difference we would like to use a numerical characteristic or a metric. One possible choice is the cyclical Hamming distance between two words - $\mu_h(w, \overline{w})$.

To define $\mu_h(w, \overline{w})$, first we modify the words to be of the same length. This can be done by appending some previously unused symbol ('$', for example) to the shorter word as many times as necessary, so that $w$ and $\overline{w}$ now have the same length $l$. Let us define $w[i]$ to be the letter of a word $w$ at position $i$, where $1 \leq i \leq l$ and define a new function $c(i, w, \overline{w})$:

$$c(i, w, \overline{w}) : \begin{cases} 1, & if w[i] = \overline{w}[i] \\ 0, & if w[i] \neq \overline{w}[i] \end{cases}$$

The *Hamming distance* $d(w, \overline{w})$ is then

$$d(w, \overline{w}) = \sum_{i=1}^{l} c(i, w, \overline{w})$$

The Hamming distance may be able to shed some light on how well $w$ is "hidden" when taken in its Dehornoy form $\overline{w}$. Does $\overline{w}$ still contain some significant information on $w$? In case of words the significant information may mean that $w$ and $\overline{w}$ share

a long common subword, which will increase the hamming distance. $d(w, \overline{w})$ will also increase if there are some common letters at the same positions in both words. Hence, the bigger the Hamming distance between $w$ and $\overline{w}$, the more information leakage we have in its Dehornoy form.

We also need to notice that simply computing $d(w, \overline{w})$ may not be enough to establish if $w$ and $\overline{w}$ have a common subword. The local reduction process replaces subwords by subwords, thereby changing the length of the word and shifting parts of it. In other words, the long common subword may still be there, but it might be shifted and not accounted for by $d(w, \overline{w})$, which works on one position at a time. To account for the shifts we use the cyclical Hamming distance $h(w, \overline{w})$ instead

$$h(w, \overline{w}) = \max_{S} d(w, \overline{w}),$$

where S is the set of all cyclical permutation of $w$.

Dividing $h(w, \overline{w})$ by the length of $w$ will give us the characteristic of how many percents of information from $w$ is being leaked. We will name this characteristic $\mu_h(w, \overline{w})$ and use it to evaluate our experiments:

$$\mu_h(w, \overline{w}) = \frac{h(w, \overline{w})}{|w|}$$

### 7.2.2 Experimental data

In addition to data leakage, another interesting question about Dehornoy forms is their length on average when compared to the length of the original words. This might also influence the Hamming distance $\mu_h(w, \overline{w})$. For example, if the full Dehornoy form is to be much shorter than the original word, then obviously $\mu_h(w, \overline{w})$ will be small and somewhat less useful as an information leakage measure. In [59] the author state that the fully reduced form is usually shorter than the original

word. Our experiments confirm that fact and show that though the Dehornoy forms are shorter on average, they are only somewhat shorter.

The table below shows the ratio $\frac{|\overline{w}|}{|w|}$ for different braid groups with rank $n$ and different length intervals for $w$. One can see that in our experiments the lengths of Dehornoy forms on average lay between 71 and 94 percent of the lengths of the original words.

| Rank n | 3 | 5 | 10 | 15 | 20 | 50 | 80 |
|---|---|---|---|---|---|---|---|
| Length 20 - 50 | 0.776 | 0.791 | 0.746 | 0.754 | 0.764 | 0.830 | 0.869 |
| Length 20 - 100 | 0.764 | 0.813 | 0.743 | 0.736 | 0.737 | 0.786 | 0.823 |
| Length 100 - 200 | 0.750 | 0.892 | 0.775 | 0.731 | 0.717 | 0.718 | 0.735 |
| Length 100 - 500 | 0.746 | 0.939 | 0.846 | 0.763 | 0.733 | 0.707 | 0.709 |

Table 7.1: $|\overline{w}|/|w|$

Table 7.2 shows the information leakage $\mu_h(w, \overline{w})$. At first it seems that the leakage is simply too big for the Dehornoy forms to be of any use in data hiding. For example, for $n = 80$ and length interval 20-50 the leakage approaches 44%, which means that almost half of $w$ is being leaked!

| Rank n | 3 | 5 | 10 | 15 | 20 | 50 | 80 |
|---|---|---|---|---|---|---|---|
| Length 20 - 50 | 0.314 | 0.270 | 0.246 | 0.256 | 0.267 | 0.358 | 0.438 |
| Length 20 - 100 | 0.271 | 0.225 | 0.185 | 0.182 | 0.185 | 0.235 | 0.282 |
| Length 100 - 200 | 0.207 | 0.165 | 0.107 | 0.090 | 0.082 | 0.079 | 0.085 |
| Length 100 - 500 | 0.186 | 0.150 | 0.090 | 0.070 | 0.061 | 0.050 | 0.050 |

Table 7.2: $\mu_h(w, \overline{w})$

Such big values for $\mu_h(w, \overline{w})$ raise a question: is there problem with the use of Dehornoy forms for hiding or this values are big for some other reason? Let us consider the value of $\mu_h(w, \overline{w})$ in the top left corner of the table, where the value is about 31%. Here we have 3 generators and the length of $w$ is between 20 and 50. Imagine that we don't compute the Dehornoy form for $w$, but generate $\overline{w}$ randomly

of the same length, independent of the choice of $w$. What should the value of the leakage function be in this case? Intuitionally, it seems that this value should be high, since we only have 3 generators (6 possibilities), both words are short and since we consider all cyclical permutations and take the maximum value possible, it seems that the chances are pretty high, that we find a combination in which some letters will align themselves, increasing the Hamming distance and therefore the leakage function.

Table 7.3 shows the information leakage (let us call it $r_h(w, \overline{w})$) when $w$ and $\overline{w}$ are independently randomly chosen words of the same length. Looking at the top left corner again we see the value of 32%, which is very close to the 31% value we had for $\mu_h(w, \overline{w})$! But in this case both words are chosen randomly and information leakage has no meaning. We can argue then that $\mu_h(w, \overline{w})$ does not really show the information leakage, that the "absolutely no-leakage" values are given by $r_h(w, \overline{w})$ and any deviation from it should constitute a leakage. In other words, the real leakage is given by $\mu_h(w, \overline{w}) - r_h(w, \overline{w})$.

To make the value more precise let us remind that the Dehornoy forms are somewhat shorter than the original words and so computing $r_h$ for the words of the same length is not what we really want. A better way of computing $r_h(w, \overline{w})$) would be generating randomly $\overline{w}$ of the same length as the Dehornoy form for $w$. To conclude, here is the way we use to compute leakage:

1. For each rank $n$ and length interval compute $\mu_h(w, \overline{w})$ and the length ratio $|\overline{w}|/|w|$.

2. For each rank $n$ and length interval randomly generate $\overline{w}$ of length with the ratio from step 1. Using this new $\overline{w}$ compute $r_h(w, \overline{w})$. Let us call it $r'_h(w, \overline{w})$.

3. The information leakage is $\mu_h(w, \overline{w}) - r'_h(w, \overline{w})$

Table 7.4 shows the value for the new leakage function.

| Rank n | 3 | 5 | 10 | 15 | 20 | 50 | 80 |
|---|---|---|---|---|---|---|---|
| Length 20 - 50 | 0.317 | 0.221 | 0.141 | 0.110 | 0.094 | 0.057 | 0.045 |
| Length 20 - 100 | 0.295 | 0.204 | 0.128 | 0.099 | 0.084 | 0.050 | 0.039 |
| Length 100 - 200 | 0.256 | 0.171 | 0.103 | 0.078 | 0.064 | 0.037 | 0.028 |
| Length 100 - 500 | 0.237 | 0.156 | 0.092 | 0.068 | 0.056 | 0.031 | 0.023 |

Table 7.3: $r_h(w, \overline{w})$

| Rank n | 3 | 5 | 10 | 15 | 20 | 50 | 80 |
|---|---|---|---|---|---|---|---|
| Length 20 - 50 | 0.055 | 0.086 | 0.130 | 0.163 | 0.188 | 0.307 | 0.397 |
| Length 20 - 100 | 0.033 | 0.051 | 0.081 | 0.100 | 0.116 | 0.192 | 0.247 |
| Length 100 - 200 | 0.005 | 0.009 | 0.020 | 0.027 | 0.030 | 0.048 | 0.061 |
| Length 100 - 500 | 0.001 | 0.001 | 0.009 | 0.014 | 0.016 | 0.025 | 0.031 |

Table 7.4: $\mu_h(w, \overline{w}) - r'_h(w, \overline{w})$

### 7.2.3 Levenstein distance

The cyclical Hamming distance will show if two words share a big common piece. However, if there are many short common pieces and they are not aligned, we will need a more "sensitive" metric. A better choice would be to use a so-called Levenstein or edit distance between two words. This distance is the minimal number of operations necessary to rewrite one word into another. Here we will use three operations: inserting a letter, deleting a letter and replacing a letter by another letter. Table 7.5 shows the data leakage computed by the algorithm in the previous section when Levenstein distance is employed. We can see that the values are bigger than when the Hamming distance is used, showing that the Levenstein distance is a better measurement of data leakage.

### 7.2.4 Conclusions

Looking at tables 7.4 and 7.5 one may notice the following:

| Rank n | 3 | 5 | 10 | 15 | 20 | 50 | 80 |
|---|---|---|---|---|---|---|---|
| Length 20 - 50 | 0.140 | 0.232 | 0.380 | 0.468 | 0.534 | 0.735 | 0.815 |
| Length 20 - 100 | 0.101 | 0.172 | 0.299 | 0.376 | 0.432 | 0.631 | 0.724 |
| Length 100 - 200 | 0.046 | 0.080 | 0.172 | 0.230 | 0.273 | 0.428 | 0.522 |
| Length 100 - 500 | 0.031 | 0.051 | 0.121 | 0.175 | 0.213 | 0.338 | 0.411 |

Table 7.5: $\mu_h(w, \overline{w}) - r'_h(w, \overline{w})$

1. For a fixed length interval as rank of the group increases, so is the leakage function. Intuitionally, this is clear - as we have more letters, a randomly generated word will have fewer handles, therefore the local reduction will apply a fewer number of times, hence there is more leakage.

2. For a fixed rank and increasing length the leakage function is decreasing. Again, for the same reason, as the length gets bigger, a randomly generated word will have more handles and the local reduction will be applied a greater number of times.

3. The leakage is sufficiently small if a proper length interval is chosen for each group. For example for $n = 3$ the words of length 20-100 will have a 3% leakage, while the words of length 100-500 only 0.1% leakage. For $n = 80$ in order to have a 3% leakage, the words must have length 100-500, and if we need a better data hiding the length must be increased.

## 7.3 Partial attacks on AAG crypto-system

### 7.3.1 Decreasing data leakage

Experiments show that the process of rewriting a word into its Dehornoy form is not uniform, in the sense that some parts of the word get rewritten more times than others. Generally, most rewriting rules involve letters in the middle of the word and the beginning and the end of the word do not change as much.

In the AAG scheme [62] we choose 20 generators for each subgroup, each of which has length 5. So, totally a subgroup is described by 100 letters. Since the group has 80 letters, on average every letter occurs approximately 1.2 times in the description of the subgroup. Therefore, even a single letter is likely to identify a subgroup generator. In the case that we have not a letter but a longer piece of a generator left, the identification becomes even easier. Since the beginning of a password is not being rewritten as well as its middle part, some pieces of generators might be left and we might be able to identify some initial segment of the password.

Here is one example of a typical situation:

$$\begin{aligned}
Subgroup\ S_A = < \quad a_1 &= \sigma_{29}\sigma_{45}\sigma_{66}\sigma_{52}^{-1}\sigma_{18}, \\
a_2 &= \sigma_{13}\sigma_{17}^{-1}\sigma_{43}^{-1}\sigma_{26}^{-1}\sigma_{16}^{-1}, \\
a_3 &= \sigma_{64}^{-1}\sigma_{10}^{-1}\sigma_{5}\sigma_{31}^{-1}\sigma_{68}, \\
a_4 &= \sigma_{41}^{-1}\sigma_{39}^{-1}\sigma_{27}^{-1}\sigma_{48}\sigma_{30}^{-1}, \\
a_5 &= \sigma_{35}^{-1}\sigma_{66}^{-1}\sigma_{54}^{-1}\sigma_{25}^{-1}\sigma_{74}^{-1}, \\
a_6 &= \sigma_{15}\sigma_{14}\sigma_{32}^{-1}\sigma_{21}\sigma_{71}^{-1}, \\
a_7 &= \sigma_{63}\sigma_{24}\sigma_{73}\sigma_{50}^{-1}\sigma_{2}, \\
a_8 &= \sigma_{24}^{-1}\sigma_{6}^{-1}\sigma_{62}\sigma_{42}^{-1}\sigma_{76}^{-1}, \\
a_9 &= \sigma_{78}^{-1}\sigma_{58}^{-1}\sigma_{52}^{-1}\sigma_{46}^{-1}\sigma_{9}, \\
a_{10} &= \sigma_{79}^{-1}\sigma_{55}^{-1}\sigma_{61}^{-1}\sigma_{49}\sigma_{3}^{-1}, \\
a_{11} &= \sigma_{36}^{-1}\sigma_{19}^{-1}\sigma_{7}\sigma_{37}\sigma_{28}^{-1}, \\
a_{12} &= \sigma_{77}^{-1}\sigma_{20}\sigma_{57}^{-1}\sigma_{11}\sigma_{16}, \\
a_{13} &= \sigma_{43}^{-1}\sigma_{23}\sigma_{75}\sigma_{47}\sigma_{70}, \\
a_{14} &= \sigma_{44}\sigma_{34}\sigma_{80}\sigma_{17}\sigma_{72}^{-1}, \\
a_{15} &= \sigma_{65}^{-1}\sigma_{61}\sigma_{38}\sigma_{37}\sigma_{8}, \\
a_{16} &= \sigma_{42}^{-1}\sigma_{36}\sigma_{5}^{-1}\sigma_{22}\sigma_{2}^{-1}, \\
a_{17} &= \sigma_{53}\sigma_{65}^{-1}\sigma_{60}^{-1}\sigma_{4}^{-1}\sigma_{75}^{-1}, \\
a_{18} &= \sigma_{53}^{-1}\sigma_{67}^{-1}\sigma_{12}^{-1}\sigma_{1}\sigma_{72}^{-1}, \\
a_{19} &= \sigma_{51}\sigma_{25}\sigma_{40}^{-1}\sigma_{56}\sigma_{33}^{-1}, \\
a_{20} &= \sigma_{79}^{-1}\sigma_{4}^{-1}\sigma_{69}^{-1}\sigma_{21}\sigma_{59}^{-1} \quad >
\end{aligned}$$

Let $p$ be the password, chosen as random word of length 100 in generators of $S_A$ and let $d$ be its Dehornoy form. Next, we will show the initial segments of $p$ and $d$, underlying letters that match. We will also use $'|'$ to separate generators of $S_A$.

$$\begin{aligned} p = {} & \underline{\sigma_{71}}\sigma_{21}^{-1}\sigma_{32}\sigma_{14}^{-1}\sigma_{15}^{-1} \mid \underline{\sigma_{44}}\sigma_{34}\underline{\sigma_{80}}\sigma_{17}\sigma_{72}^{-1} \mid \underline{\sigma_{74}\sigma_{25}}\sigma_{54}\sigma_{66}\sigma_{35} \mid \underline{\sigma_{79}^{-1}\sigma_4^{-1}\sigma_{69}^{-1}}\sigma_{21}\sigma_{59}^{-1} \\ & \mid \underline{\sigma_{74}\sigma_{25}}\sigma_{54}\sigma_{66}\sigma_{35} \mid \underline{\sigma_{77}^{-1}\sigma_{20}}\sigma_{57}^{-1}\underline{\sigma_{11}}\sigma_{16} \mid \sigma_{64}^{-1}\sigma_{10}^{-1}\underline{\sigma_5}\sigma_{31}^{-1}\sigma_{68} \mid \sigma_{53}\sigma_{65}^{-1}\sigma_{60}^{-1}\underline{\sigma_4^{-1}\sigma_{75}^{-1}} \\ & \mid \sigma_{53}\sigma_{65}^{-1}\sigma_{60}^{-1}\underline{\sigma_4^{-1}\sigma_{75}^{-1}} \mid \sigma_2^{-1}\sigma_{50}\sigma_{73}^{-1}\underline{\sigma_{24}^{-1}}\sigma_{63}^{-1} \mid \sigma_{53}\sigma_{65}^{-1}\sigma_{60}^{-1}\underline{\sigma_4^{-1}}\sigma_{75}^{-1} \mid \underline{\sigma_{41}^{-1}\sigma_{39}^{-1}\sigma_{27}^{-1}}\sigma_{48}\sigma_{30}^{-1} \\ & \mid \sigma_2^{-1}\sigma_{50}\sigma_{73}^{-1}\sigma_{24}^{-1}\sigma_{63}^{-1} \mid \underline{\sigma_{24}^{-1}\sigma_6^{-1}}\sigma_{62}\underline{\sigma_{42}^{-1}\sigma_{76}^{-1}} \mid \underline{\sigma_8^{-1}}\sigma_{37}^{-1}\sigma_{38}^{-1}\sigma_{61}^{-1}\sigma_{65} \mid \sigma_{33}\sigma_{56}^{-1}\underline{\sigma_{40}}\sigma_{25}^{-1}\sigma_{51}^{-1} \\ & \mid \sigma_{43}^{-1}\sigma_{23}\sigma_{75}\sigma_{47}\underline{\sigma_{70}} \mid \sigma_{53}\sigma_{65}^{-1}\sigma_{60}^{-1}\underline{\sigma_4^{-1}\sigma_{75}^{-1}} \mid \underline{\sigma_{36}^{-1}}\sigma_{19}^{-1}\sigma_7\sigma_{37}\sigma_{28}^{-1} \mid \ \dots \end{aligned}$$

$$\begin{aligned} d = {} & \underline{\sigma_{71}\sigma_{44}\sigma_{80}\sigma_{74}\sigma_{25}}\sigma_{35}^{-1}\sigma_{34}\underline{\sigma_{79}^{-1}\sigma_4^{-1}\sigma_{69}^{-1}\sigma_{74}\sigma_{25}\sigma_{77}^{-1}\sigma_{20}\sigma_{11}\sigma_5\sigma_4^{-1}\sigma_{75}^{-1}\sigma_4^{-1}\sigma_{75}^{-1}}\sigma_{73}\sigma_{72}^{-1} \\ & \underline{\sigma_{24}^{-1}\sigma_4^{-1}\sigma_{41}^{-1}\sigma_{39}^{-1}\sigma_{27}^{-1}\sigma_{24}^{-1}\sigma_6^{-1}\sigma_{42}^{-1}}\sigma_{76}\sigma_{75}^{-1}\underline{\sigma_{76}^{-1}\sigma_8^{-1}}\sigma_{38}\sigma_{37}^{-1}\underline{\sigma_{40}}\sigma_{25}\sigma_{24}^{-1}\underline{\sigma_{70}\sigma_4^{-1}\sigma_{75}^{-1}}\sigma_7\underline{\sigma_{36}^{-1}} \end{aligned}$$

Looking at the Dehornoy form from left to right we first see $\sigma_{71}$, which occurs only in $a_6^{-1}$. Next comes $\sigma_{44}$, which again occurs only in one generator, $a_{14}$. The following $\sigma_{80}$ also occurs in $a_{14}$, which gives us more confidence that in fact the second five letter word in the password is $a_{14}$. The next subword of length two $\sigma_{74}\sigma_{25}$ clearly identifies $a_5^{-1}$, and so on. The example shows that in this case it is possible to identify first 19 generators of the password - that is 19% of the key!

Of course, the job is much harder when the key is not available. In a real cryptosystem we only have Dehornoy forms to work with. However, we have twenty different forms: $d_1, \dots, d_{20}$, each containing the password twice. The beginning of each $d_i$ contains pieces of the beginning of $p$, but also the end of each $d_i$ contains pieces of the beginning of $p$ inversed. So, we have forty different words to work with. Now one can write a routine, which uses each of the forty words to assign a likely candidate for the beginning of $p$ and then chooses the most likely candidate overall. We might not always get 19% as in the example above, but it seems that the first 5-10% of key can be easily identified. It is also easy to modify the genetic algorithm in section 7.3.2 and use it here.

Giving attacker 5-10% of the key is of course unacceptable, since it makes easier to do traffic analysis and in some cases may give the attacker 5-10% of the plaintext. Naturally, the problem should be not too difficult to fix. For example, one might try to multiply the password by $\sigma_1 \ldots \sigma_{80}$ from left and right before applying the Dehornoy algorithm, so that the end points of $p$ get rewritten better. After fixing this, we will also have a decrease in data leakage, as discussed in section 7.2, since some of the common pieces get eliminated.

### 7.3.2 Genetic attack

Given that the password is of length one hundred in the generators of subgroup $S_A$ and $S_A$ has twenty generators (forty if we count inverses), the brute force attack would take up to $40^{100}$ operations to break the system. This search space is too large to enumerate in practice. When we have a situation like this and don't have any additional information about the structure of the search space, it can be useful to try a heuristic search. It might get us closer to the solution or find out something about the structure, which can be used to develop better deterministic algorithms.

Genetic algorithms is an example of such a heuristic approach and we applied them successfully to a number of problems in group theory. Genetic algorithm works with a population (an array) of candidate solutions. This candidates are evaluated by some fitness criterion (a fitness function). The fitter a candidate is the better is the probability of it to be chosen for reproduction (or to stay in the population). Chosen candidates exchange information, which is called crossover, and are subject to a random change - mutation. This process repeats itself until a solution is found.

For this problem our population is an array of fifty words - candidate passwords, which are chosen randomly at the beginning: $x_1, \ldots, x_{50}$.

To evaluate fitness of a candidate we compute all twenty Dehornoy forms for it: $\overline{d_1}, \ldots, \overline{d_{20}}$ and compare them with the Dehornoy forms for the password: $d_1, \ldots, d_{20}$.

The fitness $f(x_i)$ is then:

$$f(x_i) = \sum_{i=1}^{50} L(d_i, \overline{d_i}),$$

where $L(w_1, w_2)$ is the Levenstein distance between words $w_1$ and $w_2$.

We implement crossover by performing exchange of letters between words, where positions and the number of exchanges are chosen randomly.

Mutations are random changes is words. Through the process of improving the algorithm the following mutations have proved useful:

- changing, deleting or inserting a letter in a word;
- swapping two letters at random positions;
- freely reducing a word;
- inserting a trivial pair at a random position.

A genetic algorithm like this successfully reverses the password for a reduced version of the crypto-system. If the key's length is 25% of the recommended key length, then the algorithm terminates and reverses the password for most inputs. The average time of the computation is about ten minutes on an average PC, which is an exceptional performance, since the brute-force approach will still need $40^{25}$ iterations, which is impossible to do.

The good performance of a genetic algorithm hints about a hidden structure in the search space. An effort to use pattern recognition techniques to uncover this structure is currently under way. Every bit of information uncovered can be possibly used to improve the genetic algorithm by using this information in a fitness function or in genetic operators. For example, using the pattern described in the next section and other information in this section, we improved the algorithm, which now recovers keys of 35% of the recommended key length.

### 7.3.3 Search space reduction

This attack is based on another non-uniformity of the Dehornoy algorithm, exposed by experiments. Let's generate subgroups $S_A =< a_1, \ldots, a_{20} >$ and $S_B =< b_1, \ldots, b_{20} >$, where each $a_i$ and $b_i$ has length five and a password $p$ of length one hundred in generators of $S_A$. In the crypto-system, we compute Dehornoy forms

$$d_i = pb_ip^{-1},$$

where $i = 1, \ldots, 20$. $d_i$'s are public information and we would like to recover $p$ by investigating them.

$p$ can be rewritten in terms of the generators of the group $B_{81} =< \sigma_1, \ldots, \sigma_{80} >$. For $p$ we can build a table of exponents - for each $\sigma_i$ we count the number of times $\sigma_i$ occurs in $p$ with a positive power and the number of times $\sigma_i$ occurs in $p$ with a negative power. Since $p$ is generated randomly, we won't find any interesting structure in the table of exponents. Now, let's compute $d_i$'s and build their tables of exponents. Here's an example of the table for $d_1$, where $b_1 = \sigma_{72}\sigma_{80}^{-1}\sigma_{41}\sigma_6\sigma_{21}$ (we denote all non-zero exponents by '*'):

| Range | # positive exponents | # negative exponents |
|---|---|---|
| $\sigma_1, \ldots, \sigma_1$ | 0 | 0 |
| $\sigma_2, \ldots, \sigma_{11}$ | * | * |
| $\sigma_{12}, \ldots, \sigma_{18}$ | 0 | 0 |
| $\sigma_{19}, \ldots, \sigma_{27}$ | * | * |
| $\sigma_{28}, \ldots, \sigma_{33}$ | 0 | 0 |
| $\sigma_{34}, \ldots, \sigma_{44}$ | * | * |
| $\sigma_{45}, \ldots, \sigma_{67}$ | 0 | 0 |
| $\sigma_{68}, \ldots, \sigma_{80}$ | * | * |

Table 7.6: Exponents in $d_1$

We immediately see that the table has a very particular structure - there are some ranges of $\sigma_i$'s with non-zero exponents separated by ranges of zeroes. In fact,

each letter in $b_1$ has a somewhat “narrow” non-zero range around it. This lead us to the idea that perhaps the occurrence of each letter in $d_i$’s is determined by only a “narrow” range of letters in $p$, and not by all letters in $p$, which would reduce the size of the search space.

Let’s call by $E(w, k_1, k_2)$ a word computed from word $w$ by deleting all letters $\sigma_k$, where k is not in the range $\overline{k_1, k_2}$. Experiments show that in most cases there exist a “small” $\varepsilon$, such that $E(d_i, k_1, k_2)$ can be computed from $E(p, k_1 - \varepsilon, k_2 + \varepsilon)$. Therefore, we can compute pieces of $d_i$’s separately and then combine them together to compute $d_i$’s. More importantly, in reverse, it means that we can compute pieces of the password separately and then combine then to compute the password.

We reduced the original task to solving two tasks of a reduced size. However this reduction is not enough for the genetic algorithm from section 7.3.2 to achieve a full break.

# Chapter 8

# Key hiding

## 8.1 Introduction

In this chapter we introduce a key hiding method, which is based on Dehornoy forms in braid groups. The general idea is to replace 3DES keys by subroutines that compute them. We will demonstrate that keys do not appear in the code, in fact they do not even appear in memory at run time and an attacker could hope to extract only their pieces. Because there is no absolute defense against hackers, we could only hope to make this "piece extraction" more difficult. We discuss the ways we make the debugging difficult. In addition the system is built in such a way that breaking one copy of software would not necessarily mean that all copies are broken, as key hiding method behavior will vary depending on the key.

As for the cryptanalytic attacks, we argue that breaking this system would mean inverting an one-way function. We refer to Section 7.1.1 for the theoretical background necessary for understanding the system. Here we describe how the system works as well as practical problems, such as improving performance. We also demonstrate that the overhead on 3DES encryption/decryption is reasonable enough.

## 8.2 The key hiding scheme

### 8.2.1 Modifications in the 3DES algorithm

The 3DES algorithm performs a three step encryption/decryption by using the DES algorithm. On each step a DES key is being used. Let us call these keys $K_1, K_2$ and $K_3$. In the test version of the Umbanet software for which this key hiding scheme is written, the keys $K_1, K_2$ and $K_3$ are chosen independently and therefore have different values.

Looking inside the DES algorithm, one might notice that key $K_i$ is being used to generate 16 subkeys. These subkeys are then used to encrypt and decrypt data. After each subkey is used DES also performs $XOR$ as well as other math operations on the data. 3DES will have total of 48 subkeys, which we will call $k_1, ..., k_{48}$.

The first step of the key hiding method is to modify the 3DES algorithm, so that all subkey names are explicitly present in the program. In other words, one will need to replace "for" loops by repetition of the code. For example, for a single run of DES, replace

```
for( int i = 0; i < 16; ++i ) {
   ...
   encryptWithSubkey(k_i);
   ...
}
```

by

```
   ...
   encryptWithSubkey(k_1);
   ...
   encryptWithSubkey(k_2);
```

```
...
...
encryptWithSubkey(k16);
...
```

The second step of the key hiding method is to replace all identifiers $k_1, ..., k_{48}$ by a call to a corresponding *hiding function* $k_1(), ..., k_{48}()$. We will define a *hiding function* as a C++ function (or a function in any other programming language), which performs a number of math operations and returns the value of the subkey that corresponds to its name. For example, function $k_1()$ returns the value of subkey $k_1$, function $k_2()$ returns the value of subkey $k_2$ and so on. A hiding function computes a key as opposed to containing a key.

These hiding functions are generated by executable kh.exe and will be described further in this section. In addition, $k_1(), ..., k_{48}()$ are defined as inlined (macro-substitutions). In other words, once the encryption/decryption code is compiled, it will contain function bodies instead of function calls. We may also consider to define encryptWithSubkey() functions as inlined.

At this point we achieved the following:

- The encryption/decryption 3DES key is not present in the code (key hiding).
- Not only the key is not present in the code, but the 48 subkeys, from which the key can be reconstructed, are also not present in the code. What we have instead is pieces of math code, which compute the subkeys. So, subkeys are only present in memory at the time of encryption/decryption.
- The compiled code for the 3DES algorithm becomes a long chain of math operations, in which it becomes difficult to distinguish between a computation of a subkey and 3DES encryption with that subkey. In other words, it even

becomes difficult to extract subkeys from memory. An additional difficulty comes from the fact that hiding functions might have different lengths, which depend on the subkey, and might contain math code, which looks like 3DES encryption code.

### 8.2.2 Generating hiding functions

kh.exe generates hiding functions for each given subkey. This process consists of the following steps:

**Step 1.** A subkey, which is a 48 bit integer, is converted to a word $w$ in a chosen braid group $B_n$.

There is of course a variety of ways in which this can be done. Also, if we would like to make the key hiding algorithm harder to break, we could choose $B_n$ and generate $w$ using the same parameters as recommended by the AAG crypto-system (see [62]) of even stronger. In the current implementation we used weaker parameters than in the AAG, though of course, one can easily specify different parameters in the source code.

Here's how we generate $w$. The group we are working with is $B_5$, which means that we have 5 generators and 5 inverses. The first 32 bits of the subkey is used to generate word $a$ and the last 16 bits are used to generate word $b$. These words are generated as follows:

- each bit corresponds to a letter in $B_5$ or its inverse;
- if a bit has value 1 it is converted to a randomly chosen (by uniform pseudo-random number generator) letter in $B_5$;
- if a bit has value 0 it is converted to a randomly chosen inverse of a letter in $B_5$;

We apply the rules above to the bits from left to right. At the end we will have word $a$ of length 32 and word $b$ of length 16. Then we construct word $w$ as the product:

$$w = a^{-1}ba$$

$w$ is constructed in a similar way to the AAG scheme in order to increase the number of handles, or in other words, to ensure that $w$ is well scrambled by the Dehornoy algorithm used in the next step.

**Step 2.** A Dehornoy form of $w$ (see section 7.1.1) is computed. Let us call it $d$ and the sequence of rewriting rules from $w$ to $d$ by $D$:

$$D = (r_1, r_2, ..., r_n)$$

Next we can compute the reverse of each of the rewriting rule and therefore compute the reverse sequence, which can rewrite $d$ into $w$. Let us call the reverse sequence $D^{-1}$:

$$D^{-1} = (r_n^{-1}, ..., r_1^{-1})$$

**Step 3.** Function $k_i()$ is generated.

$k_i()$ does the following:

1. keeps the value $d$;
2. applies $D^{-1}$ to $d$, therefore computing $w$;
3. converts $w$ to the value of $k_i$ (the reverse of the algorithm from Step 1) and returns it.

### 8.2.3 Security of the key hiding method

In section 8.2.1 we discussed the measures taken to make it difficult for a hacker to debug the encryption/decryption routine in order to recover hidden keys. Here we would like to say a few words about how strong the system is against cryptanalysis.

In essence the strength of the system is based on the fact that inverting Dehornoy form would mean an inverting an one-way function. We were able to invert a sequence of Dehornoy rewriting rules only because we had the sequence itself, i.e. the sequence, which takes $w$ to $d$. We just followed one particular path in a tree of all possible paths. If, however, we are just given $w$ and $d$, then finding a way from $d$ to $w$ by brute force would mean traversing all possible paths. Let us estimate the size of such a tree.

Let us consider a word $w$ of length $l$ in a braid group with $n$ generators. The Dehornoy algorithm removes handles. The number of handles removed for the words of the form used in this scheme is about $O(l)$ [68]. Going back from $d$ to $w$ we will need to insert handles. Inserting one handle would mean choosing a position for the first letter of the handle, a position for the last letter of the handle and the first letter of the handle. This would mean $O(l^2n)$ possibilities. Traversing all paths of length $O(l)$ would mean enumerating $O((l^2n)^l)$ possibilities. For the parameters similar to the parameters of the AAG scheme this value is at least $10^{6000}$, and even for much weaker parameters used in the current implementation, this value is at least $10^{100}$ - quite impossible to enumerate.

Of course, the brute force attack is not the only attack possible. However, choosing parameters similar to a known crypto-system like the AAG crypto-system [62], we can make sure that our system is at least as strong. There are, in fact, number of attacks on the AAG crypto-system [66, 67]. However these attacks only challenge some particular parameters and the system itself is still not broken.

In addition, we would like to refer to [68] for the evaluation on how well the data

is scrambled by the Dehornoy algorithm and how large is its data leakage.

### 8.2.4 Improving performance

The Dehornoy algorithm is very efficient in practice [59], however it may not be efficient enough if we run it every time we need access to a subkey. For example, to encrypt 1 MB of data we will need to recompute 8 million subkeys. This will put a large overhead on the encryption time, where the latter needs to be not more than a few seconds. To improve the performance, we can use only a subset of Dehornoy rules, say 30, instead of the full sequence from $w$ to $d$. And we can further optimize the execution of these rules.

A more dramatic improvement in performance can be achieved if we recompute the subkeys only a portion of the time instead of every time when we need them. In the mean time the subkeys can be kept in memory. The current implementation supports two ways, in which this can be done. One way is to recompute the subkeys only once per $n$ DES blocks, the other is to recompute them every $t$ seconds. The current parameters are 2048 blocks and 0.1 seconds. After 0.1 seconds are expired or 2048 block have been encrypted the subkeys will be recomputed and stored in a different memory location. The experiments with a variety of different file sizes showed that the encryption time increases by no more than 10% if this key hiding method is used.

# Bibliography

[1] Magnus home page: http://zebra.sci.ccny.cuny.edu/web/

[2] T. Mitchell. Machine learning, 249-274. McGraw-Hill Companies, Inc. 1997

[3] M.Mitchell. An introduction to genetic algorithms. The MIT Press, Cambridge, MA, 1998.

[4] L.J.Eshelman, J.D.Schaffer. Preventing premature convergence in genetic algorithms by preventing incest.

[5] E.Patrick, Fundamentals of Pattern Recognition, Prentice Hall Inc., 1972.

[6] K.Fukunaga, Introduction to Statistical Pattern Recognition, Academic Press Inc., 1990.

[7] E. Schalkoff, Pattern Recognition, John Wiley and Sons Inc., 1992.

[8] F.Baader, J.H.Siekmann. Unification Theory, in Handbook of Logic in Artificial Intelligence and Logic Programming, Vol.2, (D.Gabbay et al, ed.). Clarendon Press, Oxford, 1994.

[9] J.I.Hmelevskiĭ. Equations in free semigroups, Trudy Mat. Inst. Steklov. 107, 1971. English translation: Proc. Steklov Inst. Math. 107, 1971.

[10] Yu. Matiyasevich. A connection between systems of word and length equations and Hilbert's Tenth Problem (in russian), 1968. English translation in: Seminars in Mathematics, V.A.Steklov Mathematical Institute, 8, pp. 61-67, 1970.

[11] Equations in Free Monoids, in Automata Languages and Programming (M. Nivat ed.), North Holland, pp. 67-85, 1972.

[12] G.D.Plotkin, Building-in equational theories, Mach. Int. 7, pp. 73-90, 1972.

[13] J. Siekmann, A modification of Robinson's Unification Procedure, M.Sc. Thesis, 1972.

[14] G.S. Makanin, The problem of solvability of equations in a free semigroup. Mat. Sbornik 103, pp. 147-236, 1976 (in Russian). English translation in Math. USSR Sbornik 32(2), pp. 129-198, 1977.

[15] G.S. Makanin. Equations in a free group. Izvestiya NA SSSR 46, pp. 1199-1273, 1982 (in Russian). English translation in Math USSR Izvestiya, Vol.21, No.3, 1983.

[16] J.Jaffar, Minimal and Complete Word Unification. Journal ACM, Vol.37, No.1. January 1990, pp. 47-85.

[17] A.A. Razborov, On systems of equations in a free group, Izvestiya AN SSSR 48, pp. 779-832, 1984 (in Russian). English translation in Math. USSR Izvestiya 25, pp. 115-162, 1985.

[18] J.P.Péruchet. Equations avec constantes et algoritme de Makanin, Thèse de doctorat, Laboratoire d'informatique, Rouen, 1981.

[19] H.Abdulrab, Résolution d'équations sur les mots: étude et implémentation LISP de l'algoritme de Makanin, Ph.D. dissertation, Univ. Rouen, Rouen, 1987.

[20] K.U. Schulz. Makanin's Algorithm for Word Equations: two improvements and a generalization, in Schulz, K.U. ed., Word Equations and related topics, LNCS 572, pp. 85-150, 1990.

[21] K.U. Schulz. Word Unification and Transformation of Generalized Equations. Journal of Automated Reasoning 11, pp. 149-184, 1993.

[22] A. Kościelski, L. Pacholski. Complexity of Makanin's Algorithm. Journal of the ACM, Vol.43, July 1996, pp. 670-684.

[23] A. Kościelski, L. Pacholski. Makanin's algorithm is not primitive recursive, Theoretical Computer Science 191, pp. 145-156, 1998.

[24] C. Gutiérrez. Solving Equations in Strings: On Makanin's Algorithm, in Proceedings of LATIN'98, Third Latin American Symposium on Theoretical Informatics, Campinas, Brazil, April 1998, C. Lucchesi, A. Moura, ed., LNCS 1380, pp. 358-373.

[25] C. Gutiérrez. Satisfiability of Word Equations with Constants is in Exponential Space, in Proc. FOCS 98.

[26] V. Diekert, Makanin's Algorithm for Solving Word Equations with Regular Constraints, (Preliminary version of the chapter in M. Lothaire, *Algebraic Combinatorics on Words.*) Report Nr.1998/02, Fakultät Informatik, Universität Stuttgart.

[27] W. Plandowski. Satisfiability of Word Equations with Constants is in PSPACE, in. Proc. FOCS'99.

[28] R.C.Lyndon. Equations in free groups. Trans. Amer. Math. Soc., 96:445-457, 1960.

[29] K. Appel. One-variable equations in free groups. Proc. Amer. Math. Soc., 19, 912-918, 1968

[30] L.P.Comerford and C.C. Edmunds. Quadratic equations over free groups and free products. Journal of Algebra, 68:276-297, 1981.

[31] L.P.Comerford Jr. and C.C. Edmunds. Solutions of equations in free groups. Walter de Gruyter, Berlin, New York, 1989.

[32] G.S.Makanin. Equations in a free group (Russian). Izv. Akad. nauk SSSR, Ser. Mat., 46:1199-1273, 1982 transl. in Math USSR Izv., V.21, 1983; MR 84m:20040.

[33] O.Kharlampovich and A.Myasnikov. Irreducible affine varieties over a free group. 2: systems in triangular quasi-quadratic form and description of residually free groups. 1998.

[34] J. Comerford and Y. Lee,"Product of two commutators as a square in a free group," CANAD. MATH. BULL. 32(2) (1990), 190 - 196

[35] J.A.Comerford, L.P. Comerford Jr., C.C.Edmunds. Powers as products of commutators. Communications in algebra, 19(2), 675-684, 1991.

[36] R.Gilman, A.G.Myasnikov, *One Variable Equations in Free Groups via Context Free Languages*, Contemporary Mathematics, **349**, 2004.

[37] K. Appel, *One-variable equations in free groups.* Proc. Amer. Math. Soc., **19** (1968), 912–918.

[38] J.-M. Autebert, J. Bersteland L. Boasson, *Context-free languages and pushdown automata*, *in* "Handbook of Formal Languages," vol. 1, Springer Verlag, 1997.

[39] I. Chiswell and V. N. Remeslennikov, *Equations in free groups with one variable I*, J. Group Theory, **3** no. 4 (2000), 445–466.

[40] R. Gilman, On the definition of word hyperbolic groups, Mathematische Zeitschrift, **242** (2002) 529-541.

[41] R. Gilman, Formal Languages and their Application to Combinatorial Group Theory, Contemporary Mathematics, American Mathematical Society, to appear.

[42] R. H. Gilman, A. G. Myasnikov, A. Kvaschuk and V. N. Remeslennikov, *Parametric solutions of one-variable equations*, preprint.

[43] S. Ginsburg and E. Spanier, *Bounded ALGOL-like languages*, Trans. Amer. Math. Soc. **113** (1964), 333–368.

[44] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[45] A. Lorenc, *Coefficient-free equations in free groups*, (Russian) Dokl. Akad. Nauk SSSR, **160** (1965), 538–540.

[46] R. Lyndon, *Equations in free groups.* Trans. Amer. Math. Soc., **96** (1960), 445–457.

[47] D. Muller and P. Schupp, Groups, the theory of ends and context-free languages. J. Computer and System Sciences 26 (1983) 295–310.

[48] G. Rozenberg and A. Salomaa eds., Handbook of Formal Languages, vols. 1-3, Springer Verlag, 1997.

[49] M.Cohen, W.Metzler, A.Zimmermann, *What does a basis of* $F(a, b)$ *look like?*, Math. Ann. 257, 1981, 435-445.

[50] J.H.C.Whitehead, *On equivalent sets of elements in a free group*, Annals of Mathematic, 37(4), 1936.

[51] A.D.Miasnikov and A.G.Myasnikov, *Whitehead method and genetic algorithms*, Contemporary Mathematics, Volume 349, 2004.

[52] R.Haralick, A.Miasnikov, A.Myasnikov, *Pattern recognition approaches to solving combinatorial problems in free groups*, Contemporary Mathematics, Volume 349, 2004.

[53] N.Draper, H.Smith, *Applied Regression Analysis*, Wiley, 3rd edition, 1998.

[54] T.Ryan, *Modern Regression Methods*, John Wiley and Sons Inc., 1968.

[55] R.Haralick, A.Miasnikov, A.Myasnikov, *Pattern recognition and minimal words in free groups of rank 2*, Preprint, 2003.

[56] The GNU Scientific Library: http://sources.redhat.com/gsl/ .

[57] J.Nielsen, *Die Isomorphismen der allgemeinen unendlichen Gruppe mit zwei Erzeugenden*, Math. Ann. 78, 1918, 385-397.

[58] R.O.Duda, P.E.Hart, and D.G.Stork, *Pattern Classification*, Wiley-Interscience, 2nd edition, 2000.

[59] P.Dehornoy, A fast method for comparing braids, Advances in Mathematics 125(1997), 200-235.

[60] P.Dehornoy, Braid groups and left distributive operators, Trans. Amer. Math. Soc. 345. No. 1(1994), 115-151.

[61] I.Anshel, M.Anshel, D.Goldfeld, An Algebraic Method for Public-Key Cryptography, Mathematical Research Letters 6 (1999), 287-291.

[62] I.Anshel, M.Anshel, B.Fischer, D.Goldfeld, New Key Agreement Protocols in Braid Group Cryptography, CT-RSA 2001, LNCS 2020 (2001), 13-27.

[63] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, Choonsik Park, New Public-key Cryptosystem using Braid Groups, Crypto 2000.

[64] Alexandre V. Borovik, Alexei G. Myasnikov, and Vladimir Shpilrain, Measuring sets in infinite groups, In: Computational and Statistical Group Theory. Amer. Math. Soc., Contemporary Math. 298 (2002), 21-42.

[65] Alexandre V. Borovik, Alexei G. Myasnikov, and Vladimir Remeslennikov, Multiplicative measures on groups, J.of Algebra and Computation. To appear.

[66] J. Hughes, A Tannenbaum, Length-Based Attacks for Certain Group Based Encryption Rewriting Systems, Workshop SECI02 SEcurit de la Communication sur Intenet, September, 2002, Tunis, Tunisa,

[67] J. Hughes, A Linear Algebraic Attack on the AAFG1 Braid Group Cryptosystem, The 7th Australasian Conference on Information Security and Privacy ACISP 2002, Lecture Notes in Computer Science, vol. 2384, pp. 176–189, Springer-Verlag, New York 2002.

[68] D.Bormotov, Experimenting with Dehornoy forms, Preprint, 2003.