Normal Forms in Artin Groups for Cryptographic Purposes

Renaud Brien

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of Master of Science in Mathematics [1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

© Renaud Brien, Ottawa, Canada, 2012

[1]The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

With the advent of quantum computers, the security of number-theoretic cryptography has been compromised. Consequently, new cryptosystems have been suggested in the field of non-commutative group theory. In this thesis, we provide all the necessary background to understand and work with the Artin groups. We then show that Artin groups of finite type and Artin groups of large type possess an easily-computable normal form by explicitly writing the algorithms. This solution to the word problem makes these groups candidates to be cryptographic platforms. Finally, we present some combinatorial problems that can be used in group-based cryptography and we conjecture, through empirical evidence, that the conjugacy problem in Artin groups of large type is not a hard problem.

Résumé

L'annonce des ordinateurs quantiques a remis en question la sécurité de la cryptographie basée sur la théorie des nombres. Ce faisant, de nouveaux cryptosystèmes ont été suggérés dans le domaine de la théorie des groupes non-abéliens. Dans cette thèse, nous présentons tout d'abord la théorie nécessaire pour comprendre et utiliser les groupes de Artin. Nous montrons ensuite que les groupes de Artin de type fini et de type large possèdent une forme normale qui se calcule facilement en donnant explicitement les algorithmes. Ces solutions au problème de mot font de ces groupes des candidats possible pour des platformes cryptographique. Finalement, nous présentons certain problèmes combinatoires qui peuvent être utilisés en cryptographie sur des groupes et, à l'aide de résultats expérimentaux, nous posons la conjecture que le problème de conjugaison n'est pas un problème difficile dans les groupes de Artin de type large.

Acknowledgements

I would like to thank my supervisors, Monica Nevins and Hadi Salmasian, for all the support and insight they gave me during this project. I would also like to thank examiners Mike Newman and Qiang Wang for reading my thesis in its entirety and giving me useful feedback. I also thank my father for helping me with proofreading, even though Mathematics is not his domain.

Dedication

To my familly, especially my father and my mother, who supported me unconditionally through the high and low of this adventure.

Contents

List of Figures **viii**

Introduction **1**

1 Coxeter Groups **6**

1.1 Basics 7
1.2 Geometry 9
1.3 Root systems 14
1.4 Exchange and Deletion conditions 18
1.5 Finite reflection groups 23
1.6 Some finite reflection groups 28
1.7 Lattice structure 30

2 Artin Groups **35**

2.1 Definitions 35
2.2 Reduced elements 37
2.3 Artin groups of finite type 40
2.4 Equivalence of reduced expressions 53

3 A Normal Form for Artin Groups of Finite Type **59**

3.1 Partial order on $\mathcal{A}$ 59
3.2 Left-weighted factorisation 61

3.3 Algorithm for left-weighted normal form 77
3.4 Proofs of correctness for the meet algorithms 82

4 Normal Form in Artin Groups of Large Type 87
4.1 Definitions and the dihedral case 88
4.2 The normal form 94
4.3 The algorithms 101

5 Application of Normal Forms to Cryptography 109
5.1 Hard problems and group requirements 111
5.2 Some group-based cryptographic protocols 113
5.3 The CSP in Artin groups of large type 116

A Meet Algorithms 123

B Java Implementation of the Algorithm 126
B.1 Basic object classes 126
B.2 Main algorithm 130
B.3 RRS algorithm 130
B.4 LLS algorithm 134
B.5 Testing classes 138

List of Figures

2.1 The trivial braid in $\mathcal{B}_4$. 41

2.2 The braid σ_1 in $\mathcal{B}_4$. 42

2.3 The braid σ_2^{-1} in $\mathcal{B}_4$. 42

2.4 The braid diagram for $\sigma_1\sigma_2\sigma_3^{-1}\sigma_1\sigma_2^{-1}$ in B_4. 43

2.5 The relation $\sigma_1\sigma_1^{-1} = 1$ in $\mathcal{B}_2$. 44

2.6 The relation $\sigma_1\sigma_3 = \sigma_3\sigma_1$ in $\mathcal{B}_4$. 44

2.7 The relation $\sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2$ in $\mathcal{B}_3$. 44

2.8 The fundamental braid Δ in $\mathcal{B}_5$. 45

3.1 Flowchart of the proof of Lemma 3.2.15 74

Introduction

Cryptography has played an important role in history from the moment humanity wished to keep some communications secret. Its evolution has been rather slow, up to the moment computers became more widely used. This increased computational power rendered many old ciphers obsolete. As such new methods of encryption were required. With the ever increasing number of computers connected together, the advent of public key cryptography allowed for secure communications and key exchange between any two parties without requiring the storage of countless secret shared keys.

Many of the different public-key cryptosystems that are widely used, such as RSA and elliptic curve cryptography, are based on number-theoretic schemes such as the prime factorization and the discrete logarithm problems. These problems had been studied for hundreds of years and were hard to solve, even with the increasing computational power of computers, at least until the advent of quantum computers. Indeed, quantum computers were shown to be able to solve these number-theoretic problems quite efficiently, provided the computer was big enough. This made the search for new cryptographic primitives, which are not based on number theory, an important area of research.

An important such alternative is non-commutative group cryptography, which is based on combinatorial group theory. The advantages of group-based cryptography is that it has an abundance of examples and quite a lot is already known about various algorithmic problems such as the conjugacy and word problems. Among the major

papers in this budding area is one by Anshel, Anshel and Goldfeld, [AAG99]. They were the first to suggest the use of the braid groups in cryptography.

Braid groups show up in various areas of mathematics and physics and are as such well known. They also offer an easily-computable normal form and were thought to have a hard conjugacy problem. The first fundamental results on braid groups were pioneered by Artin [Art47] and Chow [Cho48]. In 1969, Garside [Gar69] gave a solution to both the word and conjugacy problems by using reduced braids, although the main downside was that both algorithms had a complexity of $O(n!)$. Using the ideas of Garside, a new normal form was found by William Thurston, [ECH+92], and Elrifai and Morton, [ERM94]. This new normal form has a similar form to Garside's normal form, but it can be computed in quadratic time. Efficient algorithms are given in [CKL+01].

The research for this thesis began with the study of the braid group and the algorithms to compute its normal forms. Understanding the normal form relied heavily on the braid group itself and on the associated permutation group. We then realized that the algorithms presented in [CKL+01] could easily be generalized to Artin groups associated to the finite reflection groups, which are a generalization of the braid groups. This led us to a more in-depth study of Coxeter systems and finite reflection groups culminating in an easy implementation of the normal form in these groups.

The conjugacy problem is conjectured to be cryptographically insecure for randomly generated elements in Artin groups of finite type, following the work of Franco and Gonzalès-Meneses in [FGM03]. Nevertheless, having a computable normal form will allow for the search of other harder problems.

In the course of our research, we then looked at Artin groups of large type, which are characterized by having no pairs of commuting generators, and their normal forms. This normal form has been found only recently in [HR11] and was only described theoretically. We derived an explicit algorithm from their paper and implemented it

in Java. The results obtained by our implementation of this normal form strongly indicate that the Artin groups of large type have an easy conjugacy problem.

This thesis is a streamlined approach to both normal forms. As such, reading each chapter in order will cover all the material required by the two normal forms. Some basic literature we can refer the reader to includes [Hum90], for all the Coxeter systems, [KT08] for the braid group itself and [ECH+92] for the theory of algorithms and automata in groups. Additional references are included throughout the thesis.

The main contributions of the thesis are:

- The proof of Theorem 2.4.5, which extends a key statement about reduced elements from the braid group to all Artin groups. The proof had been given for the braid group in [KT08], but not for the other Artin groups.
- The extension of the normal form for the braid group to other Artin groups of finite type. This is achieved by adapting the algorithm from [CKL+01] to these new groups.
- The proof of a technical statement which is essential in showing that left-weighted factorization is unique. This is Lemma 3.2.15. Garside proved the original version for the Braid group in [Gar69].
- The explicit pseudocode algorithm for the normal form in Artin groups of large type. This was derived from the theoretical map from [HR11]. A Java implementation is also included in the appendix.
- Experimental evidence that the Artin groups of large type are insecure for current cryptographic protocols based on the conjugacy problem.

The main use of the normal forms presented in this thesis is to provide an efficient solution to the *word problem*. This allows us to verify if two elements are equivalent in the group. Constructing a normal form is not the only way to solve the word problem, but it is one of the more convenient solutions.

In Chapter 1, we give an in-depth introduction to Coxeter systems and finite reflection groups including only the results needed to work with the Artin groups. We first give general properties of Coxeter systems and then proceed to build a geometry on these groups as well as to define root systems. We then prove some main results such as the Exchange Condition and the Deletion Condition. We then focus on some finite reflection groups before proving that Coxeter systems have an important lattice structure.

In Chapter 2 we define Artin groups and present some of their properties. We then construct the set of reduced elements for an arbitrary Artin group. This set has the important property of being in bijection with the corresponding Coxeter group and is necessary to the construction of a normal form for some Artin groups. We then introduce Artin groups of finite type and their properties while using the braid group as the standard example. In the last section of this chapter, we prove Theorem 2.4.5 stating that given two reduced expressions for an element in a Coxeter system, one can be obtained from the other without the use of relations of the form $ss^{-1} = 1$. We show that this implies the reduced expressions from the Coxeter group lifts uniquely to elements of the Artin group.

Chapter 3 is dedicated to the construction of an efficient normal form for Artin groups of finite type. We introduce the notions of a left-weighted factorization and of a left-weighted normal form. We then give explicit algorithms for Artin groups associated to the three infinite families of finite reflection groups, those of type A_n, B_n and D_n, and we finish the chapter by giving a proof of correctness of the algorithm.

In Chapter 4 we present an efficient normal form for Artin groups of large type. This normal form is based on the shortlex minimal representative of any element w. In the first part of the chapter, we present the necessary results for Artin groups on two generators and then generalize to any Artin group of large type. In the second part we give explicit algorithms to obtain the normal form.

Chapter 5 provides the necessary background to set the thesis in the crypto-

graphic context. We introduce a list of properties any group should satisfy to be used as platforms, then we define some of the common combinatorial problems considered in non-commutative group cryptography. We then present four cryptographic protocols that have been suggested in the literature and we present how an attacker should proceed to break these protocols. We finish the chapter by a study the cryptographic properties of the normal form built in Chapter 4 and we conjecture that the conjugacy problem is insecure in Artin groups of large type.

Future work in this domain includes finding a subset of braids that have a hard conjugacy problem and finding other combinatorial problems or cryptographic primitives that are hard in the various Artin groups considered in this thesis. It is also an open problem to find a normal form for the remaining Artin groups, such as the Artin groups associated to affine reflection groups.

Chapter 1

Coxeter Groups

The theory of Coxeter systems and finite reflection groups is the first step towards the study of the braid groups and their generalizations, the Artin groups. Indeed, understanding how the Coxeter groups work will give us great insight on what happens inside the Artin groups. We will prove most of the results stated in this chapter. The missing proofs are readily available in the literature.

In Section 1.1 we introduce Coxeter systems and then proceed to build a geometry on them in Section 1.2. In Section 1.3 we define the root systems of a Coxeter system and we use them to prove the Exchange and Deletion conditions in Section 1.4. Section 1.5 covers results on finite reflection groups, while we present, in Section 1.6, some finite reflection groups. We finish this chapter, in Section 1.7, by showing that any Coxeter system admits a semi-lattice structure. Sections 1.1 to 1.5 are largely based on [Hum90] and Section 1.7 is based on [BB05].

1.1 Basics

Definition 1.1.1. A *Coxeter system* is a pair (W, S), where W is a group and $S \subset W$ is a set of generators together with the relations

$$(ss')^{m(s,s')} = 1,$$

with $m(s, s) = 1$ and $m(s, s') = m(s', s) \geq 2$ for $s \neq s'$. If $(ss')^n \neq 1$ for every $n \neq 0$, we say by convention that $m(s, s') = \infty$.

Having $m(s, s) = 1$ gives us that $s^2 = 1$.

The *rank* of (W, S) is the cardinality of S. Let S^* be the free group on S and let $N \subset S^*$ be the normal subgroup generated by all the elements of the form $(ss')^{m(s,s')}$. Then, we have that $W = S^*/N$ as a quotient group.

Given the definition of W by generators and relations or as a quotient group, it is unclear if W is trivial of not. That is, if we consider the canonical map $S \subset S^* \to W$, it is not clear if the map is injective. It is also unclear if $m(s, s')$ is the order of ss' in W. Fortunately, W is non trivial, the $m(s, s')$ are the order of ss', and all the $s \in S$ are mapped to distinct elements by the canonical map $S^* \to W$, as we shall see in Proposition 1.2.8.

Throughout this chapter, we will assume S to be finite, although we do note that most results are still true for infinite S. Since W is assumed to be finitely generated, we will label the elements of S by $s_1, \ldots, s_n$ and we write $m(i, j)$ for $m(s_i, s_j)$.

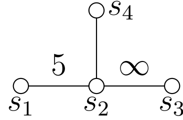
The relations are then a set of values $m(i, j) \in \mathbb{N} \cup \{\infty\}$ with $m(i, i) = 1$ and $m(i, j) \geq 2$ for $i \neq j$. We sometimes give the $m(i, j)$'s in a symmetric matrix, where the entries $m_{i,j}$ are exactly the values $m(i, j)$.

Equivalently, a Coxeter system (W, S) can be described by an undirected *Coxeter graph* where the set of vertices is S and where the vertices s and s' are joined by an edge labelled by $m(s, s')$ whenever $m(s, s') \geq 3$. By convention, we do not write the

label on the edge when $m(s, s') = 3$ and if two distinct vertices s and s' are not joined by an edge, then $m(s, s') = 2$.

Trivially, given two Coxeter systems (W, S) and (W', S'), the groups W and W' are isomorphic whenever their Coxeter graphs are isomorphic.

Example 1.1.2. The Coxeter graph



describes the Coxeter system (W, S) were $S = \{s_1, s_2, s_3, s_4\}$ and

$$\begin{aligned} m(1,3) = m(1,4) = m(3,4) = 2, & \quad m(1,2) = 5, \\ m(2,3) = \infty & \quad m(2,4) = 3. \end{aligned}$$

As can be seen, drawing the Coxeter graph is an efficient way to encode a Coxeter system.

In general, we omit the labels of the nodes on a Coxeter graph, since they depend only on how we labeled S.

Proposition 1.1.3. *There is a unique group homomorphism $\epsilon : W \to \{-1, 1\}$ sending every $s \in S$ to -1.*

Proof: We start with the homomorphism $\epsilon : S^* \to \{-1, 1\}$, generated by $s \mapsto -1$. Since all the elements of the form $(ss')^{m(s,s')}$ are mapped to 1, this map factors uniquely to the desired homomorphism $\epsilon : W \to \{-1, 1\}$. ∎

We now introduce a notion of length on W. Given any element $w \in W$, we can write $w = s_{i_1} \cdots s_{i_r}$, a *word* for w in the alphabet S. If r is the smallest integer among all such expressions for w, we call it the *length of* w, denoted $\lambda(w)$, and the word $s_{i_1} \cdots s_{i_r}$ is called a *reduced expression* for w. By convention, we put $\lambda(1) = 0$.

Proposition 1.1.4. *Let $w, w' \in W$. The length function has the following properties:*

1. $\epsilon(w) = (-1)^{\lambda(w)}$,
2. $\lambda(w) = \lambda(w^{-1})$,
3. $\lambda(w) = 1$ *if and only if* $w \in S$,
4. $\lambda(w) - \lambda(w') \leq \lambda(ww') \leq \lambda(w) + \lambda(w')$,
5. $\lambda(sw) = \lambda(w) \pm 1$ *for all* $s \in S$,
6. $\lambda(ws) = \lambda(w) \pm 1$ *for all* $s \in S$.

Proof: Properties 1 to 4 are straightforward, so we will only prove properties 5 and 6. From properties 3 and 4, we have that $\lambda(w) - 1 \leq \lambda(ws) \leq \lambda(w) + 1$. But we also have that $\epsilon(sw) = \epsilon(s)\epsilon(w) = -\epsilon(w)$, so $\lambda(sw) \neq \lambda(w)$. It follows that $\lambda(sw) = \lambda(w) \pm 1$. The argument is identical for ws. ∎

Finally, given a reduced expression $s_{i_1} \cdots s_{i_r}$ it is easy to see that any subexpression $s_{i_j} \cdots s_{i_k}$ is reduced for any $1 \leq j \leq k \leq r$.

1.2 Geometry

The study of general Coxeter systems is an extension of the study of groups generated by reflections in Euclidean spaces. As such it is natural to build a geometry for any Coxeter system (W, S).

Definition 1.2.1. Given a vector space V, a symmetric bilinear form $B : V \times V \to \mathbb{R}$ and a non-zero vector $\alpha \in V$ such that $B(\alpha, \alpha) \neq 0$, a *reflection* in α is a map sending α to its negative while fixing $H_\alpha = \{v | B(v, \alpha) = 0\}$, the hyperplane orthogonal to α. We denote this reflection by σ_α.

Remark 1.2.2. We remark that if V, B and $\alpha \in V$, with $B(\alpha, \alpha) \neq 0$, are fixed, the reflection σ_α is unique. Indeed, if the dimension of V is n, then the dimension of H_α is $n - 1$ and so for any $v \in V$ we can write $v = v_h + c\alpha$, where $v_h \in H_\alpha$. Thus,

$\sigma_\alpha(v) = \sigma_\alpha(v_h + c\alpha) = v_h + c\sigma_\alpha(\alpha) = v_h - c\alpha$ by linearity. It follows that the property of sending α to $-\alpha$ while fixing H_α uniquely determines a reflection.

We thus see that if we are to define a geometry for (W, S), we need to define a bilinear form B on V. We begin the construction by taking $V = \mathbb{R}^{|S|}$ with a basis $\{\alpha_s | s \in S\}$ in bijection with S. We define a symmetric bilinear form B by

$$B(\alpha_s, \alpha_{s'}) = -\cos\left(\frac{\pi}{m(s, s')}\right),$$

when $m(s, s') < \infty$ and by $B(\alpha_s, \alpha_{s'}) = -1$ when $m(s, s') = \infty$. Since $m(s, s) = 1$, we have $B(\alpha_s, \alpha_s) = 1$ and it follows that all the α_s are unit vectors that are not self-orthogonal. If $S = \{s_1, s_2, \ldots, s_n\}$, we can write this bilinear form as a symmetric matrix B, where $B_{i,j} = B(\alpha_{s_i}, \alpha_{s_j})$.

The first important geometric characteristic of W we obtain from B is the angle between α_s and $\alpha_{s'}$. Indeed, we define the *angle*, θ, between α_s and $\alpha_{s'}$ by

$$B(\alpha_s, \alpha_{s'}) = -\cos\left(\frac{\pi}{m(s, s')}\right) = \cos\left(\pi - \frac{\pi}{m(s, s')}\right) = |\alpha_s||\alpha_{s'}|\cos(\theta),$$

But we know that α_s and $\alpha_{s'}$ are unit vectors, so it follows that $\theta = \pi - \frac{\pi}{m(s,s')}$. We thus obtain that the hyperplanes H_s and $H_{s'}$ meet at an angle of $\frac{\pi}{m(s,s')}$.

Remark 1.2.3. The geometry we build will not be Euclidean unless the bilinear form B is positive definite, that is if $B(u, u) > 0$ for all non-zero $u \in V$. The notion of angle can thus be counterintuitive. For example, in the case where $m(s, s') = \infty$, we have that α_s and $\alpha_{s'}$ are two linearly independent vectors making an angle of π.

Now, for each $s \in S$, we consider the map $\sigma_s : V \to V$ by

$$\sigma_s(v) = v - 2B(\alpha_s, v)\alpha_s.$$

Let us verify that the map σ_s is a reflection in the sense of Definition 1.2.1. We first compute

$$\sigma_s(\alpha_s) = \alpha_s - 2B(\alpha_s, \alpha_s)\alpha_s = \alpha_s - 2\alpha_s = -\alpha_s.$$

Next, we let $v \in H_s$ and compute

$$\sigma_s(v) = v - 2B(\alpha_s, v)\alpha_s = v - 0 = v.$$

Therefore σ_s is the reflection in α_s.

Definition 1.2.4. If $s \in S$, the reflection σ_s is called a *simple reflection.*

The next proposition gives us that the geometry is preserved by these reflections.

Proposition 1.2.5. *The reflection σ_s preserves the bilinear form B, that is for all $u, v \in V$, $B(\sigma_s(u), \sigma_s(v)) = B(u, v)$.*

Proof: Let $u, v \in V$ and compute

$$\begin{aligned} B(\sigma_s(u), \sigma_s(v)) &= B(u - 2B(\alpha_s, u)\alpha_s, v - 2B(\alpha_s, v)\alpha_s) \\ &= B(u, v) - 4B(\alpha_s, v)B(u, \alpha_s) + 4B(\alpha_s, u)B(\alpha_s, v) \\ &= B(u, v). \end{aligned}$$

∎

Proposition 1.2.6. *For any $s \in S$, we have that $\sigma_s^2 = 1$.*

Proof: Let $s \in S$ and let $v \in V$. We have

$$\begin{aligned} \sigma_s^2(v) &= \sigma_s(v - 2B(\alpha_s, v)\alpha_s) \\ &= \sigma_s(v) - 2B(\alpha_s, v)\sigma_s(\alpha_s) \\ &= (v - 2B(\alpha_s, v)\alpha_s) - 2B(\alpha_s, v)(-\alpha_s) \\ &= v. \end{aligned}$$

∎

Example 1.2.7. Consider the Coxeter group $I_2(m)$, arising from the Coxeter system (W, S) given by

$$\overset{m}{\underset{s_1}{\circ}\!\!-\!\!\!-\!\!\underset{s_2}{\circ}}$$

where $m < \infty$. We will build its geometry in the same way as above. By the previous construction, we take $V = \mathbb{R}^2$ with basis $\{\alpha_1, \alpha_2\}$ and $B(\alpha_i, \alpha_j) = -\cos\left(\frac{\pi}{m(i,j)}\right)$. We first observe that B is positive definite in this situation. Indeed, let $u = a\alpha_1 + b\alpha_2$ for $a, b \in \mathbb{R}$ and compute

$$\begin{aligned} B(u,u) &= a^2B(\alpha_1,\alpha_1) + 2abB(\alpha_1,\alpha_2) + b^2B(\alpha_2,\alpha_2) \\ &= a^2 - 2ab\cos(\tfrac{\pi}{m}) + b^2 \\ &= (a^2 - 2ab\cos(\tfrac{\pi}{m}) + b^2\cos(\tfrac{\pi}{m})^2) - b^2\cos(\tfrac{\pi}{m})^2 + b^2 \\ &= (a - b\cos(\tfrac{\pi}{m}))^2 + b^2(1 - \cos(\tfrac{\pi}{m})^2) \\ &= (a - b\cos(\tfrac{\pi}{m}))^2 + b^2\sin^2(\tfrac{\pi}{m}) \geq 0. \end{aligned}$$

If $\sin(\frac{\pi}{m}) \neq 0$, which is the case whenever $m < \infty$, then $B(u,u) > 0$ for all $u \in V - \{0\}$. In this example m is finite, so we are working in the familiar Euclidean plane. Let σ_1 and σ_2 be the reflections associated to α_1 and α_2 respectively. These reflections are made about the lines H_1 and H_2, which cross each other at an angle of $\frac{\pi}{m}$.

The transformation $\sigma_1\sigma_2$ is then a reflection about the line H_2 followed by a reflection about the line H_1, which can be shown to be a rotation of angle $\frac{2\pi}{m}$ in the Euclidean plane. It follows that the transformation $\sigma_1\sigma_2$ has order m in $GL(V)$.

Now let G be the group generated by $\{\sigma_1, \sigma_2\}$. Since $\sigma_i^2 = 1$, we can see that G is also generated by $\{\sigma_1, \sigma_1\sigma_2\}$. It follows that G a group generated by a reflection of order 2 and a rotation of order m that preserves the geometry of V. But such a group is exactly the dihedral group of order $2m$: $\mathcal{D}(2m)$.

Thus, we have that the Coxeter group $I_2(m)$, with $m < \infty$, is the familiar dihedral group of orthogonal transformations that preserve the regular m-gon.

Proposition 1.2.8. *Given a Coxeter system (W, S) and distinct elements $s, s' \in S$ then*

1. *s and s' are distinct elements in W,*
2. *the order of ss' in W is $m(s,s')$.*

Proof: Let $V = \mathbb{R}^{|S|}$ as above. We show this result by showing that there exists a group homomorphism $\sigma : W \to GL(V)$ sending $s \in S$ to σ_s. We first note that if $s \neq s'$ in S, then α_s and $\alpha_{s'}$ are two linearly independent vectors in V and, as such, $\sigma_s \neq \sigma_{s'}$ in $GL(V)$.

Next, we compute the order of $\sigma_s\sigma_{s'}$ in $GL(V)$. If $s = s'$ then $\sigma_s\sigma_{s'} = \sigma_s^2 = 1$. Suppose $s \neq s'$, put $m = m(s,s')$ and consider the subspace $V_{s,s'}$ generated by α_s and $\alpha_{s'}$. We observe that σ_s and $\sigma_{s'}$ act on $V_{s,s'}$ while fixing its complement via B, $H_{\alpha_s} \cap H_{\alpha_{s'}}$. It follows that the order of $\sigma_s\sigma_{s'}$ is the same in $V_{s,s'}$ and in V.

If m is finite, we have seen in Example 1.2.7 that B is positive definite on $V_{s,s'}$ and that $\sigma_s\sigma_{s'}$ is a rotation through angle $\frac{2\pi}{m}$. We then have that $\sigma_s\sigma_{s'}$ has order $m(s,s')$ in $V_{s,s'}$ and thus in V.

If $m = \infty$, we have that $B(\alpha_s, \alpha_{s'}) = -1$. Consider $u = \alpha_s + \alpha_{s'}$. We observe that $B(u,\alpha_s) = B(u,\alpha_{s'}) = 0$. Indeed, we can compute

$$B(u,\alpha_s) = B(\alpha_s,\alpha_s) + B(\alpha_{s'},\alpha_s) = 1 - 1 = 0.$$

We thus have that both σ_s and $\sigma_{s'}$ fix u. We now consider

$$\begin{aligned} \sigma_s\sigma_{s'}(\alpha_s) &= \sigma_s(\alpha_s - 2B(\alpha_s,\alpha_{s'})\alpha_s') \\ &= \sigma_s(\alpha_s + 2\alpha_{s'}) \\ &= \sigma_s(u + \alpha_{s'}) \\ &= u + \alpha_{s'} + 2\alpha_s \\ &= 2u + \alpha_s. \end{aligned}$$

It is then clear that $(\sigma_s\sigma_{s'})^k(\alpha_s) = 2ku + \alpha_s$ for all $k \in \mathbb{Z}$ and it follows that the order of $\sigma_s\sigma_{s'}$ is infinite in $V_{s,s'}$ and thus in V.

We have just shown that the subgroup of $GL(V)$ generated by the σ_s, with $s \in S$, satisfies the same relations as the Coxeter system (W,S). It follows that $\sigma : W \to GL(V)$, with $s \mapsto \sigma_s$, is a group homomorphism.

Since $\sigma_s \neq \sigma_{s'}$ whenever $s \neq s'$, we have that s and s' are sent to distinct elements by the canonical map $S^*/N \to W$. Thus, if s, s' are distinct in S, they are distinct in S^* and in W.

Finally, we obtain that $m(s,s')$ is the order of ss'. Indeed, we have that the order, d, of ss' in W divides $m(s,s')$. But the order of $\sigma_s\sigma_{s'}$, $m(s,s')$, must divide d since σ is a group homomorphism. Thus it follows that $d = m(s,s')$ is the order of ss' in W. ∎

This last theorem gives us that any Coxeter system (W,S) is non-trivial whenever S is non-empty. From this proof it is still unknown if σ is faithful, but this will be shown in the next section.

1.3 Root systems

Now that we have a geometric representation σ for W, we are able to obtain more properties of the length function λ. From this point on, we write $w(\alpha_s)$ instead of $\sigma(w)(\alpha_s)$, since W acts on V through σ.

Definition 1.3.1. The *root system* Φ of W is the set

$$\Phi = \{w(\alpha_s) | w \in W, s \in S\}.$$

An element $\alpha \in \Phi$ is called a *root*. The roots α_s, for any $s \in S$, are called *simple roots*.

Because $s(\alpha_s) = -\alpha_s$, we clearly have that $\Phi = -\Phi$. Moreover, since the α_s were chosen to form a basis for V, we observe that for any root $\alpha \in \Phi$,

$$\alpha = \sum_{s \in S} c_s \alpha_s$$

where the $c_s \in \mathbb{R}$ are uniquely determined.

Definition 1.3.2. A root $\alpha \in \Phi$ is said to be *positive* (resp. *negative*) if for all $s \in S$, $c_s \geq 0$ (resp. $c_s \leq 0$). We denote it by $\alpha > 0$ (resp. $\alpha < 0$).

We denote the set of all positive (resp. negative) roots by Φ^+ (resp. Φ^-).

With these definitions in hand, we can state an important theorem about the length function. We direct the reader to [Hum90, Section 5.4] for the proof.

Theorem 1.3.3. *Let $w \in W$ and $s \in S$. If $\lambda(ws) > \lambda(w)$, then $w(\alpha_s) > 0$. If $\lambda(ws) < \lambda(w)$, then $w(\alpha_s) < 0$.*

Corollary 1.3.4. *A root $\alpha \in \Phi$ is either positive or negative.*

Proof: We first observe that every simple root α_s is clearly positive. Let $\alpha = w(\alpha_s)$ for some $w \in W$, $s \in S$. We then consider ws. If $\lambda(ws) < \lambda(w)$, then $\alpha = w(\alpha_s) < 0$ by Theorem 1.3.3, making α negative. On the other hand, if $\lambda(ws) > \lambda(w)$, then $\alpha = w(\alpha_s) > 0$, making α positive. Since by Proposition 1.1.4 it is impossible for $\lambda(ws)$ to be equal to $\lambda(w)$, we have that α is either positive or negative. ∎

Remark 1.3.5. Theorem 1.1.4 can be restated as follows: if $w \in W$ and $s \in S$, then

$$\lambda(ws) > \lambda(w) \iff w(\alpha_s) > 0.$$

Indeed, by Corollary 1.3.4, either $w(\alpha_s) > 0$ or $w(\alpha_s) < 0$ and the result follows directly.

A second corollary of Theorem 1.3.3 is in link with the geometry we built.

Corollary 1.3.6. *The representation $\sigma : W \to GL(V)$ is faithful.*

Proof: Let $w \in \ker(\sigma)$ be such that $w \neq 1$. If $w \neq 1$, there exists some $s \in S$ such that $\lambda(ws) < \lambda(w)$. By Theorem 1.3.3, this implies $w(\alpha_s) < 0$, but $w(\alpha_s) = \alpha_s > 0$ since $w \in \ker(\sigma)$. This is a contradiction, thus $\ker(\sigma) = \{1\}$ and σ is faithful. ∎

We can now take a closer look at how W acts on V.

Proposition 1.3.7. *If $s \in S$, then s sends α_s to $-\alpha_s$, but permutes the other positive roots.*

Proof: Let $s \in S$ and let $\alpha \neq \alpha_s$ be a positive root. The first part of the statement is already known, so we focus on the second part. Since $\alpha = w(\alpha_{s'})$ for some $w \in W$ and $s' \in S$, α is a unit vector and is thus not a multiple of α_s. We can then write

$$\alpha = \sum_{t \in S} c_t \alpha_t$$

where all the c_t's are nonnegative and where $c_t \neq 0$ for some $t \in S \backslash \{s\}$. Now, recall that $s(\alpha) = \sigma_s(\alpha)$ and that

$$s(\alpha) = \alpha - B(\alpha, \alpha_s)\alpha_s = \sum_{t \neq s} c_t \alpha_t + (c_s - B(\alpha, \alpha_s))\alpha_s.$$

Thus, the only coefficient of α that is modified by s is c_s and the remaining coefficients are still nonnegative, since we only subtracted a multiple of α_s. Then, $s(\alpha)$ is not a negative root and is clearly distinct from α_s. Therefore, $s(\Phi^+ \backslash \{\alpha_s\}) \subset \Phi^+ \backslash \{\alpha_s\}$. The reverse inclusion is obtained by applying s on both sides to obtain $s^2(\Phi^+ \backslash \{\alpha_s\}) = \Phi^+ \backslash \{\alpha_s\} \subset s(\Phi^+ \backslash \{\alpha_s\})$. It follows that $s(\Phi^+ \backslash \{\alpha_s\}) = \Phi^+ \backslash \{\alpha_s\}$, proving the desired result. ∎

The next proposition will tie the length function to the geometric representation of a Coxeter system.

Proposition 1.3.8. *For any $w \in W$, the length of w, $\lambda(w)$, is equal to the number of positive roots sent to negative roots by w.*

Proof: Let $w \in W$ and define $n(w)$ to be the number of positive roots sent to negative roots by w. We can write

$$n(w) = |\Pi(w)|, \text{ where } \Pi(w) = \Phi^+ \cap w^{-1}(\Phi^-).$$

Our goal is to show that $\lambda(w) = n(w)$.

We claim that for any $s \in S$ and $w \in W$, $n(ws) = n(w) \pm 1$. Namely if $w(\alpha_s) > 0$, then $ws(\alpha_s) < 0$ and $\alpha_s \notin \Pi(w)$. Furthermore, Proposition 1.3.7 gives us that s stabilizes $\Phi^+ \backslash \{\alpha_s\}$ and thus it follows that $\Pi(ws) = \Pi(w) \cup \{\alpha_s\}$. As such, we have that $n(ws) = n(w) + 1$ whenever $w(\alpha_s) > 0$.

On the other hand, if $w(\alpha_s) < 0$, then $ws(\alpha_s) > 0$ and $\alpha_s \in \Pi(w)$. We then get that $\Pi(ws) = \Pi(w) \backslash \{\alpha_s\}$ and that $n(ws) = n(w) - 1$ whenever $w(\alpha_s) < 0$.

We now proceed to show that $\lambda(w) = n(w)$ by induction on $\lambda(w)$. If $\lambda(w) = 0$, then $w = 1$ and it is clear that $n(w) = 0$. If $\lambda(w) = 1$, then $w = s$ for some $s \in S$ and $n(w) = 1$ by Proposition 1.3.7. Now take $w \in W$ such that $\lambda(w) = n(w)$ and let $s \in S$. By Theorem 1.3.3 and Proposition 1.1.4, we have that $\lambda(ws) = \lambda(w) + 1$ whenever $w(\alpha_s) > 0$ and $\lambda(ws) = \lambda(w) - 1$ whenever $w(\alpha_s) < 0$. Yet, we just proved that, respectively, in these cases $n(ws) = n(w) + 1$ and $n(ws) = n(w) - 1$. Thus we have $\lambda(ws) = n(ws)$ for any $w \in W$ and $s \in S$ by induction. ∎

We know that $s \in S$ sends the simple root α_s to $-\alpha_s$ through the reflection σ_s, but we could also look at the reflection sending any root $\alpha = w(\alpha_s)$ to its negative. We observe that such a reflection is given by the transformation wsw^{-1}. Indeed, for

any $u \in V$ we compute

$$\begin{aligned} wsw^{-1}(u) &= w(\sigma_s(w^{-1}(u))) \\ &= w(w^{-1}(u) - 2B(w^{-1}(u), \alpha_s)\alpha_s) \\ &= ww^{-1}(u) - 2B(w^{-1}(u), \alpha_s)w(\alpha_s) \\ &= u - 2B(ww^{-1}(u), w(\alpha_s))w(\alpha_s) \\ &= u - 2B(u, w(\alpha_s))w(\alpha_s) \\ &= u - 2B(u, \alpha)\alpha. \end{aligned}$$

The transformation wsw^{-1} thus stabilizes the plane H_α and sends α to $-\alpha$ and we conclude that wsw^{-1} is the reflection about α. We can thus associate a reflection s_α to any root $\alpha \in \Phi$ with the desired property that $s_\alpha = s_{-\alpha}$.

Lemma 1.3.9. *If the roots $\alpha, \beta \in \Phi$ are such that, for some $w \in W$, $\beta = w(\alpha)$, then $s_\beta = ws_\alpha w^{-1}$.*

Proof: From the previous computation, we can observe that for any reflection s_α, the transformation $ws_\alpha w^{-1}$ is a reflection about the root $w(\alpha) = \beta$. This gives us that $s_\beta = ws_\alpha w^{-1}$ and we are done. ∎

The last proof shows that, in $GL(V)$, any two reflections are conjugate.

1.4 Exchange and Deletion conditions

In this section, we introduce the Exchange and Deletion conditions, two important results for Coxeter systems, as well as some of their corollaries. In this section, when we write $s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}$, we will mean the expression $s_{i_1} \cdots s_{i_r}$ with s_{i_k} removed.

Theorem 1.4.1 (Exchange Condition)**.** *Let $w = s_{i_1} \cdots s_{i_r}$ with each $s_{i_k} \in S$. Suppose that for some $s \in S$, $\lambda(ws) < \lambda(w)$. Then $ws = s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}$ for some index k.*

If the expression for w is reduced, then the index k is unique and the expression $s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}$ is reduced.

Proof: Since $\lambda(ws) < \lambda(w)$, we have that $w(\alpha_s) < 0$ by Theorem 1.3.3. Since $\alpha_s > 0$, there exists an index k such that $s_{i_{k+1}} \cdots s_{i_r}(\alpha_s) > 0$ and $s_{i_k} \cdots s_{i_r}(\alpha_s) < 0$. From Proposition 1.3.7, we have $s_{i_{k+1}} \cdots s_{i_r}(\alpha_s) = \alpha_{s_{i_k}}$, since it is the root sent to negative by s_{i_k}.

From Lemma 1.3.9 and recalling that $s^2 = 1$, we can conclude that

$$s_{i_{k+1}} \cdots s_{i_r} s s_{i_r} \cdots s_{i_{k+1}} = s_{i_k}$$

and thus that $s = s_{i_r} \cdots s_{i_{k+1}} s_{i_k} s_{i_{k+1}} \cdots s_{i_r}$. It follows that

$$ws = s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}.$$

We now suppose that $w = s_{i_1} \cdots s_{i_r}$ is a reduced expression and that there are two indices j and k such that

$$ws = s_{i_1} \dots \widehat{s_{i_j}} \dots s_{i_k} \dots s_{i_r} = s_{i_1} \dots s_{i_j} \dots \widehat{s_{i_k}} \dots s_{i_r}.$$

By left and right cancelation, we obtain

$$s_{i_{j+1}} \dots s_{i_k} = s_{i_j} \dots s_{i_{k-1}}$$

or, if we multiply by s_{i_j} on the left,

$$s_{i_j} \dots s_{i_k} = s_{i_{j+1}} \dots s_{i_{k-1}}.$$

This implies that $w = s_{i_1} \cdots s_{i_r} = s_{i_1} \dots \widehat{s_{i_j}} \dots \widehat{s_{i_k}} \dots s_{i_r}$ and this contradicts the fact that $s_{i_1} \cdots s_{i_r}$ is reduced.

Therefore, whenever $w = s_{i_1} \cdots s_{i_r}$ is reduced, the index k is unique when $ws = s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}$. Furthermore, since $\lambda(ws) = \lambda(w) - 1 = r - 1$ by Proposition 1.1.4, we have that $s_{i_1} \dots \widehat{s_{i_k}} \dots s_{i_r}$ is a reduced expression for ws. ∎

Although the statement of the Exchange condition is given for ws with $w \in W$ and $s \in S$, it is also true when $\lambda(sw) < \lambda(w)$.

Corollary 1.4.2 (Exchange Condition)**.** *Let $w = s_{i_1} \cdots s_{i_r}$ with each $s_{i_k} \in S$. Suppose that for some $s \in S$, $\lambda(sw) < \lambda(w)$. Then $sw = s_{i_1} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$ for some index k. If the expression for w is reduced, then the index k is unique and the expression $s_{i_1} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$ is reduced.*

Proof: Let $w = s_{i_1} \cdots s_{i_r}$ such that $\lambda(sw) < \lambda(w)$. We can apply the exchange condition to $(sw)^{-1} = w^{-1}s = s_{i_r} \cdots s_{i_1} s$ and obtain $w^{-1}s = s_{i_r} \ldots \widehat{s_{i_k}} \ldots s_{i_1}$. Taking the inverse again, we get that $sw = s_{i_1} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$ and we obtain the Exchange condition for sw. ∎

The Exchange condition has many useful consequences that will give us insight into the reduced expressions in (W, S). Indeed, we can obtain insights on which simple reflection $s \in S$ can be found at the beginning of a reduced expression for $w \in W$ or how to find a reduced expression for any given expression $w = s_{i_1} \cdots s_{i_k}$.

Corollary 1.4.3. *Let $w = s_{i_1} \cdots s_{i_r}$ and suppose that for some $s \in S$, $\lambda(sw) < \lambda(w)$. If $s \neq s_{i_1}$, then $sw = s_{i_1} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$ and $k \neq 1$.*

Proof: We will prove this by contradiction. Suppose that $\lambda(sw) < \lambda(w)$ and that the Exchange condition gives

$$sw = \widehat{s_{i_1}} s_{i_2} \cdots s_{i_r} = s_{i_2} \cdots s_{i_r}.$$

We then have that

$$w = ssw = ss_{i_2} \cdots s_{i_r} = s_{i_1} \cdots s_{i_r}.$$

It follows by right cancellation that $s = s_{i_1}$, which is a contradiction. ∎

Definition 1.4.4. Given a Coxeter system (W, S), for any pair i, j such that $m(i, j) < \infty$, define

$$ {}_{m_{i,j}}(s_i, s_j) = \begin{cases} (s_i s_j)^{\frac{m(i,j)}{2}} & \text{if } m(i,j) \text{ is even,} \\ (s_i s_j)^{\frac{m(i,j)-1}{2}} s_i & \text{if } m(i,j) \text{ is odd.} \end{cases} $$

This is the alternating sequence of s_i and s_j, starting with s_i, of length $m_{i,j}$.

From the relations of a Coxeter system, we have that ${}_{m_{i,j}}(s_i, s_j) = {}_{m_{i,j}}(s_j, s_i)$ for all pairs i, j and that ${}_{m_{i,i}}(s_i, s_i) = s_i$.

Corollary 1.4.5. *Let $w = s_i A = s_j B$, where $s_i A$ and $s_j B$ are reduced expressions. Then there exists a $C \in W$ such that $w = {}_{m_{i,j}}(s_i, s_j)C$, and ${}_{m_{i,j}}(s_i, s_j)C$ is a reduced expression.*

Proof: If $s_i = s_j$, then we are done, since ${}_{m_{i,i}}(s_i, s_i) = s_i$ and we can take $C = A = B$. We thus suppose that $s_i \neq s_j$ and that $\lambda(w) = r$. We first consider

$$ s_i w = A = s_i s_j B. $$

Since $\lambda(s_i w) = \lambda(A) < \lambda(w)$, Corollary 1.4.3 gives us that

$$ s_i s_j B = s_j B_1, $$

where B_1 is a reduced expression obtained by removing a generator from B with the Exchange condition. We thus have that $s_i w = A = s_j B_1$ and multiplying everything by s_i on the left yields

$$ w = s_i A = s_j B = s_i s_j B_1. $$

If $m(i, j) = 2$, we have that $w = s_i s_j B_1 = {}_{m_{i,j}}(s_i, s_j)B_1$ and we are done.

If $m(i, j) > 2$, then we can write

$$ s_j w = B = s_j s_i s_j B_1. $$

Since $\lambda(s_j w) = \lambda(B) < \lambda(w)$ and $s_j \neq s_i$, we can use Corollary 1.4.3 again. This yields either

$$ s_j s_i s_j B_1 = s_i B_1 \text{ or } s_j s_i s_j B_1 = s_i s_j B_2, $$

where B_2 is an expression obtained by removing a generator from B_1 with the Exchange condition. The first possibility gives us, by right cancellation of B_1, that $s_j s_i s_j = s_i$ and thus that $m(i,j) = 2$, but this is a contradiction. Thus, $s_j w = B = s_i s_j B_2$ and we have that

$$w = s_i A = s_j B = s_j s_i s_j B_2.$$

At this point, if $m(i,j) = 3$, we are done since $w = {}_{m_{i,j}}(s_i, s_j)B_2$.

If $m(i,j) > 3$, we proceed in a similar way. We repeat the argument by applying the Exchange condition alternatively to $s_i w$ and $s_j w$ until we obtain ${}_{m_{i,j}}(s_i, s_j)B_k$. We note that the expressions ${}_{m_{i,j}}(s_i, s_j)B_k$ are reduced, since at every step we find a reduced expression for $s_i w = A$ and $s_j w = B$ through the Exchange condition. ∎

Corollary 1.4.6. *Let $w = s_i A$ where $s_i A$ is a reduced expression. For any j such that $m(i,j) > \lambda(w)$, there are no reduced expressions of the form $s_j B$ for w.*

Proof: We proceed by contradiction. Suppose that there exists a reduced expression for w of the form $s_j B$. Then, by Corollary 1.4.5, there exists a reduced expression of the form ${}_{m_{i,j}}(s_i, s_j)C$ for w. But we observe that

$$\lambda({}_{m_{i,j}}(s_i, s_j)C) \geq \lambda({}_{m_{i,j}}(s_i, s_j)) = m(i,j) > \lambda(w).$$

This is a contradiction. ∎

Another useful consequence of the Exchange condition is the following proposition.

Proposition 1.4.7. *Let $w \in W$ and $s \in S$ be such that $\lambda(sw) < \lambda(w)$. Then there is a reduced expression for w starting with s.*

Proof: Let $s_{i_1} \cdots s_{i_r}$ be a reduced expression for w. By the Exchange condition, $sw = s_{i_1} \cdots \widehat{s_{i_k}} \cdots s_{i_r}$, for some index k. We then write $s(sw) = w = s(s_{i_1} \cdots \widehat{s_{i_k}} \cdots s_{i_r})$. It follows that $s(s_{i_1} \cdots \widehat{s_{i_k}} \cdots s_{i_r})$ is a reduced expression for w starting with s. ∎

Finally, we state and prove the Deletion condition, a direct corollary of the Exchange condition.

Corollary 1.4.8 (Deletion Condition)**.** *Let $w = s_{i_1} \cdots s_{i_r} \in W$ with $\lambda(w) < r$. Then $w = s_{i_1} \ldots \widehat{s_{i_j}} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$ for some $j < k$.*

Proof: Since $\lambda(w) < r$, it follows that there exists some k such that

$$\lambda(s_{i_1} \cdots s_{i_{k-1}} s_{i_k}) < \lambda(s_{i_1} \cdots s_{i_{k-1}}).$$

Using the Exchange condition, we obtain that

$$s_{i_1} \cdots s_{i_k} = s_{i_1} \ldots \widehat{s_{i_j}} \ldots s_{i_{k-1}}$$

for some index j. It follows that $w = s_{i_1} \ldots \widehat{s_{i_j}} \ldots \widehat{s_{i_k}} \ldots s_{i_r}$. ∎

The Deletion condition gives us an inductive way to find a reduced expression for some element $w \in W$. Indeed, if an expression $s_{i_1} \cdots s_{i_k}$ for w is not reduced, that is if $k > n(w)$, then we can shorten the expression by removing two of its elements with the Deletion condition. We can then continue this process until the expression for w becomes reduced.

1.5 Finite reflection groups

In this section we look at the case when the group W in our Coxeter system (W, S) is finite. It can be shown, see [Hum90, Section 6.4], that this happens if and only if

the bilinear form B is positive definite, that is, exactly when we are working in the standard Euclidean space. We will be proving results specific to the finite groups W that will be useful for our study of the Artin groups of finite type.

Definition 1.5.1. Let (W, S) be a Coxeter system where the group W is finite. We say that W is a *finite reflection group.*

Since W is finite, we have that Φ is finite, thus Φ^+ needs to be finite, and there exists an element w_0 that has maximal length.

Proposition 1.5.2. *Let (W, S) be a finite reflection group. There exists a unique $w_0 \in W$ such that $\lambda(w_0) \geq \lambda(w)$ for all $w \in W$. Furthermore, $w_0(\Phi^+) = \Phi^-$.*

Proof: For the existence we use a constructive approach. We first pick some $w \in W$ and compute $\lambda(ws)$ for some $s \in S$. If $\lambda(ws) < \lambda(w)$ for all $s \in S$, then w is an element of longest length. If $\lambda(ws) > \lambda(w)$ for some $s \in S$, we do the process again for $w' = ws$ until we obtain an element of longest length.

This longest element w_0 is such that $w_0(\Phi^+) = \Phi^-$. Indeed, suppose $n(w_0) = |\Pi(w)| = |\Phi^+ \cap w_0^{-1}(\Phi^-)| < |\Phi^+|$, then there exists a positive root $\alpha \in \Phi^+$ such that $\alpha \notin \Pi(w)$. This means that $n(w_0 s_\alpha) > n(w_0)$ and $\lambda(w_0 s_\alpha) > \lambda(w_0)$ by Proposition 1.3.8, but this contradicts the maximality of $\lambda(w_0)$. Hence, $w_0(\Phi^+) = \Phi^-$.

For the unicity, suppose that w, w' are two elements of longest length. We thus have that $w(\Phi^+) = w'(\Phi^+) = \Phi^-$ and that $w^{-1}(\Phi^-) = w'^{-1}(\Phi^-) = \Phi^+$. We then have that $w'^{-1}w(\Phi^+) = \Phi^+$ and it follows that $n(w'^{-1}w) = 0$. From Proposition 1.3.8, we get that $\lambda(w'^{-1}w) = 1$ and thus that $w'^{-1}w = 1$. We then conclude that $w = w'$. ∎

Two direct consequences of the construction of the longest element w_0 are the following results.

Corollary 1.5.3. *Given any reduced expression $w \in W$, one can find a reduced expression $w' \in W$ such that ww' is a reduced expression for w_0.*

Corollary 1.5.4. *Given a finite reflection group W, we have $w_0^{-1} = w_0$.*

We now look at the action of W on V described in Section 1.2 in more depth with the goal of describing the *fundamental domain* for this group action. We begin by defining the set $A_{\alpha_s} = \{v \in V | B(v, \alpha_s) > 0\}$ for each $s \in S$ as the *open (positive) half-space associated to* s, while the *negative half-space* is the set $-A_{\alpha_s} = \{v \in V | B(v, \alpha_s) < 0\}$. We then consider the set

$$C = \cap_{s \in S} A_{\alpha_s}$$

and its closure

$$D = \cap_{s \in S}(A_{\alpha_s} \cup H_s) = \{v \in V | B(v, \alpha_s) \geq 0 \text{ for all } s \in S\}.$$

We will show that D is a fundamental domain for the action of W on V, that is, every $v \in V$ is in the W-orbit of a unique $u \in D$.

Lemma 1.5.5. *If $s \in S$, then $s(A_{\alpha_s}) = -A_{\alpha_s}$. Moreover, $-A_{\alpha_s} = A_{-\alpha_s}$.*

Proof: Let $u \in A_{\alpha_s}$. We compute $B(su, \alpha_s) = B(u, s(\alpha_s)) = B(u, -\alpha_s) = -B(u, \alpha_s) < 0$. We thus obtain that

$$s(A_{\alpha_s}) = \{v \in V | B(v, \alpha_s) < 0\} = -A_{\alpha_s}$$

and that $-A_{\alpha_s} = A_{-\alpha_s}$. ∎

Proposition 1.5.6. *Let $u \in V$. Then there exists some $v \in D$ such that u is in the W-orbit of v, $W(v)$.*

Proof: We define a partial order on V by $u \leq v$ if and only if $v - u$ is a linear combination of the α_s's where all the coefficients are non-negative. We verify that this is indeed a partial order. The reflexivity is clear, since $u - u = 0$. For the symmetry,

$u \leq v$ and $v \leq u$ implies that all the coefficients of $u - v$ and $v - u$ are non-negative, which forces $u - v = 0$ and it follows that $u = v$. For the transitivity, suppose $u \leq v$ and $v \leq w$. We then write $w - u = (w - v) + (v - u)$ and since all the coefficients of $w - v$ and $v - u$ are non-negative, it follows that $u \leq w$.

We now let $u \in V$ and consider the set $X = \{v \in W(u) | u \leq v\}$, where $W(u)$ is the W-orbit of u. This set is clearly non-empty since $u \in X$, so we can choose $v \in X$ to be a maximal element. Now, for any $s \in S$, $s(v)$ is in the orbit of u and $s(v) = v - B(v, \alpha_s)\alpha_s$. Because we chose v to be maximal, we have that $B(v, \alpha_s) \geq 0$. Indeed, if $B(v, \alpha_s) < 0$, then $s(v) - v = -B(v, \alpha_s)\alpha_s$ is a positive linear combination of the α_s and we would get that $v < s(v)$, which is a contradiction. It follows that $B(v, \alpha_s) \geq 0$ for all $s \in S$ and that $v \in D$.

Finally, since $v \in W(u)$, we have that $u \in W(v)$ as needed. ∎

Proposition 1.5.7. *If $u, v \in D$ are such that $w(u) = v$ for some $w \in W$, then $u = v$ and any reduced expression for w is a product of simple reflections $s \in S$ such that $s(u) = u$.*

Proof: We proceed by induction on $\lambda(w)$. If $\lambda(w) = 0$, then $w = 1$ and we are done. If $\lambda(w) > 0$ and w, then $w(\alpha_s) < 0$ for some $s \in S$ and $\lambda(ws) = \lambda(w) - 1$ for such an s. Since $w(\alpha_s) = -\alpha_s$, we have that $B(v, w(\alpha_s)) = -B(v, \alpha_s) \leq 0$. We then have

$$0 \leq B(u, \alpha_s) = B(w^{-1}(v), \alpha_s) = B(v, w(\alpha_s)) \leq 0.$$

Thus $B(u, \alpha_s) = 0$ and $s(u) = u$. It follows that $ws(u) = v$ and we obtain by induction that $u = v$ and that ws is a product of simple reflections. As this applies to any choice of s, we obtain that any reduced expression for w is a product of simple reflections $s \in S$ such that $s(u) = u$, as needed. ∎

From these two propositions we obtain the next theorem.

Theorem 1.5.8. *The set D is a fundamental domain for the action of W on V. Furthermore, if $v \in C = int(D)$, then $\{w \in W | w(v) = v\} = \{1\}$.*

Proof: Combining Propositions 1.5.6 and 1.5.7, we obtain that any $u \in V$ is in the W-orbit of exactly one $v \in D$, as needed.

If $v \in C$, then $B(v, \alpha_s) > 0$ for all $s \in S$. It follows that $s(v) \neq v$ for any $s \in S$. As such, any reduced expression for a $w \in W$ fixing v is such that $\lambda(w) = 1$. Thus $\{w \in W | w(v) = v\} = \{1\}$. ∎

Corollary 1.5.9. *If $v \in C$ and $w, w' \in W$ are such that $w(v) = w'(v)$, then $w = w'$.*

Proof: We observe with our hypothesis that $w'^{-1}w(v) = v$. Since $v \in C$, Theorem 1.5.8 then gives us that $w'^{-1}w = 1$ and thus that $w = w'$. ∎

Corollary 1.5.10. *Let $v \in C$ and $\alpha \in \Phi$. We have that $B(v, \alpha) > 0$ if and only if $\alpha \in Phi^+$.*

Proof: Write $\alpha = \sum_{s_i \in S} c_i \alpha_{s_i}$. Then $B(v, \alpha) = \sum_{s_i \in S} c_i B(v, \alpha_{s_i})$. The proof follows from the definition of C and of a positive root. ∎

Now that we have a fundamental domain for W over V, it is easier to describe the elements of W. Indeed, there is a bijection between W and $W(v)$ for any $v \in C$ that associates the vector $w(v)$ to the element w. It also gives a solution to the word problem in W, because $w = 1$ if and only if $w(v) = v$.

1.6 Some finite reflection groups

We now introduce three infinite families of finite reflection groups: the groups of type A_n, B_n and D_n. We give their Coxeter graphs, their realization in Euclidean space, the set of simple roots we use when working with them and a reduced expression for their longest element. We omit all the proofs for this section because these are results that are readily available in the literature (see [Hum90, Section 2.10]) and because the classification of finite reflection groups is well known (see [Hum90, Section 2.7]).

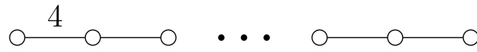
Example 1.6.1. A finite reflection group W of *type* A_n is given by the following Coxeter graph.

To each node, from left to right, we associate the elements $s_1, \dots, s_n$. We take $V \subset \mathbb{R}^{n+1}$ and the simple roots are the vectors $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \dots, n$. The group W is the permutation group S_{n+1}. Given a vector $x = (x_1, \dots, x_{n+1})$, the action of s_i on V is given by

$$s_i : (x_1, \dots, x_i, x_{i+1}, \dots, x_{n+1}) \mapsto (x_1, \dots, x_{i+1}, x_i, \dots, x_{n+1}).$$

The longest element is $w_0 = s_1 \dots s_n s_1 \dots s_{n-1} \dots s_1 s_2 s_1$.

Example 1.6.2. A finite reflection group W of *type* B_n is given by the following Coxeter graph.

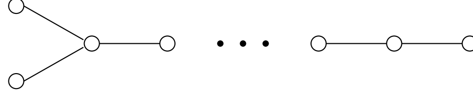


To each node, from left to right, we associate the elements $s_0, s_1 \dots, s_{n-1}$. We take $V = \mathbb{R}^n$ and the simple roots are the vectors $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \dots, n-1$ and $\alpha_{s_0} = \epsilon_1$. The group W is the semi-direct product of S_n and $\mathbb{Z}_2^n$, where the action of s_i on V is given by

$$s_0 : (x_1, x_2, \dots, x_n) \mapsto (-x_1, x_2, \dots, x_n)$$
$$s_i : (x_1, \dots, x_i, x_{i+1}, \dots, x_n) \mapsto (x_1, \dots, x_{i+1}, x_i, \dots, x_n) \quad \text{if } i \neq 0.$$

The longest element is $w_0 = (s_0 \dots s_{n-1})^n$.

Example 1.6.3. A finite reflection group W of *type* D_n is given by the following Coxeter graph.



To each node, from left to right, we associate the elements $s_0, s_1 \dots, s_{n-1}$ with s_0 and s_1 being the two nodes on the fork. We take $V = \mathbb{R}^n$ and the simple roots are the vectors $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \dots, n-1$ and $\alpha_{s_0} = \epsilon_1 + \epsilon_2$. The group W is the semi-direct product of S_n and $\mathbb{Z}_2^{n-1}$, where the action of s_i on V is given by

$$s_0 : (x_1, x_2, x_3, \dots, x_n) \mapsto (-x_2, -x_1, x_3, \dots, x_n)$$
$$s_i : (x_1, \dots, x_i, x_{i+1}, \dots, x_n) \mapsto (x_1, \dots, x_{i+1}, x_i, \dots, x_n) \quad \text{if } i \neq 0.$$

The longest element is $w_0 = (s_0 \dots s_{n-1})^{n-1}$.

Our choice of labels starting at 0 instead of 1 for generators of B_n and D_n is for consistency. In this labelling, the generator s_i with $i \neq 0$ acts in the same way on V for all three groups.

Remark 1.6.4. To find the fundamental domain D of W when the simple roots α_s are known is fairly simple. Pick some root α_s for $s \in S$ and write α_s as a linear combination of the canonical basis $\epsilon_1, \dots, \epsilon_n$: $\alpha_s = \sum_{i=1}^n c_i \epsilon_i$. An arbitrary vector $v = \sum_{i=1}^n a_i \epsilon_i$ is in the positive halfspace A_{α_s} if $B(v, \alpha_s) > 0$. We can then compute

$$\begin{aligned} B(v, \alpha_s) &= \textstyle\sum_{i=1}^n \sum_{j=1}^n a_i c_j B(\epsilon_i, \epsilon_j) \\ &= \textstyle\sum_{i=1}^n a_i c_i, \end{aligned}$$

since we have that $B(\epsilon_i, \epsilon_j) = 0$ due to the space being Euclidean. Then from requiring $B(v, \alpha_s) > 0$, we get the inequality

$$\sum_{i=1}^n a_i c_i > 0.$$

By repeating this process for all simple roots, we obtain a system of inequalities that we can solve for the c_i's to obtain a characterization of the interior of the fundamental domain C.

Example 1.6.5. In this example, we compute the fundamental domain D for our representation of A_n. We have that $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$ for $i = 1, \ldots, n$. Let $x = \sum_{i=1}^{n+1} x_i \epsilon_i$. Then, $x \in A_{\alpha_{s_i}}$ if $B(x, \alpha_s) = x_{i+1} - x_i > 0$, that is, whenever $x_i < x_{i+1}$.

It follows that $x_i \in C = \cap_{s \in S} A_{\alpha_s}$ whenever $x_1 < x_2 < \ldots < x_n$. As such, with our choice of simple roots, the set $C' = \{x = (x_1, \ldots, x_n) \in V | 0 < x_1 < x_2 < \ldots < x_n\}$ is contained in the interior C of the fundamental domain.

For all three families of finite reflection groups we presented, we chose the simple roots in such a way that $C' = \{x = (x_1, \ldots, x_n) \in V | 0 < x_1 < x_2 < \ldots < x_n\} \subset C$. In particular, the vector vector $(1, 2, \ldots, n) \in C'$ is in C.

1.7 Lattice structure

In this section, we introduce a partial order on (W, S) and we will show that, under this order, all posets have a minimum element. In this section, unless otherwise stated, let (W, S) be some fixed arbitrary, possibly infinite, Coxeter system. We begin by giving a general definition.

Definition 1.7.1. Let X be a poset with partial order $\leq$ and let $u, v \in X$. The *meet* of u and v, $u \wedge v$, is the greatest lower bound of u and v, which is the maximum element of the set

$$\{x | x \leq u \text{ and } x \leq v\}.$$

The *join* of A, $u \vee v$, is the least upper bound of u and v, which is the minimum element of the set $\{x | u \leq x \text{ and } v \leq x\}$.

The meet or join might not exist for all pairs u and v, but if they do exist for a given pair, then they are unique for that pair. If the meet (resp. join) do exist for all pairs u and v, we say that X is a *meet-semilattice* (resp. *join-semilattice*). If both the meet and the join exist for any pair u, v, we say that X is a *lattice*.

By definition of the meet, we have that if $z = u \wedge v$ and x is such that $x \leq u$ and $x \leq v$, then $x \leq z$.

Definition 1.7.2. For any $a, b \in W$, we say that a is a *left divisor* of b and write $a \leq_L b$ if $b = au$ for some reduced expressions a, u and au. Similarly, we say that a is a *right divisor* of b and write $a \leq_R b$ if $b = ua$, with reduced expressions a, u and ua.

We call the order $\leq_L$ (resp. $\leq_R$) the *left* (resp. *right*) *weak order* on (W, S).

Proposition 1.7.3. *The left and right weak orders are partial orders on W.*

Proof: We prove that $\leq_L$ is a partial order. The proof for $\leq_R$ is similar. For any $a \in W$, we have $a = a \cdot 1$ and thus $a \leq_L a$. The reflexivity of $\leq_L$ follows.

For the transitivity, suppose that $a \leq_L b$ and $b \leq_L c$, for some $a, b, c \in W$. Then, $b = ad_1$ and $c = bd_2$ for some reduced expressions $d_1, d_2 \in W$. It follows that $c = ad_1d_2$ is a reduced expression and that $a \leq_L c$.

For the antisymmetry, suppose that $a \leq_L b$ and $b \leq_L a$ for some $a, b \in W$. Then, we have $a = bc$ and $b = ad$, where a, b, c, bc, ad are reduced expressions. We then have

$$\lambda(a) = \lambda(b) + \lambda(c)$$
$$\lambda(b) = \lambda(a) + \lambda(d)$$

and it follows that

$$\lambda(c) = \lambda(a) - \lambda(b) \geq 0$$
$$\lambda(d) = \lambda(b) - \lambda(a) \geq 0.$$

It follows that

$$0 \geq \lambda(c) = -\lambda(d) \leq 0$$

and that $c = d = 1$. We conclude that $a = b$. ∎

We remark that if $a \leq_L b$, then $a^{-1} \leq_R b^{-1}$. We will soon show that (W, S) is a meet-semilattice under these orders.

Definition 1.7.4. Let $w \in W$. The *starting set* $S(w)$ of w is the set

$$S(w) = \{s \in S | s \leq_L w\}.$$

Similarly, the *finishing set* of w is the set

$$F(w) = \{s \in S | s \leq_R w\}.$$

Proposition 1.7.5. *The starting set of $w \in W$ is $S(w) = \{s \in S | \lambda(sw) \leq \lambda(w)\}$.*

Proof: Let $w \in W$ be a reduced expression. If $s \in S(w)$, then $w = su$ for some reduced expression u. It follows that $\lambda(sw) = \lambda(u) < \lambda(w)$.

Conversely, if $\lambda(sw) \leq \lambda(w)$, then by Proposition 1.4.7 we have that $s \leq_L w$. ∎

Proposition 1.7.6. *Let $u, w \in W$ be reduced expressions and suppose $s \in S(u) \cap S(w)$. Then $u \leq_L w$ if and only if $su \leq_L sw$.*

Proof: With $s \in S(u) \cap S(w)$, we can write $w = sw'$ and $u = su'$ for some reduced w' and u'. We now suppose $u \leq_L w$. Then $w = ua$ for some reduced expression ua. We then have $sw' = su'a$ and by left cancelation, we have $w' = u'a$, thus $u' \leq_L w'$. But we have that $w' = sw$ and $u' = su$. Hence, $su \leq_L sw$. The proof of the converse is similar. ∎

Theorem 1.7.7. *The Coxeter system (W, S) with $\leq_L$ is a meet-semilattice.*

Proof: We show that for any $x, y \in W$, the meet $x \wedge y$ exists by using induction on $\lambda(x)$. If $\lambda(x) = 0$, then we have that $x \wedge y = 1_W$.

For any $x, y \in W$, we have that if $S(x) \cap S(y) = \emptyset$, then $x \wedge y = 1_W$. We will thus assume that $\lambda(x) > 0$ and that $E = \{w | w \leq_L x \text{ and } w \leq_L y\} \neq \{1\}$. We also note that $E \neq \{1\}$ if and only if $S(x) \cap S(y) \neq \emptyset$.

Pick $z \in E$ with maximal length. We claim that $S(z) = S(x) \cap S(y)$. Indeed, suppose that $s \in S(x) \cap S(y)$ and $s \notin S(z)$. We write the reduced expressions $z = s_{i_1} \cdots s_{i_k}$, $x = zx' = s_{i_1} \cdots s_{i_k} s'_{i_1} \cdots s'_{i_p}$ and $y = zy' = s_{i_1} \cdots s_{i_k} s'_{j_1} \cdots s'_{j_q}$. Then, since $s \notin S(z)$, $\lambda(sz) > \lambda(z)$ and $\lambda(sx) < \lambda(x)$, using the Exchange condition yields

$$\begin{aligned} sx &= s_{i_1} \cdots s_{i_k} s'_{i_1} \ldots \widehat{s'_{i_k}} \ldots s'_{i_p} \\ x &= ss_{i_1} \cdots s_{i_k} s'_{i_1} \ldots \widehat{s'_{i_k}} \ldots s'_{i_p} \\ &= szs'_{i_1} \ldots \widehat{s'_{i_k}} \ldots s'_{i_p}. \end{aligned}$$

Similarly for y, we obtain

$$y = szs'_{j_1} \ldots \widehat{s'_{j_l}} \ldots s'_{j_q}.$$

Thus, $sz \leq x$ and $sz \leq y$, with $\lambda(sz) \geq \lambda(z)$. But this contradicts the maximality of the length of z. This proves that $S(x) \cap S(y) \subset S(z)$ and the reverse inclusion follows from z being a left divisor of both x and y.

Next we prove the unicity of z. Let $w \in E \backslash \{1\}$. Then $S(w) \subset S(x) \cap S(y) = S(z)$. We let $s \in S(w)$. Since $\lambda(sx) < \lambda(x)$, the meet of sx and sy exists by the induction hypothesis. Let $z' = sx \wedge sy$, then $z' \leq_L sx$ and $z' \leq_L sy$. By using Proposition 1.7.6, we have $sz' \leq_L x$ and $sz' \leq_L y$. Thus, $sz' \in E$ and it follows that $\lambda(sz') \leq \lambda(z)$, since $\lambda(z)$ is maximal. Proposition 1.7.6 also yields

$$\begin{aligned} sw \leq_L sx \quad & sw \leq_L sy \\ sz \leq_L sx \quad & sz \leq_L sy. \end{aligned}$$

Because $z' = sx \wedge sy$, we have that $sw \leq_L z'$ and $sz \leq_L z'$. Thus $\lambda(sz) \leq \lambda(z')$, and we can write

$$\lambda(z') \geq \lambda(sz) = \lambda(z) - 1 \geq \lambda(sz') - 1 = (\lambda(z') + 1) - 1 = \lambda(z').$$

Thus, $\lambda(z') = \lambda(sz)$ and it follows that $sz = z'$.

Since $sw \leq_L z'$, we obtain that $sw \leq_L sz$ and thus that $w \leq_L z$. Thus, we showed that for any $w \in E$, $w \leq z$ and it follows that $z = x \wedge y$. ∎

Chapter 2

Artin Groups

In this chapter, we introduce two types of Artin groups, the Artin groups of finite type and of large type, and some of their properties. In Section 2.1, we introduce various definitions used throughout the chapter and we define the set of reduced elements in Section 2.2. We then focus on the Artin groups of finite type, in Section 2.3, while using the braid group as the main example of such Artin groups. Finally, in Section 2.4 we prove that equivalent reduced elements can be obtained from one another without using any relations that modify the lengths of the elements. This last section is necessary to have the set reduced elements be well defined.

2.1 Definitions

Recall that a Coxeter system is an ordered pair (W, S), where W is a group with set of generators $S = \{s_1, \ldots, s_n\}$, such that the relations between pairs of generators are given by $(s_i s_j)^{m(i,j)} = 1$ (see 1.1.1). Note that we can write the relations of a Coxeter group in a slightly different way. Indeed for each pair of i and j with $m(i, j) \neq \infty$,

we will rewrite the relations as

$$\begin{aligned} &(s_is_j)^{\frac{m(i,j)}{2}} = (s_js_i)^{\frac{m(i,j)}{2}} && \text{if } m(i,j) \text{ is even,} \\ &(s_is_j)^{\frac{m(i,j)-1}{2}}s_i = (s_js_i)^{\frac{m(i,j)-1}{2}}s_j && \text{if } m(i,j) \text{ is odd.} \end{aligned} \tag{2.1.1}$$

Definition 2.1.1. Given a Coxeter system (W,S) on n generators with corresponding $m(i,j)$, the *Artin group*, $\mathcal{A}$, *associated to* W is the group generated by $S' = \{\sigma_1,\dots,\sigma_n\}$ with the following relations: for each pair i,j with $i \neq j$ and $m(i,j) \neq \infty$,

$$\begin{aligned} &(\sigma_i\sigma_j)^{\frac{m(i,j)}{2}} = (\sigma_j\sigma_i)^{\frac{m(i,j)}{2}} && \text{if } m(i,j) \text{ is even,} \\ &(\sigma_i\sigma_j)^{\frac{m(i,j)-1}{2}}\sigma_i = (\sigma_j\sigma_i)^{\frac{m(i,j)-1}{2}}\sigma_j && \text{if } m(i,j) \text{ is odd.} \end{aligned} \tag{2.1.2}$$

If the associated Coxeter group is a finite reflection group, we say that the Artin group is of *finite* or *spherical type*, otherwise the Artin group is of *infinite type*. If $m(i,j) \geq 3$ for all $i \neq j$, we say the Artin group is of *large type*, while if $m(i,j) \geq 4$ for all $i \neq j$, we say it is of *extra large type*. The generators $\sigma_1,\dots,\sigma_n$ are called the *Artin generators* for $\mathcal{A}$ and the identity element will be denoted by 1. We put $A = S' \cup S'^{-1}$.

Definition 2.1.2. A *word* over A is an expression $w = \sigma_{i_1}^{\epsilon_1}\sigma_{i_2}^{\epsilon_2}\dots\sigma_{i_k}^{\epsilon_k}$, where $\sigma_{i_j} \in \{\sigma_1,\dots,\sigma_n\}$ and $\epsilon_i = \pm 1$.

The index k is called the *word length* and is denoted by $\ell(w)$.

We denote the *empty word* by 1 and we put $\ell(1) = 0$. We will denote the set of all words over A by A^*. Together with concatenation, A^* is the free group on A.

Remark 2.1.3. We note that the word length is a notion of length on the expressions, not on the elements of $\mathcal{A}$. As such, $\ell(\sigma_1\sigma_1^{-1}\sigma_1) = 3$ and $\ell(\sigma_1) = 1$.

There is an obvious surjective group homomorphism $\xi : A^* \to \mathcal{A}$ given by taking the quotient of A^* by the relations in (2.1.2) and the relations $\sigma_i\sigma_i^{-1} = 1$. This map sends a word w to an element $\overline{w} = \xi(w)$ called the *element represented by the word* w.

We denote by $\mathcal{A}^+$ the submonoid of $\mathcal{A}$ generated by S'.

Definition 2.1.4. An element $b \in \mathcal{A}$ is said to be *positive* if it is contained in $\mathcal{A}^+$.

We say that $b \in \mathcal{A}$ is *negative* if it is contained in $\mathcal{A}^- = (\mathcal{A}^+)^{-1}$.

Definition 2.1.5. For any $a, b \in \mathcal{A}$, we say that a is a *left* (respectively *right*) *divisor* of b if $b = ap$ (resp. $b = pa$) for some $p \in \mathcal{A}^+$.

Remark 2.1.6. Due to the lack of inverses in $\mathcal{A}^+$ and the fact that all the relations in (2.1.2) are balanced, if two words $a, b \in \mathcal{A}^+$ are such that $a = b$ as elements, then $\ell(a) = \ell(b)$.

As such, for any element $a \in \mathcal{A}^+$, $\ell(a)$ is the number of generators used in any expression representing a in $\mathcal{A}^+$.

2.2 Reduced elements

As can be seen from the way the relations of $\mathcal{A}$ are defined, adding the relations $\sigma_i^2 = 1$ for all i gives us the Coxeter group W associated to $\mathcal{A}$. We thus have a natural surjective group homomorphism $\rho : \mathcal{A} \to W$ given by $\rho(\sigma_i) = s_i$.

We would like to find a nice subset X of $\mathcal{A}$ such that the map $\rho : X \to W$ is a bijection. The construction of this set X will be crucial for the Artin groups of finite type, for it is the set that will help us build a unique normal form for these groups. We note that X cannot be a subgroup of $\mathcal{A}$ since $\mathcal{A}$ has fewer relations than W. The problem is choosing the right set X.

We first begin by defining a map from W to $\mathcal{A}$ in a natural fashion. Given an element $w \in W$ with $w = s_{i_1} \cdots s_{i_k}$, define the map $\phi : S^* \to A^*$ by

$$\phi(s_{i_1} \cdots s_{i_k}) = \sigma_{i_1} \cdots \sigma_{i_k}.$$

In other words, we have the following commutative diagram:

$$\begin{array}{ccc} W & \xleftarrow[\rho]{} & \mathcal{A} \\ \uparrow & & \uparrow \xi. \\ S^* & \xrightarrow[\phi]{} & A^* \end{array}$$

Note for example that $\phi(s_1^2) = \sigma_1^2$, where $s^2 = 1$ in W, but $\sigma^2 \neq 1$ in $\mathcal{A}$. Thus, ϕ does not factor through W.

Our goal is to build a simple set $X \subset \mathcal{A}$ such that the map

$$W \xrightarrow[\rho^{-1}]{} X.$$

is a bijection. This is done in the following way for each $w \in W$:

- Choose one reduced expression $\omega = s_{i_1} \cdots s_{i_k}$ for w.

- We have $\omega \in S^*$, so we take $\widetilde{w} = \xi \circ \phi(\omega) \in \mathcal{A}$.

Then, clearly $\rho(\widetilde{w}) = w$. We then put the set $X \subset A^*$ as the image through ϕ of these reduced words ω.

Definition 2.2.1. Let $R(W) \subset S^*$ be a set of reduced expressions such that for every $w \in W$, there is exactly one expression in $R(W)$ representing w. Define $\mathcal{A}^{Red}_{R(W)} = \xi \circ \phi(R(W))$, the set of *reduced elements associated to* $R(W)$.

Such a set $\mathcal{A}^{Red}_{R(W)}$ has the property that $\rho : \mathcal{A}^{Red}_{R(W)} \to W$ is a bijection. We emphasize that, a priori, $\mathcal{A}^{Red}_{R(W)}$ depends on the choice of $R(W)$. To show that it is actually independent can be reduced to proving the next theorem, whoose proof will be deferred until Section 2.4 for it is quite involved.

Theorem 2.4.5. *If $s_{i_1} \cdots s_{i_k}$ and $s_{j_1} \cdots s_{j_k}$ are reduced expressions for the same element $w \in W$, then $\sigma_{i_1} \cdots \sigma_{i_k} = \sigma_{j_1} \cdots \sigma_{j_k}$ in $\mathcal{A}^+$.*

Another formulation of this result is that given a reduced expresion ω for $w \in W$, one can obtain every other reduced expression for w using only the relations given in equation (2.1.1). Knowing this, we shall write $\mathcal{A}^{Red} = \mathcal{A}^{Red}_{R(W)}$ and we will denote by a_w the element of $\mathcal{A}^{Red}$ with $\rho(a_w) = w$.

Definition 2.2.2. An element $a_w \in \mathcal{A}^{Red}$ is said to be a *reduced element* of $\mathcal{A}$.

By our construction, the map $\rho : \mathcal{A}^{Red} \to W$ is a bijection. We also obtain, by our definition of ϕ that all the reduced elements of $\mathcal{A}$ are positive, that is, we have that $\mathcal{A}^{Red} \subset \mathcal{A}^+$. We will now proceed to give some properties of these elements as well as characterizing them. We will assume that Theorem 2.4.5 is true and that the set $\mathcal{A}^{Red}$ is well defined for the rest of this section.

Recall that for an element w of W, the length of w, $\lambda(w)$, is the number of elements r in any reduced expression for w. From the definition of the homomorphism ρ, we have that for every element $a \in \mathcal{A}^+$,

$$\lambda(\rho(a)) \leq \ell(a).$$

If a is reduced, then we have the following proposition.

Proposition 2.2.3. *A positive element $a \in \mathcal{A}^+$ is reduced if and only if $\lambda(\rho(a)) = \ell(a)$.*

Proof: If a is reduced, then $\lambda(\rho(a)) = \ell(a)$ follows by our construction of $\mathcal{A}^{Red}$. Conversely, suppose $\lambda(\rho(a)) = \ell(a) = r$. Write $a = \sigma_{i_1} \cdots \sigma_{i_r}$, then $\rho(a) = s_{i_1} \cdots s_{i_r}$, which is a reduced expression for $\rho(a)$ since $\lambda(\rho(a)) = r$. Thus a is reduced because it is the image of a reduced expression through ϕ. ∎

Proposition 2.2.4. *Any left or right positive divisor of a reduced element is reduced.*

Proof: Let $a, b \in \mathcal{A}^+$ be such that $ab \in \mathcal{A}^{Red}$. Then

$$\ell(a) + \ell(b) = \ell(ab) = \lambda(\rho(ab)) = \lambda(\rho(a)\rho(b)) \leq \lambda(\rho(a)) + \lambda(\rho(b)) \leq \ell(a) + \ell(b).$$

Since the equality holds, we obtain that $\lambda(\rho(a)) = \ell(a)$ and $\lambda(\rho(b)) = \ell(b)$. Thus, by Proposition 2.2.3, $a, b \in \mathcal{A}^{Red}$. ∎

From Proposition 2.2.3, we deduce that the identity element and all the generators of an Artin group are reduced elements, since $\ell(\sigma_i) = \lambda(s_i) = 1$.

2.3 Artin groups of finite type

In this section, we consider the properties of the Artin groups of finite type. These groups are those associated to a Coxeter system (W, S) where W is a finite reflection group.

Definition 2.3.1. Given a Coxeter system (W, S) where W is a finite reflection group, denote by $\mathcal{A}(W)$ the Artin group associated to this Coxeter system.

If W is a finite reflection group then there exists an element of $\mathcal{A}(W)^{Red}$ associated to the longest element $w_0 \in W$. This reduced element $a_{w_0} \in \mathcal{A}^{Red}$ plays an important role in solving the word problem in the Artin groups of finite type. As such, we give it a special symbol.

Definition 2.3.2. The *fundamental element*, Δ, of an Artin group of finite type $\mathcal{A}(W)$ is the reduced element such that $\rho(\Delta) = w_0$.

Proposition 2.3.3. *An element $a \in \mathcal{A}^+$ is reduced if and only if it is a left (or right) divisor of Δ.*

Proof: Proposition 2.2.4 gives us that all left (and right) divisors of Δ are reduced.

Now, suppose $a \in \mathcal{A}^{Red}$. Recall from Corollary 1.5.3 that, in a finite Coxeter system (W, S), given any reduced expression for $w \in W$, one can find a reduced expression for some $w' \in W$ such that ww' is a reduced expression for w_0.

Choose a reduced expression w' for the element $\rho(a)^{-1}w_0 \in W$. Then, $\omega_0 = \rho(a)w'$ is a reduced expression for w_0, with $\rho(a)$ and w' reduced expressions. Thus, we have

$$\Delta = \phi(\omega_0) = \phi(\rho(a)w') = \phi(\rho(a))\phi(w') = a\phi(w'),$$

where $\phi(w') \in \mathcal{A}^{Red} \subset \mathcal{A}^+$ by construction. It follows that a is a left divisor of Δ. The proof to show that a is a right divisor is similar. ∎

We will only be working with the infinite classes of finite reflection groups, that is with groups of type A_n, B_n, D_n as well as the dihedral groups $I_2(m)$. We will be using the braid groups, $\mathcal{A}(A_n)$, as the fundamental example for all the results.

Example 2.3.4. The *braid group on n strands* is the Artin group $\mathcal{B}_n = \mathcal{A}(A_{n-1})$. It is the group generated by $\sigma_1, ..., \sigma_{n-1}$ together with the relations

$$\begin{aligned} \sigma_i\sigma_j &= \sigma_j\sigma_i \quad \text{if } |i-j| \geq 2, \text{ and} \\ \sigma_i\sigma_j\sigma_i &= \sigma_j\sigma_i\sigma_j \quad \text{if } |i-j| = 1. \end{aligned} \tag{2.3.1}$$

In the case of the braid group, an element is typically called a *braid* and an arbitrary word in $\mathcal{B}_n$ is called a *braid word.* The relations given in (2.3.1) are called the *braid relations.*

An interesting property of $\mathcal{A}(A_n)$, $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$ is that they have a geometric representation called a *braid diagram.* For the braid group the diagrams are intuitive and we will give their construction. For the Artin groups $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$, the diagrams are more complicated and they involve the notions of orbifolds and cone points (see [All02]).

The notion of a braid diagram is as follows. For $\mathcal{B}_n$, align two columns of n nodes and attach a string to each starting node (on the left column) and attach the other end to a node on the right column, taking note of when a string crosses over or under another string. We label the strings from 1 to n starting from the bottom. Do note that, in the literature, the braid diagrams are sometime drawn vertically from top to bottom.

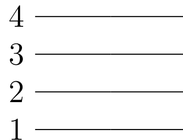


Figure 2.1: The trivial braid in $\mathcal{B}_4$.

The restrictions are that every node can only have one string attached to it and that strings are not allowed to backtrack, that is, any plane parallel to the columns of nodes intersects each string exactly once. The trivial braid is defined as the braid where no two strings cross and where string i goes to node i on the second column (see Figure 2.1).

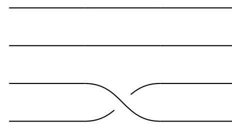


Figure 2.2: The braid σ_1 in $\mathcal{B}_4$.

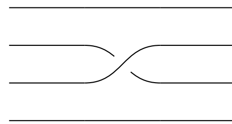


Figure 2.3: The braid σ_2^{-1} in $\mathcal{B}_4$.

We generally omit the labels on the strings when the order is understood. Our braids will always be read from left to right and the process of associating an algebraic braid, that is, an element of $\mathcal{B}_n$, to a geometric braid, that is, a braid diagram, is as follows. We first let the Artin generators σ_i represent strand $i+1$ crossing *over* strand i and let σ_i^{-1} represent strand $i+1$ crossing *under* strand i (see Figure 2.2 and 2.3). Then, reading the algebraic (resp. geometric) braid from the left, we replace every Artin generator (resp. crossing) by its corresponding crossing (resp. Artin generator) (see Figure 2.4).

The close resemblance between the geometric braids and knots motivates the next definition for all Artin groups of finite type.

Definition 2.3.5. An Artin generator σ_i is said to be a *positive crossing*, while σ_i^{-1} is called a *negative crossing*.

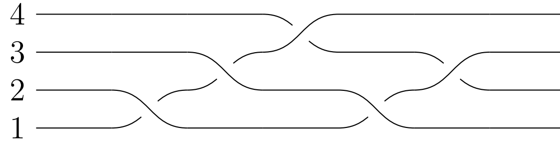


Figure 2.4: The braid diagram for $\sigma_1\sigma_2\sigma_3^{-1}\sigma_1\sigma_2^{-1}$ in B_4.

More formally, a braid in $\mathcal{B}_n$ can be parametrized as a set of n disjoint paths.

Definition 2.3.6. A *braid diagram* in $\mathcal{B}_n$ is a set of n disjoint continuous paths

$$\eta_i : [0,1] \to [0,1] \times \mathbb{R}^2$$

given by $\eta_i(t) = (t, x_i(t), y_i(t))$ and satisfying $\eta_i(0) = (0,0,i)$ and $\eta_i(1) = (1,1,\pi(i))$, where π is some permutation of $\{1,\dots,n\}$. Furthermore, we require the number of crossings to be finite.

The path η_i is said to be the i-th *string* in the braid.

Remark 2.3.7. Let b be a braid and consider the permution π of its braid diagram. Then, we have that $\pi = \rho(b)^{-1}$.

These braid diagrams can then be manipulated using topological tools in the same way as knots. We can then say that two braid diagrams represent the same braid if the two diagrams are *isotopic*, that is, if one can be continuously deformed through a continuous family of homeomorphisms into the other one. These deformations can be represented by sequences of Reidemeister-like moves, for they resemble the Reidermeister moves on knots. These moves express all the relations present in the braid group (see Figures 2.5, 2.6 and 2.7) and it can be shown (see [KT08, Chapter 1]) that two braid diagrams are isotopic if and only if there exists a finite sequence of these three types of moves that transforms the first diagram into the second diagram.

The fact that the braid relations can be translated as Reidemeister moves on the diagrams means that braid words representing the same braid yield isotopic braid

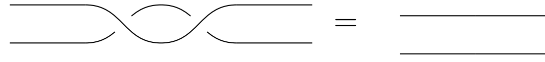


Figure 2.5: The relation $\sigma_1\sigma_1^{-1} = 1$ in $\mathcal{B}_2$.

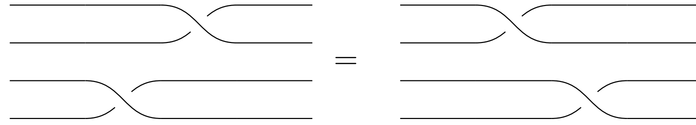


Figure 2.6: The relation $\sigma_1\sigma_3 = \sigma_3\sigma_1$ in $\mathcal{B}_4$.

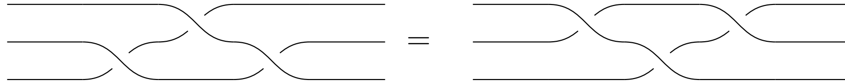


Figure 2.7: The relation $\sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2$ in $\mathcal{B}_3$.

diagrams. The converse is also true, that is, isotopic braid diagrams yield braid words representing the same braid.

We shall not focus on the geometric aspect of the braid groups since it is not the main point of this thesis. It is nonetheless an important property of the braid groups to keep in mind, for some results and concepts are more readily proven or understandable using braid diagrams. These diagrams are a great didactic tool when working with the braid groups.

As previously mentioned, the Artin groups of finite type possess a *fundamental element* Δ.

Lemma 2.3.8. *In $\mathcal{B}_n$, the* fundamental braid *is the braid*

$$\Delta = \sigma_1(\sigma_2\sigma_1)\cdots(\sigma_{n-2}\cdots\sigma_1)(\sigma_{n-1}\cdots\sigma_1).$$

Proof: From Definition 2.3.2, we have that $\Delta = \Phi(\omega_0)$ where ω_0 is a reduced

expression for $w_0 \in A_{n-1}$. Also, from Example 1.6.1, we have that

$$\omega_0 = s_1(s_2 s_1)\cdots(s_{n-2}\cdots s_1)(s_{n-1}\cdots s_1)$$

is an expression for $w_0 \in A_{n-1}$. As such,

$$\Delta = \phi(\omega_0) = \sigma_1(\sigma_2\sigma_1)\cdots(\sigma_{n-2}\cdots\sigma_1)(\sigma_{n-1}\cdots\sigma_1).$$

∎

We give an example of Δ in Figure 2.8.

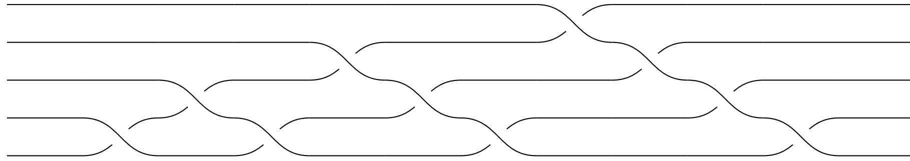


Figure 2.8: The fundamental braid Δ in $\mathcal{B}_5$.

We now present some of the properties of Δ.

Proposition 2.3.9. *Let Δ be the fundamental braid of the Artin group of finite type $\mathcal{A}$. Then there exists an automorphism τ of $\mathcal{A}$ such that for any Artin generator σ_i, we have that $\sigma_i\Delta = \Delta\tau(\sigma_i)$.*

The map $\tau : \mathcal{A} \to \mathcal{A}$ given by $\tau(\sigma_i) = \Delta^{-1}\sigma_i\Delta$ satisfies Proposition 2.3.9. This result can be generalized slightly to the inverses.

Lemma 2.3.10. *Given an Artin group of finite type and any Artin generator $\sigma_i \in \mathcal{A}$, we have*

$$\sigma_i^{\epsilon}\Delta^{\delta} = \Delta^{\delta}\tau(\sigma_i)^{\epsilon}$$

for any $\epsilon, \delta \in \{\pm 1\}$.

Proof: The result follows from right and left multiplication. ∎

Lemma 2.3.11. *Let (W, S) be a finite Coxeter system with longest element w_0 and let $\mathcal{A}$ be the associated Artin group of finite type. For any i, let j be such that $s_i w_0 = w_0 s_j$ in W. Then $\sigma_i \Delta = \Delta \sigma_j$ in $\mathcal{A}$.*

Proof: Since $s_i w_0$ is not reduced in W, choose a reduced expression $w = s_{i_1} \cdots s_{i_k}$ for $s_i w_0$. We have that $\lambda(w) = \lambda(w_0) - 1$, so there exists some $s_j \in S$ such that $w s_j = w_0$. By construction, we have that $w_0 = s_i w = w s_j$ are reduced expressions. Let $a = \Phi(w) = \sigma_{i_1} \cdots \sigma_{i_k}$.

From Theorem 2.4.5, we have that $\Delta = \sigma_i a = a \sigma_j$. We can then write

$$\sigma_i \Delta = \sigma_i (a \sigma_j) = (\sigma_i a) \sigma_j = \Delta \sigma_j,$$

proving the desired equality. ∎

A direct corollary of Lemma 2.3.11 is the following.

Corollary 2.3.12. *The map $\tau : \mathcal{A}^+ \to \mathcal{A}^+$ given by $\tau(\sigma_i) = \Delta^{-1} \sigma_i \Delta$ is an automorphism of $\mathcal{A}^+$.*

We will now determine τ explicitly for each of the four families of Artin groups of finite type $\mathcal{B}_n$, $\mathcal{A}(B_n)$, $\mathcal{A}(D_n)$ and $\mathcal{A}(I_2(m))$. Lemma 2.3.11 will let us work in the Coxeter systems instead of directly computing τ in the Artin groups. We begin with the braid group, $\mathcal{B}_n$, while using the presentation of A_{n-1} given in Example 1.6.1. We then continue by defining explicitly the Artin groups $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$ before giving their fundamental elements and describing τ for these two groups. These groups shall be defined over the Coxeter systems given in Definitions 1.6.2 and 1.6.3. Recall that in these presentations, the generators s_i, $i \neq 0$, act in the same way as s_i in A_n.

For the Artin group of type A_{n-1}

Proposition 2.3.13. *In the Artin group $\mathcal{B}_n = \mathcal{A}(A_{n-1})$, we have that for every i,*

$$\tau(\sigma_i) = \sigma_{n-i}.$$

Proof: Recall that in A_{n-1}, the longest element is given by $w_0 = s_1 s_2 s_1 \ldots s_{n-1} \ldots s_1$. It acts on $\mathbb{R}^n$ as the map

$$w_0 : (x_1, x_2, \ldots, x_n) \mapsto (x_n, \ldots, x_2, x_1).$$

We apply $w_0^{-1} s_i w_0 = w_0 s_i w_0$ to $(1, 2, \ldots, n) \in C$, where C is the interior of the fundamental domain, as given in Section 1.6.

$$\begin{aligned}
(w_0^{-1} s_i w_0)((1, 2, \ldots, n)) &= w_0(s_i(w_0((1, 2, \ldots, n)))) \\
&= w_0(s_i((n, \ldots, 2, 1))) \\
&= w_0((n, \ldots, n-i+2, n-i, n-i+1, n-i-1, \ldots, 1)) \\
&= (1, \ldots, n-i-1, n-i+1, n-i, n-i+2, \ldots, n) \\
&= s_{n-i}((1, 2, \ldots, n)).
\end{aligned}$$

We thus have that $w_0^{-1} s_i w_0 = s_{n-i}$ and it follows that $s_i w_0 = w_0 s_{n-1}$. From Lemma 2.3.11 we obtain that

$$\sigma_i \Delta = \Delta \sigma_{n-i}$$

and thus that $\tau(\sigma_i) = \sigma_{n-i}$. ∎

Definition 2.3.14. The Artin group $\mathcal{A}(B_n)$ is the group generated by $\sigma_0, ..., \sigma_{n-1}$ with the relations

$$\begin{aligned}
\sigma_i \sigma_j &= \sigma_j \sigma_i && \text{if } |i-j| \geq 2, \\
\sigma_i \sigma_j \sigma_i &= \sigma_j \sigma_i \sigma_j && \text{if } |i-j| = 1 \text{ and } i, j \geq 1 \\
\sigma_0 \sigma_1 \sigma_0 \sigma_1 &= \sigma_1 \sigma_0 \sigma_1 \sigma_0.
\end{aligned} \tag{2.3.2}$$

Recall from Section 1.5 that with this choice of labels on the simple reflections, the vector $(1, 2, \ldots, n)$ is in the fundamental domain of B_n and that the generators act on $\mathbb{R}^n$ in the following fashion:

$$s_0 : (x_1, x_2, \ldots, x_n) \mapsto (-x_1, x_2, \ldots, x_n)$$
$$s_i : (x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) \mapsto (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n) \quad \text{if } i \neq 0.$$

We also know that the longest element of B_n is given by $w_0 = (s_0 s_1 \ldots s_{n-1})^n$. By a simple computation, we see that it is the map

$$w_0 : (x_1, x_2, \ldots, x_n) \mapsto (-x_1, -x_2, \ldots, -x_n).$$

Lemma 2.3.15. *In $\mathcal{A}(B_n)$, the fundamental element is the element given by*

$$\Delta = (\sigma_0 \sigma_1 \ldots \sigma_{n-1})^n.$$

For the Artin group of type B_n

Proposition 2.3.16. *In the Artin group $\mathcal{A}(B_n)$, for every i, we have $\tau(\sigma_i) = \sigma_i$.*

Proof: We first begin by applying $w_0^{-1} s_i w_0 = w_0 s_i w_0$ in B_n to $(1, 2, \ldots, n)$. If $i \neq 0$, we have

$$\begin{aligned}
(w_0^{-1} s_i w_0)((1, 2, \ldots, n)) &= w_0(s_i(w_0((1, 2, \ldots, n)))) \\
&= w_0(s_i((-1, -2, \ldots, -n))) \\
&= w_0((-1, \ldots, -(i-1), -(i+1), -i, -(i+2), \ldots, -n)) \\
&= (1, \ldots, i-1, i+1, i, i+2, \ldots, n) \\
&= s_i((1, 2, \ldots, n)).
\end{aligned}$$

When $i = 0$, we have

$$\begin{aligned}
(w_0^{-1} s_0 w_0)((1, 2, \ldots, n)) &= w_0(s_0(w_0((1, 2, \ldots, n)))) \\
&= w_0(s_0((-1, -2, \ldots, -n))) \\
&= w_0((1, -2, \ldots, -n)) \\
&= (-1, 2, \ldots, n) \\
&= s_0((1, 2, \ldots, n)).
\end{aligned}$$

We thus have that $w_0^{-1} s_i w_0 = s_i$ for all i, and it follows that $s_i w_0 = w_0 s_i$. From Lemma 2.3.11 we obtain that

$$\sigma_i \Delta = \Delta \sigma_i$$

and thus that $\tau(\sigma_i) = \sigma_i$. ∎

For the Artin group of type D_n

Definition 2.3.17. The Artin group $\mathcal{A}(D_n)$ is the group generated by $\sigma_0, ..., \sigma_{n-1}$ with the relations

$$\begin{aligned} \sigma_i \sigma_j &= \sigma_j \sigma_i && \text{if } |i-j| \geq 2, i, j \geq 1, \\ \sigma_i \sigma_j \sigma_i &= \sigma_j \sigma_i \sigma_j && \text{if } |i-j| = 1 \text{ and } i, j \geq 1 \\ \sigma_0 \sigma_i &= \sigma_i \sigma_0 && \text{if } i \neq 2, \\ \sigma_0 \sigma_2 \sigma_0 &= \sigma_2 \sigma_0 \sigma_2. \end{aligned} \tag{2.3.3}$$

Again, the choice of labels gives us that $(1, 2, \ldots, n)$ is in the fundamental domain of D_n. The simple reflections act in the following way on a given vector:

$$\begin{aligned} s_0 &: (x_1, x_2, x_3, \ldots, x_n) \mapsto (-x_2, -x_1, x_3, \ldots, x_n) \\ s_i &: (x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) \mapsto (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n) \quad \text{if } i \neq 0. \end{aligned}$$

The longest element is given by $w_0 = (s_0 s_1 \cdots s_{n-1})^{n-1}$ and computing this reflection we have that

$$w_0 : (x_1, x_2, \ldots, x_n) \mapsto ((-1)^{n-1} x_1, -x_2, \ldots, -x_n)$$

since

$$s_0 s_1 \cdots s_{n-1} : (x_1, x_2, \ldots, x_n) \mapsto (-x_1, -x_n, x_2 \ldots, x_{n-1}).$$

This means that the element w_0 depends on the parity of n and we shall see that this will affect the automorphism τ.

Lemma 2.3.18. *In $\mathcal{A}(D_n)$, the fundamental element is the element given by*

$$\Delta = (\sigma_0\sigma_1\ldots\sigma_{n-1})^{n-1}.$$

Proposition 2.3.19. *In the Artin group $\mathcal{A}(D_n)$, we have that if n is even,*

$$\tau(\sigma_i) = \Delta^{-1}\sigma_i\Delta = \sigma_i$$

for any i. If n is odd, we have that

$$\begin{aligned}
\tau(\sigma_i) &= \sigma_i \quad \textit{for } i \geq 2,\\
\tau(\sigma_1) &= \sigma_0 \quad \textit{and}\\
\tau(\sigma_0) &= \sigma_1.
\end{aligned}$$

Proof: If n is even, the computations for $w_0^{-1}s_iw_0 = w_0s_iw_0$ applied to $(1,2,\ldots,n)$ in D_n are identical to those in B_n for $i \neq 0$, so $w_0s_iw_0 = s_i$. If $i = 0$, we have

$$\begin{aligned}
(w_0^{-1}s_0w_0)((1,2,\ldots,n)) &= w_0(s_0(w_0((1,2,\ldots,n))))\\
&= w_0(s_0((-1,-2,\ldots,-n)))\\
&= w_0((2,1,-3,\ldots,-n))\\
&= (-2,-1,3,\ldots,n)\\
&= s_0((1,2,\ldots,n)).
\end{aligned}$$

We thus have that $w_0^{-1}s_iw_0 = s_i$ for all i when n is even.

If n is odd, the computations for $w_0s_iw_0$ applied to $(1,\ldots,n)$ are as follows. When $i \geq 2$,

$$\begin{aligned}
(w_0^{-1}s_iw_0)((1,2,\ldots,n)) &= w_0(s_i(w_0((1,2,\ldots,n))))\\
&= w_0(s_i((1,-2,\ldots,-n)))\\
&= w_0((1,-2,\ldots,-(i-1),-(i+1),-i,-(i+2),\ldots,-n))\\
&= (1,2,\ldots,i-1,i+1,i,i+2,\ldots,n)\\
&= s_i((1,2,\ldots,n)).
\end{aligned}$$

When $i = 0$,

$$\begin{aligned}(w_0^{-1}s_0w_0)((1,2,\ldots,n)) &= w_0(s_0(w_0((1,2,\ldots,n)))) \\ &= w_0(s_0((1,-2,\ldots,-n))) \\ &= w_0((2,-1,-3,\ldots,-n)) \\ &= (2,1,3,\ldots,n) \\ &= s_1((1,2,\ldots,n)).\end{aligned}$$

Finally, when $i = 1$,

$$\begin{aligned}(w_0^{-1}s_1w_0)((1,2,\ldots,n)) &= w_0(s_1(w_0((1,2,\ldots,n)))) \\ &= w_0(s_1((1,-2,\ldots,-n))) \\ &= w_0((-2,1,-3,\ldots,-n)) \\ &= (-2,-1,3,\ldots,n) \\ &= s_0((1,2,\ldots,n)).\end{aligned}$$

We deduce the given formulae for $\tau(\sigma_i)$ as before. ∎

For the Artin group of type $I_2(m)$

Finally, we look at the Artin group on two generators $\mathcal{A}(I_2(m))$ as it will play an important role in Chapter 4.

Definition 2.3.20. The Artin group $\mathcal{A}(I_2(m))$ is the group generated by σ_1, σ_2 with the relation

$$_m(\sigma_1,\sigma_2) = \ _m(\sigma_2,\sigma_1) \tag{2.3.4}$$

where, for $\{i,j\} = \{1,2\}$,

$$_m(\sigma_i,\sigma_j) = \begin{cases} (\sigma_i\sigma_j)^{\frac{m}{2}} & \text{if } m \text{ is even,} \\ (\sigma_i\sigma_j)^{\frac{m-1}{2}}\sigma_i & \text{if } m \text{ is odd.}\end{cases}$$

Proposition 2.3.21. *In $\mathcal{A}(I_2(m))$ the* fundamental element *is the element given by*

$$\Delta = {}_m(\sigma_1, \sigma_2) = {}_m(\sigma_2, \sigma_1)$$

Proof: In the Coxeter group $I_2(m)$, the only relations are $s_1^2 = s_2^2 = 1$ and $(s_1 s_2)^m = 1$. It follows that any reduced expression in $I_2(m)$ is of the form $(s_i s_j)^k$ or $(s_i s_j)^k s_i$, with $k \leq \frac{m}{2}$ and $i \neq j$. As such, we have that ${}_m(s_1, s_2) = {}_m(s_2, s_i)$ are reduced expressions.

We are left to argue that, for $i \neq j$, ${}_m(s_i, s_j)$ is the longest element $w_0 \in I_2(m)$. We observe that

$${}_m(s_i, s_j)s_i = (s_i s_j)^{\frac{m}{2}} s_i = (s_j s_i)^{\frac{m}{2}} s_i = (s_j s_i)^{\frac{m}{2}-1} s_j$$

when m is even and $\{i, j\} = \{1, 2\}$. We then have that

$$\lambda({}_m(s_i, s_j)s_i) = \lambda((s_j s_i)^{\frac{m}{2}-1} s_j) = m - 1 < m = \lambda({}_m(s_i, s_j))$$

and it follows that ${}_m(s_i, s_j)s_i$ is not reduced. The argument when m is odd is similar.

As such, $w_0 = {}_m(s_1, s_2) = {}_m(s_2, s_1)$ since $\lambda({}_m(s_i, s_j)s_k) < \lambda({}_m(s_i, s_j))$ for every k. We conclude that $\Delta = {}_m(\sigma_1, \sigma_2) = {}_m(\sigma_2, \sigma_1)$. ∎

Proposition 2.3.22. *In the Artin group $\mathcal{A}(I_2(m))$, we have that if m is even,*

$$\tau(\sigma_i) = \sigma_i$$

for any i. If m is odd, we have, for $\{i, j\} = \{1, 2\}$, that

$$\tau(\sigma_i) = \sigma_j$$

Proof: Suppose that m is even. Then $\Delta = (\sigma_i \sigma_j)^{\frac{m}{2}}$ for $i \neq j$. We compute directly

$$\Delta \sigma_i = (\sigma_i \sigma_j)^{\frac{m}{2}} \sigma_i = \sigma_i (\sigma_j \sigma_i)^{\frac{m}{2}} = \sigma_i \Delta.$$

Thus for m even, $\tau(\sigma_i) = \sigma_i$. If m is odd, then $\Delta = (\sigma_i\sigma_j)^{\frac{m}{2}}\sigma_i$ for $\{i,j\} = \{1,2\}$. We again compute directly

$$\Delta\sigma_i = (\sigma_j\sigma_i)^{\frac{m}{2}}\sigma_j\sigma_i = \sigma_j\sigma_i(\sigma_j\sigma_i)^{\frac{m}{2}} = \sigma_j\Delta.$$

Thus, when m is odd, $\tau(\sigma_i) = \sigma_j$. ∎

2.4 Equivalence of reduced expressions

In this section, let (W, S) be an arbitrary Coxeter system and let $\mathcal{A}$ be the associated Artin group. Recall that we have a natural group homomorphism $\rho : \mathcal{A}^+ \to W$ given by $\rho(\sigma_i) = s_i$. We also have that λ is the length function on W and that $\ell(a)$ denotes the word length of a word a in $\mathcal{A}^+$.

The goal of this section is to prove the following theorem that we stated without proof in Section 2.2. The proof we give is independant of any result given since then. The proof and notation is based on the original proof for the braid group given in [KT08, Section 4.1].

Theorem 2.4.5. *If $s_{i_1}\cdots s_{i_k}$ and $s_{j_1}\cdots s_{j_k}$ are reduced expressions for the same element $w \in W$, then $\sigma_{i_1}\cdots\sigma_{i_k} = \sigma_{j_1}\cdots\sigma_{j_k}$ in $\mathcal{A}^+$.*

For the rest of this section, we will use a sequence notation for $\mathcal{A}^+$ to simplify and clarify the proofs and the notation. Thus, instead of writing $\sigma_{i_1}\cdots\sigma_{i_k}$ we will write $(i_1, \cdots, i_k)$. A concatenation of sequences will imply the product in the group.

We will begin by defining a new operation on vectors that will be used in the proof of the main theorem of this section.

Definition 2.4.1. Let $n > 0$ be an integer and let $\mathbf{i}, \mathbf{j} \in \mathbb{Z}^n_{>0}$, with $\mathbf{i} = (i_1, \cdots, i_n)$ and $\mathbf{j} = (j_1, \cdots, j_n)$. Define the *sliding* operation $\looparrowright : \mathbb{Z}^n_{>0} \times \mathbb{Z}^n_{>0} \to \mathbb{Z}^n_{>0}$ by

$$\mathbf{i} \looparrowright \mathbf{j} = (i_1, j_1, \cdots, j_{n-1}).$$

The $\looparrowright$ operation can be seen as pushing the first component of $\mathbf{i}$ into $\mathbf{j}$ and removing the last one of $\mathbf{j}$.

Observe that $\looparrowright$ is a non-commutative and non-associative operation.

Lemma 2.4.2. *If* $\mathbf{i}, \mathbf{j}, \mathbf{k} \in \mathbb{Z}^n_{>0}$*, then*

$$(\mathbf{i} \looparrowright \mathbf{j}) \looparrowright \mathbf{k} = \mathbf{i} \looparrowright \mathbf{k}.$$

Proof: The proof of this lemma is quite straightforward. By computation, we have:

$$\begin{aligned}(\mathbf{i} \looparrowright \mathbf{j}) \looparrowright \mathbf{k} &= (i_1, j_1, \cdots, j_{n-1}) \looparrowright \mathbf{k} \\ &= (i_1, k_1, \cdots, k_{n-1}) \\ &= \mathbf{i} \looparrowright \mathbf{k}.\end{aligned}$$

■

Using this lemma, every expression in $\looparrowright$ can be reduced to an expression of the form $\mathbf{i_1} \looparrowright (\mathbf{i_2} \looparrowright \cdots (\mathbf{i_{k-1}} \looparrowright \mathbf{i_k}) \cdots)$. Thus, without any risk of confusion, we can drop the parentheses and write $\mathbf{i_1} \looparrowright \cdots \looparrowright \mathbf{i_k}$ for the previous reduced expression.

To prove the main theorem, we introduce a notation for a logical implication that will be used in the next proofs.

Definition 2.4.3. Given $\mathbf{i}, \mathbf{j} \in \{1, 2, \cdots, n\}^k$ for some integer $k > 0$, define $P(\mathbf{i}, \mathbf{j}, k)$ to be the following logical implication:

If:

$s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k}$ and

$s_{i_1} \cdots s_{i_k}$ and $s_{j_1} \cdots s_{j_k}$ are reduced expressions in W

Then:

$(i_1, \cdots, i_k) = (j_1, \cdots, j_k)$,

where the last equality is in the monoid $\mathcal{A}^+$.

We note that if $P(\mathbf{i}, \mathbf{j}, k)$ is true, so is $P(\mathbf{j}, \mathbf{i}, k)$. The next lemma will be used in the inductive proof of the main theorem.

Lemma 2.4.4. *Let $k > 0$ be an integer and let $\mathbf{i}, \mathbf{j} \in \{1, 2, \cdots, n\}^k$. Suppose that the following conditions hold:*

1. *$s_{i_1} \cdots s_{i_k}$, $s_{j_1} \cdots s_{j_k}$ and $s_{i_1} s_{j_1} \cdots s_{j_{k-1}}$ are reduced expressions,*
2. *$s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k} = s_{i_1} s_{j_1} \cdots s_{j_{k-1}}$ in W,*
3. *For any $l < k$, the implication $P(\mathbf{u}, \mathbf{v}, l)$ is true for every $\mathbf{u}, \mathbf{v} \in \{1, 2, \cdots, n\}^l$.*

If $P(\mathbf{i} \looparrowright \mathbf{j}, \mathbf{j}, k)$ is true, then so is $P(\mathbf{i}, \mathbf{j}, k)$.

Proof: Suppose $P(\mathbf{i} \looparrowright \mathbf{j}, \mathbf{j}, k)$ is true. By hypothesis $s_{i_1} \cdots s_{i_k}$ and $s_{j_1} \cdots s_{j_k}$ are reduced expressions in W such that $s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k}$. Thus, to show that $P(\mathbf{i}, \mathbf{j}, k)$ is true, it remains to show that $(i_1, \cdots, i_k) = (j_1, \cdots, j_k)$.

We have that $s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k} = s_{i_1} s_{j_1} \cdots s_{j_{k-1}}$ are all reduced. By multiplying the set of equalities by s_{i_1} on the left, we obtain that

$$s_{i_2} \cdots s_{i_k} = s_{j_1} \cdots s_{j_{k-1}}.$$

Both of these expressions are reduced in W since they are subexpressions of a reduced expression. By the third hypothesis, we get that $(i_2, \cdots, i_k) = (j_1, \cdots, j_{k-1})$. We then observe that

$$\begin{aligned} (i_1, \cdots, i_k) &= (i_1)(i_2, \cdots, i_k) \\ &= (i_1)(j_1, \cdots, j_{k-1}) \\ &= (i_1, j_1, \cdots, j_{k-1}) \\ &= (j_1, \cdots, j_k), \end{aligned}$$

with the last equality following from $P(\mathbf{i} \looparrowright \mathbf{j}, \mathbf{j}, k)$. $P(\mathbf{i}, \mathbf{j}, k)$ then follows immediately. ∎

We can now prove the main theorem of this section.

Theorem 2.4.5. *If $s_{i_1} \cdots s_{i_k}$ and $s_{j_1} \cdots s_{j_k}$ are reduced expressions for the same element $w \in W$, then $(i_1, \cdots, i_k) = (j_1, \cdots, j_k)$ in $\mathcal{A}^+$.*

Proof: To prove this theorem, we will prove that $P(\mathbf{i}, \mathbf{j}, k)$ is true for every integer $k \geq 0$ and every $\mathbf{i}, \mathbf{j} \in \{1, \ldots, n\}^k$. If the hypotheses of $P(\mathbf{i}, \mathbf{j}, k)$ are false, that is if $s_{i_1} \cdots s_{i_k} \neq s_{j_1} \cdots s_{j_k}$ or if either expression is not reduced, we are done.

We shall thus assume that the hypotheses of $P(\mathbf{i}, \mathbf{j}, k)$ hold and will proceed to show the implication is true by induction on k. If $k = 0$, then $w = 1$ is the identity of W and has only one reduced expression. If $k = 1$ then $w = s_i = s_j$ for some $i = j$. Then w has only one reduced expression. Thus $P(\mathbf{i}, \mathbf{j}, 0)$ and $P(\mathbf{i}, \mathbf{j}, 1)$ are true.

Now let $k \geq 2$ and suppose that $P(\mathbf{u}, \mathbf{v}, l)$ is true for every integer $l < k$ and every $\mathbf{u}, \mathbf{v} \in \{1, \ldots, n\}^l$. From $s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k}$ we can obtain the equality $s_{i_2} \cdots s_{i_k} = s_{i_1} s_{j_1} \cdots s_{j_k}$, with $s_{i_2} \cdots s_{i_k}$ being a reduced expression.

By the Exchange condition (see Theorem 1.4.1) for Coxeter groups, there is some unique integer p, $1 \leq p \leq k$, such that

$$s_{i_2} \cdots s_{i_k} = s_{i_1} s_{j_1} \cdots s_{j_k} = s_{j_1} \cdots \widehat{s_{j_p}} \cdots s_{j_k}. \tag{2.4.1}$$

We observe that $s_{j_1} \cdots \widehat{s_{j_p}} \cdots s_{j_k}$ is a reduced expression. By the induction hypothesis, we have that $(i_2, \cdots, i_k) = (j_1, \cdots, \widehat{j_p}, \cdots, j_k)$. It then follows that

$$(i_1, \cdots, i_k) = (i_1, j_1, \cdots, \widehat{j_p}, \cdots, j_k). \tag{2.4.2}$$

We now have two cases: when $p < k$ and when $p = k$.

Case 1: If $p < k$, we obtain, by left cancellation from (2.4.1), that $s_{i_1} s_{j_1} \cdots s_{j_{p-1}} = s_{j_1} \cdots s_{j_p}$. Applying the induction hypothesis again on these words, we obtain that $(i_1, j_1, \cdots, j_{p-1}) = (j_1, \cdots, j_p)$. Using this in (2.4.2) we obtain

$$\begin{aligned}
(i_1, \cdots, i_k) &= (i_1, j_1, \cdots, \widehat{j_p}, \cdots, j_k) \\
&= (i_1, j_1, \cdots, j_{p-1})(j_{p+1}, \cdots, j_k) \\
&= (j_1, \cdots, j_p)(j_{p+1}, \cdots, j_k) \\
&= (j_1, \cdots, j_k),
\end{aligned}$$

thus proving the desired result.

Case 2: If $p = k$, we have that

$$s_{j_1} \cdots s_{j_k} = s_{i_1} s_{j_1} \cdots s_{j_{k-1}}. \tag{2.4.3}$$

To prove the desired equality, it suffices to show that $P(\mathbf{i} \looparrowright \mathbf{j}, \mathbf{j}, k)$ is true, by Lemma 2.4.4. We then proceed as above by multiplying (2.4.3) on the left by s_{j_1}. We then fall in either case 1, where we can complete the proof, or case 2 again. Without loss of generality, we will now only consider when the argument falls into case 2, that is when the last element is removed by the Exchange condition.

If we fall in case 2 again, left multiplication by s_{j_1} in (2.4.3) yields $s_{j_2} \cdots s_{j_k} = s_{i_1} s_{j_1} \cdots s_{j_{k-2}}$, which are reduced expressions, and we obtain that

$$s_{j_1} \cdots s_{j_k} = s_{i_1} s_{j_1} \cdots s_{j_{k-1}} = s_{j_1} s_{i_1} s_{j_1} \cdots s_{j_{k-2}}. \tag{2.4.4}$$

We then apply Lemma 2.4.4 to (2.4.4) since all three expressions are reduced. We thus only have to show that $P(\mathbf{j} \looparrowright \mathbf{i} \looparrowright \mathbf{j}, \mathbf{i} \looparrowright \mathbf{j}, k)$ is true.

If $m(i_1, j_1) = 2$, we have that $s_{i_1} s_{j_1} = s_{j_1} s_{i_1}$ and $(i_1, j_1) = (j_1, i_1)$. We thus multiply (2.4.4) by $s_{i_1} s_{j_1}$ to obtain that

$$s_{j_2} \cdots s_{j_{k-1}} = s_{j_1} \cdots s_{j_{k-2}}.$$

From the induction hypothesis, we obtain that $(j_2, \cdots, j_{k-1}) = (j_1, \cdots, j_{k-2})$ and, using the relations in $\mathcal{A}^+$, that

$$\begin{aligned}(i_1, j_1, \cdots, j_{k-1}) &= (i_1, j_1)(j_2, \cdots, j_{k-1}) \\ &= (j_1, i_1)(j_1, \cdots, j_{k-2}) \\ &= (j_1, i_1, j_1, \cdots, j_{k-2}),\end{aligned}$$

which shows that $P(\mathbf{j} \looparrowright \mathbf{i} \looparrowright \mathbf{j}, \mathbf{i} \looparrowright \mathbf{j}, k)$ holds.

We note that when we are in case 2, the conditions of Lemma 2.4.4 are satisfied and as such the implication

$$P(\mathbf{u} \looparrowright \mathbf{v}, \mathbf{v}, k) \Rightarrow P(\mathbf{u}, \mathbf{v}, k) \tag{2.4.5}$$

holds whenever the hypothesis for $P(\mathbf{u}, \mathbf{v}, k)$ are met.

In the case where $2 < m(i_1, j_1) \leq k$, we repeatedly use the implication (2.4.5) until we can apply the relation (2.1.1) and prove the desired result in the same way as for the case $m(i_1, j_1) = 2$. It is worth noting that these relations will come up, for we increase the length of the relation by exactly one at every step and the first terms are never the same on both side of the equality of the reduced expressions.

If $m(i_1, j_1) > k$, then naively doing the argument apparently leads to a contradiction, for we would end with a relation of the same form as (2.1.1) but with $k < m(i_1, j_1)$ terms. This can not happen because $m(i_1, j_1)$ is the order of $s_{i_1} s_{j_1}$. This implies that we can not have two equivalent reduced expression $s_{i_1} \cdots s_{i_k}$ and $s_{j_1} \cdots s_{j_k}$ with $m(i_1, j_1) > k$ for the same element w. ∎

This result provides us with an interesting insight on how the reduced expressions work in a Coxeter system. Indeed, the result gives us that if two reduced words are equivalent in (W, S), then their associated words in the Artin group are equivalent. This leads to the following corollary on reduced words.

Corollary 2.4.6. *If $s_{i_1} \cdots s_{i_k} = s_{j_1} \cdots s_{j_k}$ are reduced expressions in (W, S), then one expression can be obtained from the other one by using only the relations written as in (2.1.1).*

Chapter 3

A Normal Form for Artin Groups of Finite Type

In this chapter, let $\mathcal{A}$ denote an Artin group of finite type, i.e. one associated to a finite reflection group (W, S). We will only cover the cases where (W, S) is of type A_n, B_n and D_n. We will not cover the other finite reflection groups, those of type E_6, E_7, E_8, F_4, H_3, H_4 and $I_2(m)$. The reason we do not cover these finite reflection groups in detail is that these groups do not offer the possibility of arbitrarily increasing the number of generators of the group. This makes them unsuitable for cryptography, since the current solution to the conjugacy search problem in Artin groups of finite type is conjectured to be polynomial in the number of generators (see [FGM03]). This chapter is a generalization of the work done for the braid group in [ERM94] and [CKL+01].

3.1 Partial order on $\mathcal{A}$

Definition 3.1.1. For any $a, b \in \mathcal{A}$ we can define the following relations:

1. $a \leq b$ if there exist $p, q \in \mathcal{A}^+$ such that $b = paq$.
2. $a \leq_L b$ if there exists a $p \in \mathcal{A}^+$ such that $b = ap$.

3. $a \leq_R b$ if there exists a $p \in \mathcal{A}^+$ such that $b = pa$.

If $a \leq_L b$ (resp. $a \leq_R b$), then a is a left (resp. right) divisor of b as per Definition 2.1.5.

It can be shown that these relations are partial orders on $\mathcal{A}$, as done in Proposition 1.7.3. From these partial orders, we can deduce that $b \in \mathcal{A}^+$ if and only if $1 \leq b$ and that $a \leq b$ if and only if $b^{-1} \leq a^{-1}$. From Propositions 2.2.3 and 2.3.3, it follows that for any Artin generator σ_i, $1 \leq \sigma_i \leq \Delta$.

Recall from Corollary 2.3.12 that the map $\tau : \mathcal{A}^+ \to \mathcal{A}^+$ defined by $\tau(\sigma_i) = \Delta^{-1}\sigma_i\Delta = \Delta\sigma_i\Delta^{-1}$ is a well defined automorphism. We refer the reader to Section 2.3 for the explicit description of τ in each of the groups we cover in this chapter. From these descriptions, we note that $\tau^2(\sigma_i) = \sigma_i$.

Proposition 3.1.2. *Let $a \in \mathcal{A}$. If $a \leq \Delta^s$ for some $s \in \mathbb{Z}$, then $a \leq_R \Delta^s$ and $a \leq_L \Delta^s$.*

Proof: Suppose $a \leq \Delta^s$ and write $\Delta^s = paq$ with $p, q \in \mathcal{A}^+$. We have

$$\Delta^s q = \tau^s(q)\Delta^s = \tau^s(q)paq.$$

Then, by right cancellation of q, it follows that

$$\Delta^s = \tau^s(q)pa = (\tau^s(q)p)a,$$

so $a \leq_L \Delta^s$, since $\tau(q) \in \mathcal{A}^+$. In a similar way for p, we obtain that $a \leq_R \Delta^s$. ∎

By a similar argument, we have the following proposition.

Proposition 3.1.3. *If $\Delta^r \leq a$ for some $r \in \mathbb{Z}$, then $\Delta^r \leq_R a$ and $\Delta^r \leq_L a$.*

3.2 Left-weighted factorisation

We will now work mostly in $\mathcal{A}^+$ and with the factorisations of positive elements due to the following lemma.

Lemma 3.2.1. *For every $b \in \mathcal{A}$, we can write $b = \Delta^r p$ for some integer $r \leq 0$ and positive element $p \in \mathcal{A}^+$.*

Proof: Since $\sigma_i \leq_R \Delta$, we have that $\Delta = q_i\sigma_i$, for some $q_i \in \mathcal{A}^+$. We then have $\sigma_i^{-1} = \Delta^{-1}q_i$.

Thus, given any element $b \in \mathcal{A}$ as a word in A^*, we can replace all σ_i^{-1} by $\Delta^{-1}q_i$. We then repeatedly use the fact that $\sigma_i\Delta^{-1} = \Delta^{-1}\tau(\sigma_i)$ to bring all powers of Δ^{-1} to the left and write $b = \Delta^{-s}p$, where $s \geq 0$ and $p \in \mathcal{A}^+$. ∎

Given an Artin group $\mathcal{A}$, let $I_\mathcal{A}$ be the set of labels on the generators.

Definition 3.2.2. Let $p \in \mathcal{A}^+$. The *starting set* $S(p) \subset I_\mathcal{A}$ of p is the set

$$S(p) = \{i | \sigma_i \leq_L p\}.$$

Similarly, the *finishing set* of p is the set

$$F(p) = \{i | \sigma_i \leq_R p\}.$$

Lemma 3.2.3. *For any element $b \in \mathcal{A}$, we have that $F(b) = S(Rev(b))$, where $Rev : \mathcal{A}^+ \to \mathcal{A}^+$ is the anti-automorphism on $\mathcal{A}^+$ given by $Rev(\sigma_{i_1} \cdots \sigma_{i_k}) = \sigma_{i_k} \cdots \sigma_{i_1}$.*

Proof: The fact that Rev is an anti-automorphism follows from the fact that all the relations in $\mathcal{A}^+$ are palindromic. Suppose $i \in F(b)$. We can then write $b = \sigma_{j_1} \cdots \sigma_{j_k}\sigma_i$. It follows that $Rev(b) = \sigma_i\sigma_{j_k} \cdots \sigma_{j_1}$ and that $i \in S(Rev(b))$. We thus have $F(b) \subset S(Rev(b))$. The reverse inclusion is proved similarly. ∎

In other words, the finishing set of an element b is the starting set of the element given by reading b backwards. It follows that if we characterize the starting sets, we have also characterized the finishing sets. As such, we shall only prove the results for the starting sets.

As an example, we consider the braid group $\mathcal{B}_n = \mathcal{A}(A_{n-1})$.

Example 3.2.4. Consider the braid $b = \sigma_1\sigma_2\sigma_3\sigma_1 \in \mathcal{B}_4^+$. Using the braid relations we can write :

$$\sigma_1\sigma_2\sigma_3\sigma_1 = \sigma_1\sigma_2\sigma_1\sigma_3 = \sigma_2\sigma_1\sigma_2\sigma_3.$$

These are all the representations of b as words in $\mathcal{A}^+$. Indeed, applying the braid relations to any of the three expressions gives no new ones. It then follows from these equalities that

$$S(b) = \{1, 2\} \text{ and } F(b) = \{1, 3\}.$$

As seen in Example 3.2.4, this method of finding the starting sets and finishing sets of $p \in \mathcal{A}^+$ is not efficient, because we are required to first find all the possible representations of p. The next proposition will give us an efficient way to find the starting set of a reduced element of $\mathcal{A}$.

Proposition 3.2.5. *Let $a_w \in \mathcal{A}^{Red}$ with $\rho(a_w) = w \in W$. Let α_{s_i} be the simple root associated with the simple reflection $s_i \in S$. Then, the following are equivalent.*

1. $i \in S(a_w)$.
2. $s_i \in S(w)$.
3. $\lambda(s_i w) < \lambda(w)$.
4. $w(\alpha_{s_i}) < 0$.

Proof: The equivalence of (2) and (3) follows from Propositions 1.7.5 and 1.4.7, while the equivalence of (3) and (4) follows from Theorem 1.3.3. We are left to prove the equivalence of (1) and (2).

Suppose $i \in S(a_w)$. Then we can write $a_w = \sigma_i\sigma_{j_1}\cdots\sigma_{j_k}$. We then have that $w = \rho(a_w) = s_i s_{j_1}\cdots s_{j_k}$. It follows that $s_i \in S(w)$.

Conversely, suppose that $s_i \in S(w)$. Then from Propositions 1.7.5 and 1.4.7, we have that there exists a reduced expression $s_i s_{j_1} \cdots s_{j_k}$ for w. It follows that $a_w = \sigma_i \sigma_{j_1} \cdots \sigma_{j_k}$ and that $i \in S(a_w)$. ∎

Recall that for the three types of finite reflection groups we will be working with, we chose, in Section 1.5, the set of simple roots, S, such that the vector $(x_1, x_2, \ldots, x_n)$ is in the interior, C, of the fundamental domain D of (W, S) whenever $0 < x_1 < x_2 < \ldots < x_n$.

Remark 3.2.6. Let (W, S) be a finite reflection group acting on $\mathbb{R}^n$ and let $x = (x_1, x_2, \ldots, x_n)$ be a vector in $\mathbb{R}^n$. Then for some reflection $w \in W$, we will write $w(x_1, x_2, \ldots, x_n) = (w(x_1), w(x_2), \ldots, w(x_n))$.

For example, if $w(1, 2, 3) = (-2, 3, 1)$, we will write $w(1) = -2$, $w(2) = 3$ and $w(3) = 1$.

We now proceed with explicit descriptions for the starting set of $a_w \in \mathcal{A}^{Red}$ for the three Artin groups of finite type that we will be studying. Unless otherwise noted, let $x = (x_1, \ldots, x_n)$ be such that $0 < x_1 < x_2 < \ldots < x_n$. We will make use of Remark 1.6.4 and the finite reflection groups presentation given in Section 1.5. In the next three proofs, B will be the bilinear form defined in Section 1.2.

Proposition 3.2.7. *Let a_w be a reduced element of $\mathcal{B}_{n+1} = \mathcal{A}(A_n)$. Then $i \in S(a_w)$ if and only if*

$$w(x_{i+1}) < w(x_i).$$

Proof: In A_n, recall that we chose our simple roots to be $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \ldots, n$. With this choice of root system, x is in the fundamental domain. By Proposition 3.2.5, $i \in S(a_w)$ if and only if $w(\alpha_{s_i}) < 0$. Since x is in the fundamental domain, this is equivalent to $B(w(x), \alpha_{s_i}) = w(x_{i+1}) - w(x_i) < 0$, i.e.

$$w(x_{i+1}) < w(x_i).$$

∎

Proposition 3.2.8. *Let a_w be a reduced element of $\mathcal{A}(B_n)$. Then $i \in S(a_w)$ if and only if*

$$\begin{aligned} &w(x_{i+1}) < w(x_i) && \text{for } i \geq 1 \\ &w(x_1) < 0 && \text{for } i = 0. \end{aligned}$$

Proof: In B_n, recall that we chose our simple roots to be $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \ldots, n-1$ and $\alpha_{s_0} = \epsilon_1$. With this choice of root system, x is in the fundamental domain. The proof when $i \geq 1$ is exactly the same as for the group $\mathcal{A}(A_n)$.

For the case $i = 0$, we have, by a similar argument, that $0 \in S(a_w)$ if and only if $B(w(x), \alpha_0) = w(x_1) < 0$, as required. ∎

Proposition 3.2.9. *Let a_w be a reduced element of $\mathcal{A}(D_n)$. Then $i \in S(a_w)$ if and only if*

$$\begin{aligned} &w(x_{i+1}) < w(x_i) && \text{for } i \geq 1 \\ &w(x_1) + w(x_2) < 0 && \text{for } i = 0. \end{aligned}$$

Proof: In D_n, recall that we chose our simple roots to be $\alpha_{s_i} = \epsilon_{i+1} - \epsilon_i$, for $i = 1, \ldots, n-1$ and $\alpha_{s_0} = \epsilon_1 + \epsilon_2$. Again, we have that x is in the fundamental domain. The proof for $i \geq 1$ is again exactly the same as for the group $\mathcal{A}(A_n)$.

For the cases $i = 0$, $s_0 \in S(a_w)$ if and only if $B(w(x), \alpha_0) = w(x_1) + w(x_2) < 0$, as required. ∎

Now that we can compute the starting sets efficiently, we will move on to prove results needed for the normal form.

Proposition 3.2.10. *Let $a \in \mathcal{A}^{Red}$. Then $\sigma_i a \in \mathcal{A}^{Red}$ if and only if $i \notin S(a)$.*

Also, $a\sigma_i \in \mathcal{A}^{Red}$ if and only if $i \notin F(a)$.

Proof: Let $a \in \mathcal{A}^{Red}$ with $\rho(a) = w$. Suppose first that $\sigma_i a \in \mathcal{A}^{Red}$. We will prove that $i \notin S(a)$ by contradiction. We thus suppose that $i \in S(a)$, so there exists a reduced expression $\sigma_i \sigma_{j_1} \cdots \sigma_{j_k}$ for a and thus $\sigma_i a = \sigma_i^2 \sigma_{j_1} \cdots \sigma_{j_k}$. We then have that

$$\lambda(\rho(\sigma_i a)) = k = \ell(\sigma_i a) - 2 \neq \ell(\sigma_i a),$$

and it follows from Proposition 2.2.3 that $\sigma_i a$ is not reduced. We have our contradiction, thus $i \notin S(a)$.

Conversely, suppose that $i \notin S(a)$. Then by Proposition 3.2.5, the root α_i is not sent to a negative by w. It then follows from Theorem 1.3.3 that $\lambda(s_i w) = \lambda(w) + 1 = \ell(a) + 1 = \ell(\sigma_i a)$ and thus that $\sigma_i a$ is reduced.

The part on the finishing set follows with the use of Lemma 3.2.3. ∎

Definition 3.2.11. A *positive factorisation* for an element $p \in \mathcal{A}^+$ is a factorisation $p = ab$ where $a \in \mathcal{A}^{Red}$ and $b \in \mathcal{A}^+$.

We say that a positive factorisation is *left-weighted* if $S(b) \subset F(a)$. These left-weighted factorisations will allow us to construct a unique normal form for the Artin groups of finite type that can be computed efficiently.

Example 3.2.12. Consider the braid $p = \sigma_1\sigma_2\sigma_1\sigma_2$ in $\mathcal{B}_3$. We first consider the factorisation

$$p = (\sigma_1\sigma_2)(\sigma_1\sigma_2) = ab.$$

In this factorisation, we have that $F(a) = \{2\}$ and $S(b) = \{1\}$. This factorisation is thus not a left-weighted factorisation. On the other hand, if we consider the factorisation

$$p = (\sigma_1\sigma_2\sigma_1)(\sigma_2) = ab,$$

we have that $F(a) = \{1, 2\}$ and $S(b) = \{2\}$. We then have that $S(b) \subset F(a)$ and that this factorisation is left-weighted.

We now proceed to proving the existence and uniqueness of left-weighted factorisations, but we require two lemmas.

Lemma 3.2.13. *Let $p \in \mathcal{A}^{Red}$. Suppose there exist $p_1, p_2 \in \mathcal{A}^+$ such that $p = \sigma_i p_1 = \sigma_j p_2$. Then $p = {}_{m_{i,j}}(\sigma_i, \sigma_j) p_3$ for some $p_3 \in \mathcal{A}^+$ where*

$$ {}_{m_{i,j}}(\sigma_i, \sigma_j) = \begin{cases} (\sigma_i \sigma_j)^{\frac{m(i,j)}{2}} & \text{if } m(i,j) \text{ is even,} \\ (\sigma_i \sigma_j)^{\frac{m(i,j)-1}{2}} \sigma_i & \text{if } m(i,j) \text{ is odd.} \end{cases} $$

Proof: Since p is reduced, we apply Corollary 1.4.5 to $\rho(p) = s_i \rho(p_1) = s_j \rho(p_2)$ to obtain that $\rho(p) = {}_{m_{i,j}}(s_i, s_j) w'$. We then apply Theorem 2.4.5 to this last equality and we obtain that $p = {}_{m_{i,j}}(\sigma_i, \sigma_j) a_{w'}$ and that this is a reduced expression. ∎

It is clear that in any Artin group ${}_{m_{i,j}}(\sigma_i, \sigma_j) = {}_{m_{j,i}}(\sigma_j, \sigma_i)$ for any i, j, because these are its defining relations. Before proving the next lemma, recall that in $\mathcal{A}^+$, $a = b$ if and only if a can be transformed into b by using a applying sequence of the defining relations of $\mathcal{A}$.

Definition 3.2.14. If $a, b \in A^*$ are positive words such that $a = b$ in $\mathcal{A}^+$, let $a \to b$ denote a sequence of relations in (2.1.2) used to transform a into b.

If such a transformation $a \to b$ is a sequence of t relations, the transformation is said to be of *chain-length* t.

We have that $a \to b$ has chain length $t = 0$ if and only if $a = b$ as words over $\mathcal{A}^+$.

Lemma 3.2.15. *Let $p \in \mathcal{A}^+$. Suppose there exist $A, B \in \mathcal{A}^+$ such that $p = \sigma_i A = \sigma_j B$. Then $p = {}_{m_{i,j}}(\sigma_i, \sigma_j) C$ for some $C \in \mathcal{A}^+$.*

This lemma is a more general version of Lemma 3.2.13. The original proof for the braid group, for the following equivalent result, is due to Garside (see [Gar69]).

Lemma 3.2.16. *Let $p \in \mathcal{A}^+$. Suppose there exist $A, B \in \mathcal{A}^+$ such that $p = \sigma_i A = \sigma_j B$. The following statements hold:*

- *If $i = j$, then $A = B$ in $\mathcal{A}^+$.*
- *If $i \neq j$, then there exists $C \in \mathcal{A}^+$ such that $A = \sigma_i^{-1}{}_{m_{i,j}}(\sigma_i, \sigma_j)C$ and $B = \sigma_j^{-1}{}_{m_{j,i}}(\sigma_j, \sigma_i)C$.*

Proof: This proof will be based on the proof given by Garside, but we generalize it to the other Artin groups of finite type. As such, we will use capital letters for words in $\mathcal{A}^+$ in analogy with the original proof.

The proof is done by induction on the word length of A and B. We begin by noting that if $\sigma_i A = \sigma_j B$ in $\mathcal{A}^+$, then $\ell(A) = \ell(B) = \ell(p) - 1$. Now, if $\ell(A) = \ell(B) = 0$, then $A = B = 1$ and $p = \sigma_i = \sigma_j$. Then, we need $i = j$.

If $\ell(A) = 1$, then $\ell(p) = 2$ and we have two cases. If $i = j$, then $\sigma_i A = \sigma_i B$ and it follows that $A = B = \sigma_k$ for some k. If $i \neq j$, then we claim that $p \in \mathcal{A}^{Red}$. The only elements of length 2 that are not reduced are elements of the form σ_k^2 and this case is impossible since $i \neq j$. We can then use Lemma 3.2.13 and obtain that $p = \sigma_i\sigma_j = \sigma_j\sigma_i$. It follows that $A = \sigma_j = \sigma_i^{-1}{}_{m_{i,j}}(\sigma_i, \sigma_j)1$ and that $B = \sigma_i = \sigma_j^{-1}{}_{m_{j,i}}(\sigma_j, \sigma_i)1$. We can then take $C = 1$ and we are done.

We have a double induction and thus two induction hypotheses.

H1 Let $r \geq 0$ be an integer and suppose that the lemma is true for all A of length $0 \leq \ell(A) \leq r$.

H2 Suppose that for $\ell(A) = r + 1$, the lemma is true for all transformations $\sigma_i A \to \sigma_j B$ of chain-length at most t.

We just proved that H1 is true for $r = 1$, so we proceed with the base cases of H2, with $\ell(A) = r + 1$ for any r. If $\sigma_i A \to \sigma_j B$ has chain-length $t = 0$, then $\sigma_i A = \sigma_j B$ as words and it follows that $\sigma_i = \sigma_j$ and $A = B$ in $\mathcal{A}^+$. If the transformation has chain-length $t = 1$, we have two possible cases. If $i = j$, then since there is no defining relations in $\mathcal{A}$ involving only σ_i, we have that the transformation occurs in A and B. Thus, $A \to B$ has chain length 1 and it follows that $A = B$ as required. If $i \neq j$, then

$\sigma_j B$ is obtained from $\sigma_i A$ by the application of exactly one of the defining relations in $\mathcal{A}$. Since the only relation involving both σ_i and σ_j is the relation ${}_{m_{i,j}}(\sigma_i, \sigma_j) = {}_{m_{j,i}}(\sigma_j, \sigma_i)$, it follows that we need $\sigma_i A = {}_{m_{i,j}}(\sigma_i, \sigma_j)C$ and $\sigma_j B = {}_{m_{j,i}}(\sigma_j, \sigma_i)C$ for some $C \in \mathcal{A}^+$. We then get the desired result that $A = \sigma_i^{-1}\,{}_{m_{i,j}}(\sigma_i, \sigma_j)C$ and $B = \sigma_j^{-1}\,{}_{m_{j,i}}(\sigma_j, \sigma_i)C$ and we are done.

Having both base cases in hand, we now proceed to the induction case. Let A and B be words of length $r+1$ such that $\sigma_i A = \sigma_j B$ and suppose that the transformation $\sigma_i A \to \sigma_j B$ has chain length $t+1$. To complete the proof of the lemma, it suffices to show that the lemma holds for these A and B. Write the transformation as

$$\sigma_i A = W_0 \to W_1 \to \cdots \to W_{t+1} = \sigma_j B$$

and pick $W_g = \sigma_k W$ for some g with $0 < g < t+1$. We then have that $\sigma_i A = \sigma_k W = \sigma_j B$ and that the transformations $\sigma_i A \to \sigma_k W$ and $\sigma_k W \to \sigma_j B$ are of chain-length less than $t+1$. We can then apply H2 to both these transformation in different ways depending on the values of i, j and k. There are many cases to cover, but they can be reduced to three types: when $k = i$ or $k = j$, when $i = j$ and $k \neq i$ and when i, j and k are all distinct.

Case 1: $k = i$ or $k = j$.

Without loss of generality, suppose $k = i$. We thus have $\sigma_i A = \sigma_i W$ and $\sigma_i W = \sigma_j B$. Since each equality has a transformation of chain-length at most t, we apply H2 to obtain that $A = W$ and that there exists a C such that $A = W = \sigma_i^{-1}\,{}_{m_{i,j}}(\sigma_i, \sigma_j)C$ and $B = \sigma_j^{-1}\,{}_{m_{j,i}}(\sigma_j, \sigma_i)C$. The result follows.

We shall now assume that k is distinct from i and j.

Case 2: $i = j$, $k \notin \{i, j\}$.

We now have that $\sigma_i A = \sigma_k W = \sigma_i B$. Applying H2 to $\sigma_i A = \sigma_k W$ and $\sigma_k W = \sigma_i B$, we obtain respectively that there exist X, Y such that

$$A = \sigma_i^{-1}\,{}_{m_{i,k}}(\sigma_i, \sigma_k)X, \quad W = \sigma_k^{-1}\,{}_{m_{k,i}}(\sigma_k, \sigma_i)X$$

and

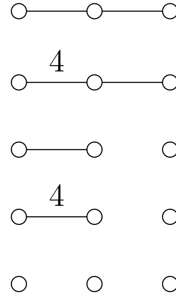
$$B = \sigma_i^{-1}{}_{m_{i,k}}(\sigma_i, \sigma_k)Y, \quad W = \sigma_k^{-1}{}_{m_{k,i}}(\sigma_k, \sigma_i)Y.$$

We observe that $\sigma_k^{-1}{}_{m_{k,i}}(\sigma_k, \sigma_i)X = \sigma_k W_1'$ and $\sigma_k^{-1}{}_{m_{k,i}}(\sigma_k, \sigma_i)Y = \sigma_i W_2'$, where $\ell(W_1') = \ell(W_2') = r$. We can then apply H1 to $W = \sigma_k^{-1}{}_{m_{k,i}}(\sigma_k, \sigma_i)X = \sigma_k^{-1}{}_{m_{k,i}}(\sigma_k, \sigma_i)Y$ exactly $m_{k,i} - 1$ times and obtain that $X = Y$. We then have that

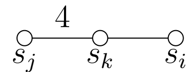
$$A = \sigma_i^{-1}{}_{m_{i,k}}(\sigma_i, \sigma_k)X = \sigma_i^{-1}{}_{m_{i,k}}(\sigma_i, \sigma_k)Y = B,$$

which is what we wanted.

We can now assume that i, j and k are all distinct. Even then, we have to cover all the possible arrangements of values for $m(i,j)$, $m(i,k)$ and $m(j,k)$. That is, to prove the theorem for any Artin group of finite type, we need to cover all the possible permutations of s_i, s_j and s_k as nodes in the following Coxeter sub-graphs:



Since the method is similar for all these cases, we will only show one of the longest ones, that is the case in $\mathcal{A}(B_n)$ where $m(i,j) = 2$, $m(i,k) = 3$ and $m(j,k) = 4$. That is, when we have the following Coxeter sub-graph:



Case 3: If i, j, k are distinct and $m(i,j) = 2$, $m(i,k) = 3$ and $m(j,k) = 4$. In this case, we have the following defining relations:

$$\sigma_i \sigma_j = \sigma_j \sigma_i,$$

$$\sigma_i \sigma_k \sigma_i = \sigma_k \sigma_i \sigma_k,$$

$$\sigma_j \sigma_k \sigma_j \sigma_k = \sigma_k \sigma_j \sigma_k \sigma_j.$$

Here ${}_{m_{i,j}}(\sigma_i, \sigma_j) = \sigma_i\sigma_j$, so our goal is showing that there exists a C such that $A = \sigma_j C$ and $B = \sigma_i C$. Recall that we have that $\sigma_i A = \sigma_k W = \sigma_j B$ and that the transformations $\sigma_i A \to \sigma_k W$ and $\sigma_k W \to \sigma_j B$ are of chain-length less than $t+1$ by choice of $\sigma_k W$.

We again start by applying H2 to $\sigma_i A = \sigma_k W$ and $\sigma_k W = \sigma_j B$ respectively to obtain that there exist X, Y such that

$$A = \sigma_i^{-1}\, {}_{m_{i,k}}(\sigma_i, \sigma_k)X = \sigma_k\sigma_i X, \quad W = \sigma_k^{-1}\, {}_{m_{k,i}}(\sigma_k, \sigma_i)X = \sigma_i\sigma_k X$$

and

$$B = \sigma_k\sigma_j\sigma_k Y, \quad W = \sigma_j\sigma_k\sigma_j Y.$$

The goal of the next series of steps in the proof is to get an element C' such that $A = A'C'$ and $B = B'C'$ by using H1 repeatedly. We shall see that through the process, both A' and B' will be words containing only σ_i, σ_j and σ_k. The final step will be transforming A' and B' in the right form for the theorem. As a reference for the reader, we have illustrated the steps in Figure 3.1.

We have that $W = \sigma_i\sigma_k X = \sigma_j\sigma_k\sigma_j Y$. We remark that since $\ell(W) = \ell(p) - 1$, we can use H1 on it as well as on all the next series of equalities. We thus apply H1 to the last equality to obtain that there exist Z_1 such that

$$\sigma_k X = \sigma_j Z_1, \quad \sigma_k\sigma_j Y = \sigma_i Z_1$$

Our first goal is to obtain expressions for X and Y. We apply H1 to $\sigma_k X = \sigma_j Z_1$ to obtain that, for some Z_2,

$$X = \sigma_j\sigma_k\sigma_j Z_2, \quad Z_1 = \sigma_k\sigma_j\sigma_k Z_2$$

and applying H1 to $\sigma_k(\sigma_j Y) = \sigma_i Z_1$ yields that, for some Z_3,

$$\sigma_j Y = \sigma_i\sigma_k Z_3, \quad Z_1 = \sigma_k\sigma_i Z_3.$$

We now have an expression for X, so we apply H1 to $\sigma_j Y = \sigma_i \sigma_k Z_3$, to get an expression for Y. We get that, for some Z_4,

$$Y = \sigma_i Z_4, \quad \sigma_k Z_3 = \sigma_j Z_4.$$

After these few steps, we have the following equalities:

$$\begin{aligned} X &= \sigma_j \sigma_k \sigma_j Z_2, & Y &= \sigma_i Z_4, \\ Z_1 &= \sigma_k \sigma_j \sigma_k Z_2 = \sigma_k \sigma_i Z_3, & \sigma_k Z_3 &= \sigma_j Z_4. \end{aligned}$$

We then look for an expression for both Z_3 and Z_4. By using H1 on $\sigma_k Z_3 = \sigma_j Z_4$, we get that, for some Z_5,

$$Z_3 = \sigma_j \sigma_k \sigma_j Z_5, \quad Z_4 = \sigma_k \sigma_j \sigma_k Z_5.$$

By replacing in previous set of equations the values of Z_3 and Z_4, we now have

$$\begin{aligned} X &= \sigma_j \sigma_k \sigma_j Z_2, & Y &= \sigma_i \sigma_k \sigma_j \sigma_k Z_5, \\ Z_1 &= \sigma_k \sigma_j \sigma_k Z_2 = \sigma_k \sigma_i \sigma_j \sigma_k \sigma_j Z_5. \end{aligned}$$

The next step is to find expressions for Z_2 and Z_5 ending with the right divisor Z_i. Since $\sigma_i \sigma_j = \sigma_j \sigma_i$, we have that

$$Z_1 = \sigma_k \sigma_i \sigma_j \sigma_k \sigma_j Z_5 = \sigma_k \sigma_j \sigma_i \sigma_k \sigma_j Z_5.$$

Applying H1 twice to $\sigma_k \sigma_j \sigma_k Z_2 = \sigma_k \sigma_j \sigma_i \sigma_k \sigma_j Z_5$ yields

$$\sigma_k Z_2 = \sigma_i \sigma_k \sigma_j Z_5.$$

Using H1 again on this equation, we get that for some Z_6,

$$Z_2 = \sigma_i \sigma_k Z_6, \quad \sigma_k \sigma_j Z_5 = \sigma_k \sigma_i Z_6.$$

Using H1 twice on the second equation first yields $\sigma_j Z_5 = \sigma_i Z_6$ and then yields that for some Z_7,

$$Z_5 = \sigma_i Z_7, \quad Z_6 = \sigma_j Z_7.$$

We now have that

$$Z_2 = \sigma_i \sigma_k (\sigma_j Z_7),$$
$$Z_5 = \sigma_i Z_7.$$

By substituting these values in the expressions for X and Y, we get that

$$X = \sigma_j \sigma_k \sigma_j \sigma_i \sigma_k \sigma_j Z_7,$$
$$Y = \sigma_i \sigma_k \sigma_j \sigma_k \sigma_i Z_7$$

and finally that

$$A = \sigma_k \sigma_i \sigma_j \sigma_k \sigma_j \sigma_i \sigma_k \sigma_j Z_7,$$
$$B = \sigma_k \sigma_j \sigma_k \sigma_i \sigma_k \sigma_j \sigma_k \sigma_i Z_7.$$

Since these repeated applications of H1 and H2 can get confusing, all the previous steps can be found in Figure 3.1. In this diagram, standard arrows represent the application of either H1 or H2, dotted arrows represent that some transformation had been made to the equations, while the "barred" arrows ($\mapsto$) represent replacement of a value.

Recall that our goal is to show that there exists some $C \in \mathcal{A}^+$ such that $A = \sigma_j C$ and $B = \sigma_i C$. Thus, using a sequence of applications of the defining relations of the group, which we omit, we get that

$$\begin{aligned} A &= \sigma_k \sigma_i \sigma_j \sigma_k \sigma_j \sigma_i \sigma_k \sigma_j Z_7 \\ &= \sigma_j (\sigma_k \sigma_j \sigma_k \sigma_i \sigma_k \sigma_j \sigma_k Z_7) \\ &= \sigma_j C \end{aligned}$$

and

$$\begin{aligned} B &= \sigma_k \sigma_j \sigma_k \sigma_i \sigma_k \sigma_j \sigma_k \sigma_i Z_7 \\ &= \sigma_i (\sigma_k \sigma_i \sigma_j \sigma_k \sigma_j \sigma_i \sigma_k Z_7) \\ &= \sigma_1 (\sigma_k \sigma_i \sigma_j \sigma_k \sigma_j \sigma_i \sigma_k Z_7) \\ &= \sigma_i C, \end{aligned}$$

where

$$C = \sigma_k \sigma_j \sigma_k \sigma_i \sigma_k \sigma_j \sigma_k Z_7.$$

We now have that $\sigma_i A = {}_{m_{i,j}}(\sigma_i, \sigma_j)C = {}_{m_{j,i}}(\sigma_j, \sigma_i)C = \sigma_j B$. It follows that $A = \sigma_j C_1 = \sigma_i^{-1}{}_{m_{i,j}}(\sigma_i, \sigma_j)C_1$ and $B = \sigma_i C_1 = \sigma_j^{-1}{}_{m_{j,i}}(\sigma_j, \sigma_i)C_1$ as required.

The process is similar for all the other combinations of values for $m(i,j)$, $m(i,k)$ and $m(j,k)$. These cases cover both the induction on the length r and on the chain-length t, thus proving the lemma. ∎

Theorem 3.2.17. *Every element $p \in \mathcal{A}^+$ has a unique left-weighted factorisation, say $p = a_1 p_1$, with $a_1 \in \mathcal{A}^{Red}$. Moreover, if $p = a'p'$ with $a' \in \mathcal{A}^{Red}$ is another positive factorisation, then $a' \leq a_1$.*

Proof: Let p be a positive element. We will begin by showing the existence of a left-weighted factorisation $p = a_1 p_1$ with $a_1 \in \mathcal{A}^{Red}$.

First, consider all $a \in \mathcal{A}^{Red}$ such that $a \leq_L p$ and pick a with $\ell(a)$ maximal and write $p = ab$. If $S(b) \subset F(a)$, then $p = ab$ is left-weighted. If $S(b) \not\subset F(a)$, then there exists $i \in S(b)$ such that $i \notin F(a)$. We then have $b = \sigma_i b'$ for some $b' \in \mathcal{A}^+$ and we can write $p = ab = a\sigma_i b'$. By Proposition 3.2.10 applied to the finishing set, $a' = a\sigma_i$ is a reduced braid and it follows that $p = a'b'$ is a factorisation with $\ell(a') > \ell(a)$. This contradicts the maximality of $\ell(a)$, and thus implies that $S(b) \subset F(a)$ and that $p = ab$ is left-weighted. We shall write $p = a_1 p_1$ for the chosen factorisation $p = ab$.

We have shown that a positive factorisation $p = a_1 p_1$, where $a_1 \in \mathcal{A}^{Red}$ and $\ell(a_1)$ is maximal, is left-weighted. It remains to show that this factorisation is unique. We will do this by showing that every other positive factorisation $p = ab$ with $a \in \mathcal{A}^{Red}$ satisfies $a_1 = aq$, for some positive element q. If $a_1 = \Delta$, the result is immediate since all reduced elements are a left divisor of Δ.

If $a_1 \neq \Delta$, we have that there exists some, possibly trivial, $c \leq_L a_1$ for which we can find an σ_i such that $c\sigma_i \in \mathcal{A}^{Red}$ and $c\sigma_i \nleq_L a_1$. Namely, given c, we can choose σ_i such that $i \in F(c)$ and $i \notin S(c^{-1}a_1)$.

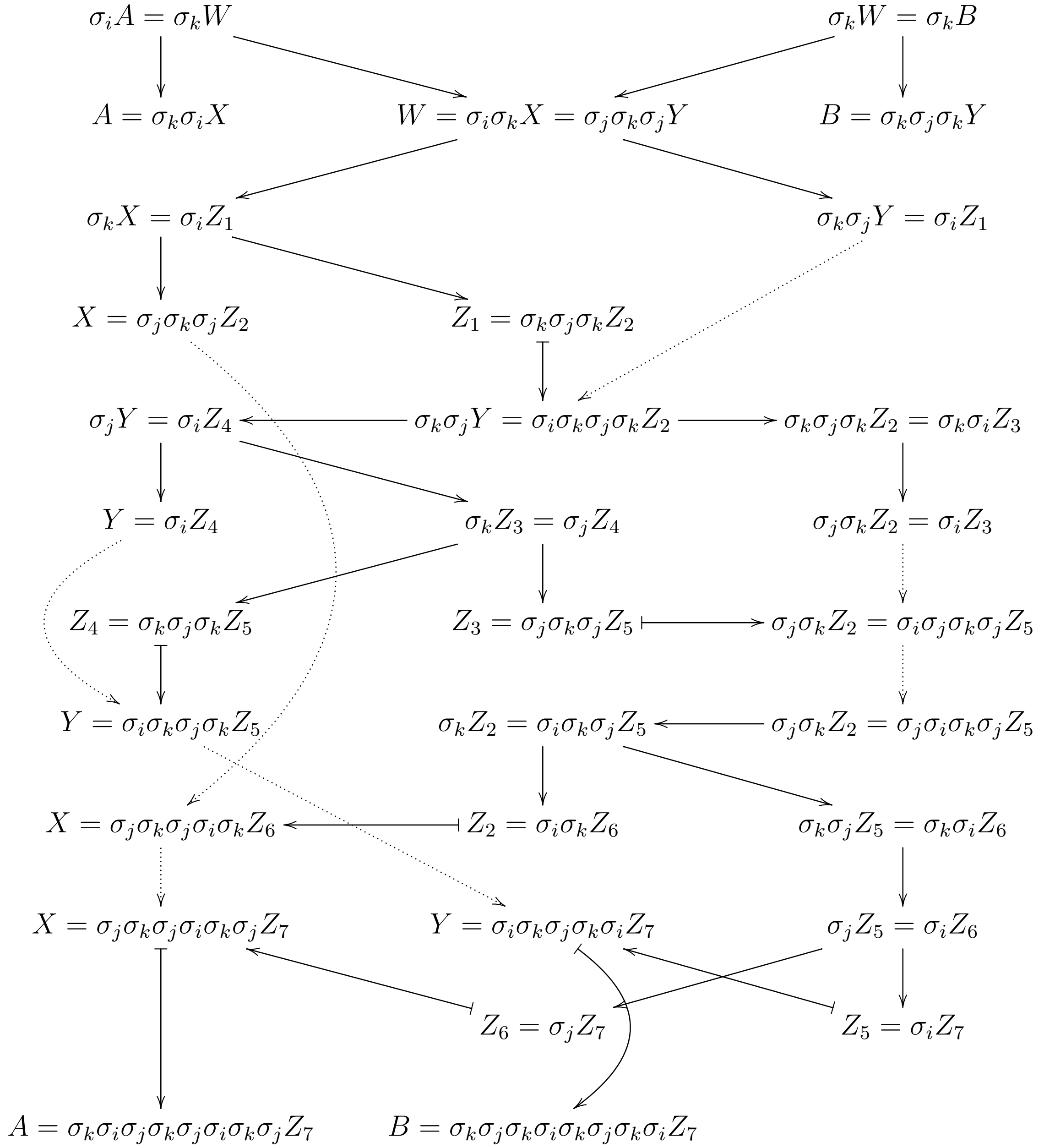


Figure 3.1: Flowchart of the proof of Lemma 3.2.15

Suppose to the contrary that for $a_1 \neq \Delta$, there exist factorisations of the form $p = c\sigma_i b_i$, where $b_i \in \mathcal{A}^+$, $c\sigma_i \in \mathcal{A}^{Red}$, $c \leq_L a_1$ and $c\sigma_i \not\leq_L a_1$. Choose such a factorisation with $\ell(c)$ maximal and, by the first part of this proof, $a_1 = cq$ is a left-

weighted factorisation. Now, since $\ell(a_1)$ is maximal, we have that $\ell(a_1) \geq \ell(c\sigma_i) > \ell(c)$. It follows that $q \neq 1$.

Since $a_1 = cq$ and $q \neq 1$, there exists $j \in S(q)$ such that $c\sigma_j \leq_L a_1$. It follows, by Proposition 2.2.4, that $c\sigma_j$ is reduced. We can now write the positive factorisation $p = c\sigma_j b_j$. Because $\mathcal{A}^+$ is left-cancellative and $p = c\sigma_i b_i = c\sigma_j b_j$, we get that $\sigma_i b_i = \sigma_j b_j$. It then follows, from Lemma 3.2.15, that $\sigma_i b_i = \sigma_j b_j = {}_{m_{i,j}}(\sigma_i, \sigma_j) b_{i,j}$ for some $b_{i,j} \in \mathcal{A}^+$.

We now have that $p = c\,{}_{m_{i,j}}(\sigma_i, \sigma_j) b_{i,j}$ is a positive factorisation. An important remark to make at this point is that $i \neq j$, since $c\sigma_j \leq_L a_1$ and $c\sigma_i \not\leq_L a_1$. We have that $m_{i,j} > 1$ and thus that $p = c\sigma_j\sigma_i q'$, for some $q' \in \mathcal{A}^+$, is a positive factorisation whith $c\sigma_j \leq_L a_1$ and $c\sigma_j\sigma_i \not\leq_L a_1$. But this contradicts the maximality of $\ell(c)$, since $\ell(c\sigma_j) > \ell(c)$.

It then follows that a left-weighted factorisation $p = a_1 p_1$ for p is unique. Indeed, if $p = uv$ is another left-weighted factorisation, then we can write $a_1 = uq$ for some $q \in \mathcal{A}^+$. Then either $q = 1$, giving $a_1 = u$ as needed, or $q \neq 1$. In the latter case, since $a_1 = uq$ and $q \neq 1$, there exists $i \in S(q)$ such that $us_i \leq_L a_1$. As such, us_i is reduced and it follows that $i \notin F(u)$. But $v = qp_1$, so $i \in S(v)$ and it follows that $S(v) \not\subset F(u)$, proving that $p = uv$ is not left-weighted. ∎

Definition 3.2.18. If $p = a_1 p_1$ is a left-weighted factorisation for a positive element p, we say that a_1 is the *maximal head* for p.

Corollary 3.2.19. *Let $p \in \mathcal{A}^+$ and let $p = a_1 p_1$ be the left-weighted factorisation with $a_1 \in \mathcal{A}^{Red}$. Then, $S(a_1) = S(p)$.*

Proof: Since $p = a_1 p_1$ is a factorisation, we have $S(a_1) \subset S(p)$.

Now suppose $i \in S(p)$. Then $\sigma_i \leq_L p$ and, by the second part of Theorem 3.2.17, we have that $\sigma_i \leq_L a_1$. Thus $i \in S(a_1)$ and we have the desired equality. ∎

With the two previous results, we obtain the existence of the *left-weighted normal form.*

Theorem 3.2.20. *For any $p \in \mathcal{A}^+$, there is a unique expression $p = a_1 a_2 \cdots a_k$ with $a_i \in \mathcal{A}^{Red}$, $a_k \neq 1$ and $S(a_{i+1}) \subset F(a_i)$ for each i. This expression is called the* left-weighted normal form *of p.*

Proof: Let $p_0 = p$ and let $a_i p_i$, $i \geq 1$, be the unique left-weighted factorisation for p_{i-1}. By induction on i we get the desired expression. The uniqueness of the expression follows because at each step, the left-weighted factorisation of p_{i-1} is unique and independent of the choice of words representing p. This gives us that all a_i's are uniquely determined by p. ∎

The algorithm to obtain the left-weighted normal form is then to repeatedly take the left-weighted factorisation of every p_i. One way of doing this is to first write $p = b_1 b_2 \cdots b_k$, where for each i, $b_i \in \mathcal{A}^{Red}$ and $b_i \neq 1$. Then for each pair b_i, b_{i+1} we verify if $b_i b_{i+1}$ is left-weighted. If it is, we check the pair b_{i+1}, b_{i+2} while if it is not, we find a left-weighted factorisation $c_i c_{i+1}$ for $b_i b_{i+1}$ and replace it in p. We keep doing this until all pairs are left-weighted.

From the way the algorithm for the normal form works, if at some point $b_i b_{i+1}$ is replaced by Δc_{i+1}, then the maximal head of $b_1 \cdots b_k$ will be Δ. We can then group all the Δ obtained this way together and write them as Δ^s. We then get a slightly more compact normal form.

Remark 3.2.21. Let p be a positive element of $\mathcal{A}$ and let $p = a_1 \cdots a_k$ be the left weighted normal form where $a_1 = \ldots = a_s = \Delta$ and either $s = k$ or $a_{s+1} \neq \Delta$. Then $p = \Delta^s a_{s+1} \cdots a_k$ is also a unique expression for p.

Now that we have the existence and uniqueness of a normal form for a positive element of $\mathcal{A}$, it follows that we have one for an arbitrary element by Lemma 3.2.1.

Proposition 3.2.22. *Let $p \in \mathcal{A}$, then there exists a unique normal form $p = \Delta^r a_1 \cdots a_k$ with $a_i \in \mathcal{A}^{Red}$ and where $S(a_{i+1}) \subset F(a_i)$ and $r \in \mathbb{Z}$.*

Proof: Using the process presented with Lemma 3.2.1, we obtain $p = \Delta^s p'$ where $s \leq 0$ and $p' \in \mathcal{A}^{Red}$. We then use Remark 3.2.21 on p' and we obtain the desired normal form. The uniqueness follows from Theorem 3.2.20. ∎

Remark 3.2.23. We use a more practical notation for the normal form when using the algorithms. Indeed, instead of writting $p = \Delta^r a_1 \cdots a_k$, we use instead use a sequence notation $p = (r, a_1, \ldots, a_k)$, where the a_i's are represented by $\rho(a_i)(1, 2, \ldots, n)$ in the Coxeter system (W, S).

Example 3.2.24. In the braid group $\mathcal{B}_4$, if the normal form of b is

$$b = \Delta^2(\sigma_1)(\sigma_1\sigma_2)(\sigma_2\sigma_3),$$

we will write $b = (2, (2, 1, 3, 4), (3, 1, 2, 4), (1, 4, 2, 3))$.

3.3 Algorithm for left-weighted normal form

In this section, we present the algorithms needed to compute the left-weighted normal form of a positive element of $\mathcal{A}(A_n)$, $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$. The main algorithm for $\mathcal{A}(A_n)$ comes from [CKL+01] and is based on the starting sets. We have used the algorithms provided for the braid groups and adapted them to the other Artin groups of finite type.

In this section we will assume, unless stated otherwise, that all the reduced elements, $a_w \in \mathcal{A}^{Red}$, are written as a vector $(w(1), w(2), \ldots, w(n)) = w(1, 2, \ldots, n)$.

That is, we represent a reduced element a_w by the value of the reflection w applied to $(1, 2, \cdots, n)$ in the associated Euclidean space.

Recall from Definition 1.7.1 that, given a partial order $\leq$ on a set, the *meet* of u and v, $u \wedge v$, is the maximum element, if it exists, of the set

$$\{x | x \leq u \text{ and } x \leq v\}.$$

We already proved (see Theorem 1.7.7) that any Coxeter system (W, S) has a semi-lattice structure for $\leq_L$ and $\leq_R$ as defined in Definition 3.1.1, that is, the meet of any two elements always exists. We denote the meet with regards to $\leq_L$ (respectively, $\leq_R$) by $\wedge_L$ (resp. $\wedge_R$). It then follows that the meet exists for $\mathcal{A}^{Red}$.

Our first algorithm (see Algorithm 1) computes the left-weighted normal form of an element $p' = \Delta^r b_1 b_2 \cdots b_k = \Delta^r p$, where the b_i's are reduced elements.

This algorithm is slightly different than the one we presented in the proof of Theorem 3.2.20 in that it is more efficient. The algorithm works as follows. From $p = b_1 b_2 \cdots b_k$, we first compute the maximal head of $b_{k-1} b_k$. Using this we can compute the maximal head of $b_{k-2} b_{k-1} b_k$ due to Corollary 3.2.19. We then proceed as such until we obtain a_1, the maximal head of $b_1 b_2 \cdots b_k$. The left weighted factorisation is then $p = a_1 p_1$. We then use the algorithm again on p_1. The algorithm ends when $p_{l+1} = 1$ and the left-weighted normal form is $p = a_1 a_2 \cdots a_l$.

The way we compute the maximal head of ab, with a and b reduced, is by computing $a^{-1}\Delta \wedge b$ (see Algorithm 1, line 6). The reason for this computation is that $c = a^{-1}\Delta \wedge b$ is the largest left divisor of b such that ac is still reduced. It is not hard to see that $S(c^{-1}b) \subset F(ac)$ and it follows that ac is the maximal head of ab by the unicity of the meet and the left-weighted factorisation.

In Algorithm 1, the operation on line 13 sets the value of i to the smallest value where b_i was changed by the algorithm. This is such that if $b_1 \cdots b_m$ is already left-weighted, then those factors will not be included in further iterations of the while loop. It also serves the purpose of only working on the part of the element that still

Algorithm 1 Algorithm for left-weighted normal form

Input: An element $p = \Delta^r b_1 b_2 \cdots b_k$, b_i's all reduced.
Output: Left-weighted normal form for p.

1: $l \leftarrow k$
2: $i \leftarrow 1$
3: **while** $i < k$ **do**
4: $\quad t \leftarrow l - 1$
5: $\quad$ **for** $j = l - 1$ to i **do**
6: $\quad\quad c \leftarrow b_j^{-1}\Delta \wedge_L b_{j+1}$
7: $\quad\quad$ **if** $C \neq 1$ **then**
8: $\quad\quad\quad t \leftarrow j$
9: $\quad\quad\quad b_j \leftarrow b_j c$
10: $\quad\quad\quad b_{j+1} \leftarrow c^{-1} b_{j+1}$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad i \leftarrow t + 1$
14: **end while**
15: **while** $l > 0$ and $a_1 = \Delta$ **do**
16: $\quad$ Remove a_1 from the expression for p
17: $\quad r \leftarrow r + 1$
18: $\quad l \leftarrow l - 1$
19: **end while**
20: **while** $l > 0$ and $a_l = 1$ **do**
21: $\quad$ Remove a_l from the expression for p
22: $\quad l \leftarrow l - 1$
23: **end while**

needs to be put into its left-weighted normal form.

As we can see, the algorithm itself is straightforward, but it assumes we know how to compute the meet of two reduced elements. This problem is more complex, because the meet is computed slightly differently depending on which Artin group we are working with.

Our algorithms for the meet are based on an algorithm given for the braid group by Thurston ([ECH+92]) and its implementation provided in [CKL+01]. His idea was to sort two permutations simultaneously using a merge sort to obtain the meet in the braid group. The algorithm was not transparent and the generalisation to $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$ was far from obvious; we thus devised our own algorithms (see Algorithm 2 in this section and Algorithms 6 and 7 in Appendix A) for the meet based on the starting sets, even for the braid group.

Our final algorithms also use a divide and conquer approach for increased efficiency, but the basic principle is as follows. Given two reduced elements, say a and b, we want to find their largest left common divisor that is also a positive element. Our first step is to compute their starting sets, $S(a)$ and $S(b)$, and look at their intersection. We saw in the proof of Theorem 1.7.7 that $S(a \wedge_L b) = S(a) \cap S(b)$.

We can then build a rooted directed graph with no loops using the starting sets in the following manner. Let the root be labeled by (a, b), then given a node labeled by (a', b'), its children will be the nodes labeled $(\sigma_i^{-1}a', \sigma_i^{-1}a')$, where $i \in S(a') \cap S(b')$. We can then label the edges between two nodes by the corresponding generator (in the previous case, the edge would be labeled "σ_i"). We can then build the full tree recursively, since both a and b are of finite length and thus the process ends when $S(a') \cap S(b') = \emptyset$. Once the graph is built, the meet is obtained by the element given by the labels of the edges traversed in one of the paths of maximal length starting from the root. Each one of these paths of longest length will give us the left meet of a and b by the unicity of the meet.

This method can make the graph extremely large, so it is not the most efficient

Algorithm 2 Recursive algorithm for left meet in $\mathcal{A}(A_{n-1})$

Input: $a = (a_s, \ldots, a_t)$, $b = (b_s, \ldots, b_t)$ sub vectors of reduced elements.
Output: The left meet $c = a \wedge_L b$.

1: $c \leftarrow (1, 2, \ldots, n)$ $\{c = 1 \text{ in } \mathcal{A}(A_{n-1})\}$
2: $m \leftarrow \left\lfloor \frac{s+t}{2} \right\rfloor$
3: Create empty queue $Q = \{\}$
4: Compute $c_1 = (a_s, \ldots, a_m) \wedge_L (b_s, \ldots, b_m)$
5: Compute $c_2 = (a_{m+1}, \ldots, a_t) \wedge_L (b_{m+1}, \ldots, b_t)$
6: $c \leftarrow c_1 c_2$
7: $a' \leftarrow c^{-1} a$
8: $b' \leftarrow c^{-1} b$
9: **if** $a'(m+1) < a'(m)$ and $b'(m+1) < b'(m)$ **then**
10: Enqueue σ_m in Q
11: **end if**
12: **while** Q is not empty **do**
13: Remove σ_i from the start of Q
14: $c \leftarrow c\sigma_i$
15: $a' \leftarrow \sigma_i^{-1} a'$
16: $b' \leftarrow \sigma_i^{-1} b'$
17: **if** $i > s$ and $a'(i) < a'(i-1)$ and $b'(i) < b'(i-1)$ **then**
18: Enqueue σ_{i-1} in Q {Has to be $i > 2$ for type D_n}
19: **end if**
20: **if** $i + 2 \leq t$) and $a'(i+2) < a'(i+1)$ and $b'(i+2) < b'(i+1)$ **then**
21: Enqueue σ_{i+1} in Q
22: **end if**
23: **end while**
24: **return** The left meet c.

way to proceed. An improvement would be to allow edges to be labeled by a product of generators, so that we can remove redundant nodes. For example, if $\{2,5\} \subset S(A) \cap S(B)$, we could add a single node with the edge $\sigma_2\sigma_5$ to replace the two paths labeled by $\sigma_2 - \sigma_5$ and $\sigma_5 - \sigma_2$, since σ_2 and σ_5 commute. By doing this and giving weight 2 to this edge, we greatly reduce the number of paths to check after the construction of the graph. In Section 3.4, we will argue that using a divide and conquer approach is equivalent to constructing a graph having a unique path, leaving us with an extremely efficient algorithm to find the meet.

3.4 Proofs of correctness for the meet algorithms

The goal of this section is to prove that the divide-and-conquer method used in our meet algorithms gives the right element. This will be done by showing that the directed graph constructed by is algorithm is a path. We start by generalizing the idea of *sorting* to multiple vectors.

Definition 3.4.1. Given a finite reflection group, (W,S), acting on $\mathbb{R}^n$ and k vectors, $a_i = (a_{i,1},\ldots,a_{i,n})$, $1 < i < k$, representing elements of W, we say that $X = (a_1,\ldots,a_k)$ is *sorted with regards to* (W,S) if $S(X) = S(a_1) \cap \ldots \cap S(a_k) = \emptyset$.

With our choice of generators for the finite reflection groups, it is clear that for $k = 1$, $v \in \mathbb{R}^n$ is sorted whenever $0 < v_1 < v_2 < \cdots < v_n$, that is whenever v is in the fundamental domain of W. We will be using this definition for the case where $k = 2$, since the meet algorithms require us to sort two vectors in the sense of Definition 3.4.1.

We note that from the definition of starting sets in W, we have that $s_i \in S(w)$ if and only if $s_i \notin S(s_iw)$. As such, given $X = (a_1,\ldots,a_k)$, if $i \in S(X)$ (respectively $i \notin S(X)$), we say that $S(s_i(X))$ is *more* (respectively *less*) *sorted*. We say that a $w \in W$ *sorts* X if $S(w(X)) = \emptyset$.

Definition 3.4.2. Let $X = (a_1, \ldots, a_k)$, with $a_i = w'(1, 2, \ldots, n)$ for some $w_i \in W$. We will write $X = (x_1, \ldots, x_n)$ with $x_i = (a_{i,1}, \ldots, a_{i,k})$ to consider X as a sequence to sort with regards to (W, S).

Remark 3.4.3. Given X as in Definition 3.4.2 and Propositions 3.2.7, 3.2.8 and 3.2.9 we obtain some characterisation for $S(X)$.

If W is of type A_{n-1}, B_n or D_n, we have

$$i \in S(X) \text{ if and only if } a_{j,i} > a_{j,i+1} \text{ for all } j.$$

If W is of type B_n, we have that

$$0 \in S(X) \text{ if and only if } a_{j,1} < 0 \text{ for all } j.$$

Finally, if W is of type D_n, we have that

$$0 \in S(X) \text{ if and only if } a_{j,1} + a_{j,2} < 0 \text{ for all } j.$$

From this point on, we assume that X is as in Definition 3.4.2, that is $X = (x_1, \ldots, x_n)$ where $x_i = (a_{1,i}, \ldots, a_{k,i})$ with $a_i = w_i(1, \ldots, n)$ for some $w_i \in W$.

We begin by an important lemma that will let us use a divide-and-conquer approach in our algorithm and still guarantee the uniqueness of the meet.

Lemma 3.4.4. *Let (W, S) be a finite reflection group and let $X = (x_1, \ldots, x_n)$. If $w_1 \in \langle s_0, \ldots, s_{\lfloor \frac{n}{2} \rfloor - 1} \rangle$ is the unique element that sorts $(x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor})$ and if $w_2 \in \langle s_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, s_n \rangle$ is the unique element that sorts $(x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n)$, then $w_1 w_2$ is the unique element such that $w_1 w_2 X$ has both halves sorted.*

Moreover, $S(w_1 w_2 X) \subset \{\lfloor \frac{n}{2} \rfloor\}$.

Proof: If w_1 and w_2 are unique, then we have that $w_1 w_2 = w_2 w_1$ and it follows that this element is unique. Indeed, w_1 is a word in $s_0, \ldots, s_{\lfloor \frac{n}{2} \rfloor - 1}$ while w_2 is a word in $s_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, s_n$. We thus recognize from the definitions that in each cases,

the generators used to write w_1 commute with those used to write w_2, making w_1w_2 unique.

Since both halves of w_1w_2X are sorted, then none of the generators

$$s_0, \dots, s_{\lfloor \frac{n}{2} \rfloor -1}, s_{\lfloor \frac{n}{2} \rfloor +1}, \dots, s_n$$

are in $S(w_1w_2X)$. As such, we have that $S(w_1w_2X) \subset \{\lfloor \frac{n}{2} \rfloor\}$.

We can then verify if $S(w_1w_2X)$ is $\{\lfloor \frac{n}{2} \rfloor\}$ or empty by computing $S(w_1w_2X)$ with Remark 3.4.3. ∎

We now introduce two subsets of I_A, the set of labels on the generators. These set will prove to be crucial for our proofs of correctness.

Definition 3.4.5. Define the sets EVEN and ODD of indices in the following fashion:

$$\text{EVEN} = \begin{cases} \{2k \mid 1 \leq 2k < n\} & \text{if } W \text{ is of type } A_{n-1} \\ \{2k \mid 0 \leq 2k < n\} & \text{if } W \text{ is of type } B_n \\ \{2k \mid 2 \leq 2k < n\} & \text{if } W \text{ is of type } D_n \end{cases}$$

and

$$\text{ODD} = \begin{cases} \{2k+1 \mid 1 \leq 2k+1 < n\} & \text{if } W \text{ is of type } A_{n-1} \\ \{2k+1 \mid 1 \leq 2k+1 < n\} & \text{if } W \text{ is of type } B_n \\ \{2k+1 \mid 1 \leq 2k+1 < n\} \cup \{0\} & \text{if } W \text{ is of type } D_n \end{cases}$$

We note that $\text{EVEN} \cup \text{ODD} = I_A$, by construction. The next lemma follows directly from the construction of the sets EVEN and ODD and by the defining relations of the Coxeter groups of type A_n, B_n and D_n.

Lemma 3.4.6. *For all $s_i, s_j \in W$, we have*

$$s_is_j = s_js_i \textit{ if } \{i,j\} \subset \text{EVEN} \textit{ or } \{i,j\} \subset \text{ODD}.$$

Proposition 3.4.7. *Let $X = (x_1, \ldots, x_n)$ be as in Definition 3.4.2 and define $\overline{w} = \prod_{i \in S(X)} s_i$. Then $\overline{w}$ is a reduced expression and:*

If $S(X) \subset \mathrm{EVEN}$, then $S(\overline{w}X) \subset \mathrm{ODD}$.

Similarly, if $S(X) \subset \mathrm{ODD}$, then $S(\overline{w}X) \subset \mathrm{EVEN}$.

Proof: Suppose $S(X) \subset \mathrm{EVEN}$. Then Lemma 3.4.6 gives us that all generators used to write $\overline{w}$ commute with one another. As such, $\overline{w}$ is reduced.

Since $S(X) \subset \mathrm{EVEN}$, it is clear that $S(\overline{w}X) \cap \mathrm{EVEN} = \emptyset$, by the choice of $\overline{w}$. As such, $S(\overline{w}X) \subset I_{\mathcal{A}} \backslash \mathrm{EVEN} = \mathrm{ODD}$.

The proof is identical if $S(X) \subset \mathrm{ODD}$. ∎

With these results, we are now able to prove that the directed graph built using a merge sort is a single path in each of the three Artin groups $\mathcal{A}(A_{n-1})$, $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$.

Proposition 3.4.8. *Let (W, S) be the finite reflection group of type A_{n-1}, B_n or D_n. Let $k = \left\lfloor \frac{n}{2} \right\rfloor$ and let $X = (x_1, \ldots, x_n) = (a, b)$ be such that both halves of X are sorted. That is*

$$S(X) \subset \{k\}.$$

Then the element w sorting X is obtained uniquely from the algorithm.

Proof: If $S(X) = \emptyset$, we are done and the element sorting X is $w = 1$. If $S(X) = \{k\}$, we will suppose, without loss of generality, that $\{k\} \subset \mathrm{EVEN}$. Put $\overline{w_1} = \prod_{i \in S(X)} s_i$, then $\overline{w_1}X$ is more sorted and

$$S(\overline{w_1}X) \subset \{k-1, k+1\} \subset \mathrm{ODD}.$$

This follows from the fact that all the other generators commute with s_k and from Proposition 3.4.7. We put $\overline{w_2} = \prod_{i \in S(\overline{w_1}X)} s_i$ and we have $S(\overline{w_2 w_1}X) \subset \{k-2, k, k+2\} \subset \mathrm{EVEN}$.

The algorithm continues this process by setting $\overline{w_k} = \prod_{i \in S(\overline{w_{k-1}} \cdots \overline{w_1} X)} s_i$ at every step. We also notice that $S(\overline{w_k} \cdots \overline{w_1} X) \in$ EVEN if and only if $S(\overline{w_{k-1}} \cdots \overline{w_1} X) \in$ ODD, and vice versa.

We claim that the process stops when $S(\overline{w_k} \cdots \overline{w_1} X) = \emptyset$ and the meet is $w = \overline{w_1} \cdots \overline{w_k}$. This follows by the nature of the starting sets, which ensures that if $\overline{w} = \prod_{i \in S(a)} s_i$, then $w \leq_L a$, and from the commutativity of the s_i for any $i \in S(\overline{w_j} \cdots \overline{w_1} X)$.

Consequently, for every j we have, by construction, that $\overline{w_j} \leq_L a'$ and $\overline{w_j} \leq_L b'$, where $a' = \overline{w_{j-1}} \cdots \overline{w_1} a$ and $b' = \overline{w_{j-1}} \cdots \overline{w_1} b$. As such, $\overline{w_j} \leq_L a' \wedge b'$ by definition of the meet. We note that for every j, $\overline{w_j}^{-1} = \overline{w_j}$ since the generators used to write w_j commute. It follows that $\overline{w_1} \cdots \overline{w_k} \leq_L a \wedge_L b$.

Let $w = a \wedge_L b$. We have $\overline{w_1} \leq_L w$ with either $\overline{w_1} = 1$, which stops the process, or $\lambda(\overline{w_1}^{-1} w) < \lambda(w)$. In the latter case, we have

$$\overline{w_1} w = \overline{w_1}^{-1} w = \overline{w_1}^{-1} a \wedge_L \overline{w_1}^{-1} b = \overline{w_1} a \wedge_L \overline{w_1} b.$$

Repeating this process inductvely, we see that the process must terminate, since the meet has shorter length at each iteration. This process thus stops exactly when $\overline{w_{k+1}} = 1$, that is when $S(\overline{w_k} \cdots \overline{w_1} X) = \emptyset$. This happens exactly when $\overline{w_k} \cdots \overline{w_1} a \wedge_L \overline{w_k} \cdots \overline{w_1} b = 1$. The meet is then $\overline{w_1} \cdots \overline{w_k} = a \wedge_L b$. The uniqueness of w follows because each $\overline{w_k}$ is uniquely determined. ∎

Proposition 3.4.8 gives us that the divide-and-conquer procedure does produce a unique element w when merging. By recursion, the final directed graph built by the algorithm has only one path. Thus, the algorithm does give the meet of two elements by using a merge sort.

Chapter 4

Normal Form in Artin Groups of Large Type

In this chapter, we present the normal form for Artin groups of large type as given in [HR11]. This normal form is a word representing w that is geodesic and is minimal with regard to some arbitrary dictionary ordering on the generating set. We first present, in Section 4.1, the preliminary results necessary to imply the correctness of the algorithm in Artin groups with two generators. We then provide the general results for any Artin group of large type in Section 4.2. Finally, in Section 4.3, we extend the original paper by giving an explicit pseudocode algorithm to obtain the normal form of any given word w over an Artin group of large type. A Java implementation is included in Appendix B.

In this chapter we will write $w = v$ if w and v are equal as *words*, while we will write $w =_G v$ if w and v represent the same element in the Artin group.

4.1 Definitions and the dihedral case

In this section, we provide the basic definitions used throughout the chapter and we present the normal form in the 2-generator case. Let us start by formulating ideas on words used in the previous chapters.

Definition 4.1.1. An *alphabet* is a finite set A, while an element $a \in A$ is called a *letter*. A *word* over A is a finite sequence of letters.

Formally, a word is a map $\omega : \{1, \ldots, n\} \to A$ where $\omega(i)$ is the i-th letter of the word. The *length* of a word ω is the integer n and is denoted $|\omega|$. When $n = 0$, we say that ω is the *empty word* over A, and it is denoted by ϵ_A.

We denote by A^* the set of all words over the alphabet A. Together with concatenation, A^* is the free monoid over A.

Definition 4.1.2. Given a word $w = pus$, where $p, u, s \in A^*$ are possibly trivial, the word p is said to be a *prefix* of w, s is a *suffix* of w and u is a *subword* of w.

Let $\mathrm{f}[w]$ and $\mathrm{l}[w]$ be the first and last letter of w respectively and let $\mathrm{pre}[w]$ and $\mathrm{suf}[w]$ be, repectively, the longest proper prefix and suffix of w. We thus have $w = \mathrm{f}[w]\mathrm{suf}[w] = \mathrm{pre}[w]\mathrm{l}[w]$.

We now introduce some notation that will be used frequently troughout the chapter. Let $\mathcal{A}_n$ be an Artin group on n generators, with generating set $S' = \{\sigma_1, \ldots, \sigma_n\}$. We recall that an Artin group is said to be of large type whenever $m(i, j) \geq 3$, possibly infinity, for all $i \neq j$. We will denote the *dihedral* Artin group $\mathcal{A}(I_2(m))$ by $\mathcal{A}_2(m)$. We note that if $m < \infty$, then $\mathcal{A}_2(m)$ is an Artin group of finite type.

The alphabet over $\mathcal{A}_n$ that we will use is the set $A = S' \cup S'^{-1}$. A letter $a \in A$ is *positive* if $a \in S'$ and is *negative* otherwise. The *name* of a letter is its positive form, that is, the letters σ_i and σ_i^{-1} have the name σ_i.

Definition 4.1.3. A word $w \in A^*$ is said to be *positive* if all its letters are positive, *negative* if all its letters are negative and *unsigned* otherwise.

We note that this definition of positive and negative is consistent with Definition 2.1.4 given earlier.

Definition 4.1.4. A word $w \in A^*$ is said to be *freely reduced* if it does not admit any subwords of the form aa^{-1} or $a^{-1}a$ for any letter a. We say that a word which is not freely reduced *admits a free reduction.*

A word $w \in A^*$ is said to be a *geodesic* word if for any other word $v \in A^*$ such that $w =_G v$, we have that $|w| \leq |v|$.

Let x, y be two distinct letters and $k \in \mathbb{N}$. We denote by ${}_k(x, y)$ the alternating word of length k starting with x and we denote by $(x, y)_k$ the one ending with y. For example, ${}_3(x, y) = xyx$ and $(x, y)_4 = xyxy$. This is a generalization of the notation used earlier in Definition 1.4.4.

The normal form we obtain is based on the shortlex total ordering, which we now define. We first fix an arbitrary total lexicographic order on the alphabet A and we extend it to a total lexicographic order on A^*, which we will denote by $<_{lex}$.

Definition 4.1.5. Let A be an alphabet. For an arbitrary lexicographic ordering on A, $\leq_{lex}$, the *shortlex* ordering on A^* is defined by

$$w <_{slex} v \text{ if and only if } |w| < |v| \text{ or } (|w| = |v| \text{ and } w <_{lex} v).$$

Definition 4.1.6. A word w is said to be a *shortlex minimal representative* if for all words v such that $w =_G v$, $w \leq_{slex} v$.

Note that a shortlex minimal representative is also a geodesic word. Our normal form consists of shortlex minimal represatative. The algorithm for the shortlex normal form in Artin groups of large type is dependent on the shortlex minimal representatives in the dihedral Artin groups $\mathcal{A}_2(m)$. As such, we start by studying the shortlex minimal words in these groups. If $m = \infty$, $\mathcal{A}_2(\infty)$ is the free group on two variables and it is easy to see that every freely reduced word $w \in \mathcal{A}_2(\infty)$ is shortlex

minimal. We will thus consider the cases where $m < \infty$. Recall that $\mathcal{A}_2(m)$ is the group on two generators, say a and b, with the relation ${}_m(a,b) = {}_m(b,a)$. Given a freely reduced word over $\mathcal{A}_2(m)$, we shall see that alternating subwords play a major role in characterizing geodesic words.

Definition 4.1.7. Let w be a freely reduced word in $\mathcal{A}_2(m)$ over the alphabet $A = \{a, b, a^{-1}, b^{-1}\}$. Consider the integers

$$r_1 = \max\{r \mid {}_r(a,b) \text{ or } {}_r(b,a) \text{ is a subword of } w\}$$

and

$$r_2 = \max\{r \mid {}_r(a^{-1},b^{-1}) \text{ or } {}_r(b^{-1},a^{-1}) \text{ is a subword of } w\}.$$

Let $p(w) = \min\{r_1, m\}$ and let $n(w) = \min\{r_2, m\}$.

The values $p(w)$ and $n(w)$ of a word in $\mathcal{A}_2(m)$ in fact characterize all the geodesics in the group as the next proposition, proven in [MM06, Proposition 4.3], shows.

Proposition 4.1.8. *Let $g \in \mathcal{A}_2(m)$ and let $w \in A^*$ be a freely reduced word representing g.*

1. *If $p(w) + n(w) < m$, then w is the unique geodesic word for g.*
2. *If $p(w) + n(w) = m$, then w is one of the geodesic words for g.*
3. *If $p(w) + n(w) > m$, then w is not geodesic. Furthermore, w has a subword w' such that $p(w') + n(w') = m$.*

Since the normal form consists of shortlex minimal words, which are geodesic, our normal forms will be a subset of all the words with $p(w) + n(w) \leq m$. When $p(w) + n(w) < m$, then w is the unique geodesic word for an element g and as such is the (unique) shortlex minimal representative for w.

In the case where $p(w) + n(w) = m$, the word w is not the only geodesic for g and as such it might not be the shortlex minimal representative of g.

Example 4.1.9. Consider the Artin group $\mathcal{A}_2(3)$ with alphabet $A = \{a, b, a^{-1}, b^{-1}\}$ and lexicographic order $a <_{lex} b <_{lex} a^{-1} <_{lex} b^{-1}$.

Consider the element $w = ab^{-1}$. We see that the word ab^{-1} is such that $p(ab^{-1}) + n(ab^{-1}) = 2 < 3$. By observation, it is clear that ab^{-1} is a shortlex minimal representative for w.

If we consider the element $w = b^{-1}ab$, then we have $p(b^{-1}ab) + n(b^{-1}ab) = 3$. It is geodesic but it is not unique, since $b^{-1}ab = aba^{-1}$. We see that aba^{-1} is another geodesic word representing w, and that $aba^{-1} <_{slex} b^{-1}ab$.

If we consider the element $w = abab^{-1}$, then we have that $p(abab^{-1})+n(abab^{-1}) = 4 > 3$. The proposition tells us that $abab^{-1}$ is not geodesic, and we observe that

$$(aba)b^{-1} =_G (bab)b^{-1} =_G ba.$$

We thus have that $abab^{-1}$ is not geodesic, since $|ba| < |abab^{-1}|$.

Definition 4.1.10. Let w be a freely reduced word in $\mathcal{A}_2(m)$. Let $\{x, y\} = \{z, t\} = \{a, b\}$ and put $p = p(w)$ and $n = n(w)$. The word w is called a *critical word* if $p + n = m$ and it has one of the following forms. In these forms, ξ represents some word in A^*.

If w is a positive word, then

$$w = \xi(x, y)_m \text{ or } w =_m (x, y)\xi,$$

where w has exactly the one alternating positive subword of length m.

If w is a negative word, then

$$w = \xi(x^{-1}, y^{-1})_m \text{ or } w =_m (x^{-1}, y^{-1})\xi,$$

where w has exactly the one alternating negative subword of length m.

If w is an unsigned word, *i.e.* when $p, n \neq 0$, then

$$w = {}_p(x, y)\xi(z^{-1}, t^{-1})_n \text{ or } w = {}_n(x^{-1}, y^{-1})\xi(z, t)_p.$$

We denote by T the set of all critical words.

Note that, by Proposition 4.1.8, all the critical words are geodesics. We can define a map δ on the set of critical words to obtain other critical words. Recall that the automorphism τ is defined as in Proposition 2.3.22, which is given by

$$\tau(x) = \begin{cases} x & \text{if } m \text{ is even,} \\ y & \text{if } m \text{ is odd,} \end{cases}$$

where $\{x, y\} = \{a, b\}$.

Definition 4.1.11. Define a map δ on the critical words as follows:

$$\begin{aligned}
(x, y)_m &\mapsto (y, x)_m \\
\xi(x, y)_m &\mapsto {}_m(t, z)\tau(\xi), \text{ where } z = f[\xi] \\
{}_m(x, y)\xi &\mapsto \tau(\xi)(z, t)_m, \text{ where } z = l[\xi] \\
(x^{-1}, y^{-1})_m &\mapsto (y^{-1}, x^{-1})_m \\
\xi(x^{-1}, y^{-1})_m &\mapsto {}_m(t^{-1}, z^{-1})\tau(\xi), \text{ where } z = f[\xi]^{-1} \\
{}_m(x^{-1}, y^{-1})\xi &\mapsto \tau(\xi)(z^{-1}, t^{-1})_m, \text{ where } z = l[\xi]^{-1} \\
{}_p(x, y)\eta(z^{-1}, t^{-1})_n &\mapsto {}_n(y^{-1}, x^{-1})\tau(\eta)(t, z)_p \\
{}_n(x^{-1}, y^{-1})\eta(z, t)_p &\mapsto {}_p(y, x)\tau(\eta)(t^{-1}, z^{-1})_n,
\end{aligned}$$

where ξ is a non-empty word and η can be empty.

Proposition 4.1.12. *Let w be a critical word.*

1. *The word $\delta(w)$ is also critical, $\delta(w) =_G w$ and $\delta(\delta(w)) = w$.*
2. *$p(\delta(w)) = p(w)$ and $n(\delta(w)) = n(w)$.*
3. *$\mathrm{f}[w]$ and $\mathrm{f}[\delta(w)]$ have different names. Similarly, $\mathrm{l}[w]$ and $\mathrm{l}[\delta(w)]$ have different names.*
4. *If w is positive or negative, $\mathrm{f}[w]$ and $\mathrm{f}[\delta(w)]$ have the same sign. Similarly, $\mathrm{l}[w]$ and $\mathrm{l}[\delta(w)]$ have the same sign*
5. *If w is unsigned, $\mathrm{f}[w]$ and $\mathrm{f}[\delta(w)]$ have different signs. Similarly, $\mathrm{l}[w]$ and $\mathrm{l}[\delta(w)]$ have different signs.*

Proof: The map δ uses a combination of the relation $(x, y)_m =_G (y, x)_m$, Propositions 2.3.10 and 2.3.22 to identify another word for the same group elements. This is done either by moving the fundamental element $\Delta =_G (x, y)_m$ through the word. In

the cases of unsigned words, we first multiply the critical words by $1 = \Delta^{-1}\Delta = \Delta\Delta^{-1}$ before using Propositions 2.3.10 moving $\Delta^{\pm 1}$ through the word.

The proof then follows from the definition of δ and of τ in $\mathcal{A}_2(m)$. ∎

This proposition gives us that if w is a critical word, then we either have $w <_{lex} \delta(w)$ or $\delta(w) <_{lex} w$ and it can be determined only looking at the first letter of w.

Definition 4.1.13. Let w be a word that admits a factorisation $w = w_1 w_2 w_3$ where w_2 is a critical word. If $w_1\delta(w_2)w_3$ admits a free reduction or if $w_1\delta(w_2)w_3 <_{lex} w_1 w_2 w_3$, we say that w admits a *critical reduction*.

The free reduction of the word $w_1\delta(w_2)w_3$ is called a *critical reduction of* w.

We call the application of the map δ, and subsequent free reductions, on a critical subword a δ*-move*.

Definition 4.1.14. We say that $w_1\delta(w_2)w_3$ is a *right* (resp. *left*) *length reduction* if $\mathrm{l}[\delta(w_2)] = \mathrm{f}[w_3]^{-1}$ (resp. $\mathrm{f}[\delta(w_2)] = \mathrm{l}[w_1]^{-1}$).

We say that $w_1\delta(w_2)w_3$ is a (*left*) *lex reduction* if $\mathrm{f}[\delta(w_2)] <_{lex} \mathrm{f}[w_2]$ and $w_1\delta(w_2)w_3$ is not a length reduction.

Example 4.1.15. Consider the group $\mathcal{A}_2(3)$ and consider the word $w = babaaba^{-1}b^{-1}$.

The transformation

$$w = baba(aba^{-1})b^{-1} =_G baba\,\delta(aba^{-1})b^{-1} = baba(b^{-1}a\underline{b})\underline{b^{-1}} =_G babab^{-1}a$$

is a right length reduction where $w_2 = aba^{-1}$ and the underlined letters cancel each other. The transformation

$$w = (bab)aaba^{-1}b^{-1} =_G \delta(bab)aaba^{-1}b^{-1} = abaaaba^{-1}b^{-1}$$

is a left lex reduction where $w_2 = bab$.

The next theorem, originally [HR11, Theorem 2.4], gives us our set of normal forms.

Theorem 4.1.16. *Let $\mathcal{R}$ be the set of all words that do not admit any right length reduction or left lex reduction. Then $\mathcal{R}$ is the set of shortlex minimal representatives of elements of $\mathcal{A}_2(m)$.*

4.2 The normal form

We now have a good grasp on the shortlex normal form in the dihedral Artin groups $\mathcal{A}_2(m)$ and we will use the results in these groups for the general algorithm. Let $\mathcal{A}_n$ be an Artin group of large type with n generators $\sigma_1, \ldots, \sigma_n$.

Given an arbitrary word $w \in A^*$, we can consider all the subwords of w involving only 2 generators, say σ_i and σ_j, as words in the corresponding dihedral group $\mathcal{A}_2(m_{i,j})$. If u is such a subword, we define $p(u)$ and $n(u)$ as before. Also, for each 2-generator subgroup $\mathcal{A}_2(m_{i,j})$ of $\mathcal{A}_n$, the maps δ and τ are defined as in Section 4.1. Although all of these maps are defined in specific dihedral Artin groups, we will always be using them on 2-generator subwords and, as such, there is no possible confusion as to the particular definition of the maps. We will denote by $T_{i,j}$ the set of critical words in the subgroup $\mathcal{A}_2(m_{i,j})$. We will also only denote by m the value $m_{i,j}$ when working in a 2-generator subgroup.

Consider a freely reduced word w over $\mathcal{A}_n$ with a factorisation $w = \alpha_1 u_1 \beta_1$, where $u_1 \in T_{i_1,j_1}$. It is possible that $\alpha_1 \delta(u_1) \beta_1$ has a factorisation $\alpha_2 u_2 \beta_2$ such that the subword $u_2 \in T_{i_2,j_2}$ has a subword in common with $\delta(u_1)$. When such a situation occurs, we say that u_2 *overlaps* $\delta(u_1)$.

Example 4.2.1. Consider the Artin group of large type, $\mathcal{A}_3$, where $m_{i,j} = 3$ for all pairs $i \neq j$. Let a, b, c denote the three generators of $\mathcal{A}$, for simplicity.

Let $w = \alpha abacb\beta$ be some word in $\mathcal{A}$. Then we can do the following δ-move:

$$w = \alpha(aba)cb\beta =_G \alpha\delta(aba)cb\beta = \alpha ba(bcb)\beta.$$

As we can see, after this δ-move, we have a new subword $u_2 = bcb$ that has exactly one letter in common with $u_1 = aba$. This situation is an example of a *right overlap* where the subword u_2 is critical and has exactly one generator in common with u_1.

The situation described in this example is central to the algorithm.

Definition 4.2.2. Let w be a freely reduced word over $\mathcal{A}_n$. Suppose that w has a factorisation $\alpha w_1 u w_2 \beta$, where $u \in T_{i,j}$.

If w_1 is a 2-generator subword on $\sigma_{i_1}, \sigma_{j_1}$ such that $|\{i_1, j_1\} \cap \{i, j\}| = 1$, the name of $\mathrm{l}[w_1]$ is not in $\{\sigma_i, \sigma_j\}$ and $w_1\mathrm{f}[\delta(u)] \in T_{i_1,j_1}$, then we say that we have a *critical left overlap.*

Similarly, if w_2 is a 2-generator subword on $\sigma_{i_2}, \sigma_{j_2}$ such that $|\{i_2, j_2\} \cap \{i, j\}| = 1$, the name of $\mathrm{f}[w_2]$ is not in $\{\sigma_i, \sigma_j\}$ and $\mathrm{l}[\delta(u)]w_2 \in T_{i_1,j_1}$, then we say that we have a *critical right overlap.*

Definition 4.2.3. Let $\alpha_1 u_1 \beta_1$ be a freely reduced word with u_1 a critical subword. Consider a sequence

$$\alpha_1 u_1 \beta_1,\ \alpha_1\delta(u_1)\beta_1 = \alpha_2 u_2 \beta_2,\ \alpha_2\delta(u_2)\beta_2 = \alpha_3 u_3 \beta_3,\ \ldots,\ \alpha_k\delta(u_k)\beta_k,$$

where all the u_i's are critical subwords. If at each step we have a left (resp. right) critical overlap, we say that this is a *leftward* (resp. *rightward*) *critical sequence.*

If a critical sequence is such that $\alpha_k\delta(u_k)\beta_k$ is not freely reduced, the sequence is called a *length reducing sequence.* If it is reduced and $\alpha_k\delta(u_k)\beta_k <_{lex} \alpha_1 u_1 \beta_1$, then it is a *lex reducing sequence.*

We note that the length reducing sequences take priority over the lex reducing sequences. As such if a given sequence is both length reducing and lex reducing, we say it is a length reducing sequence.

Definition 4.2.4. A *right length reducing sequence* (RRS) is a rightward critical sequence that is length reducing.

A *left lex reducing sequence* (LLS) is a leftward critical sequence that is lex reducing.

We recall that in the Artin groups of dihedral type, the shortlex minimal representatives were the words that did not admit any left lex or right length critical reductions. We generalize this by looking at the words that do not admit any left lex reducing sequences (LLS) or any right length reducing sequences (RRS). We first give a few results on both the RRS and the LLS before presenting the normal form.

Right length reducing sequences (RRS)

We begin with the rightward critical sequences. The next proposition follows directly from the definition of a rightward critical sequence.

Proposition 4.2.5. *If a word w admits a rightward critical sequence, then w has a factorisation $w = \alpha w_1 w_2 \cdots w_k \beta$ satisfying the following properties.*

1. *For each $1 \leq i \leq k$, w_i is a word over the 2 generators σ_{i_1} and σ_{i_2}.*
2. *For each $1 \leq i < k$, we have that $|\{i_1, i_2\} \cap \{(i+1)_1, (i+1)_2\}| = 1$, the name of $\mathrm{l}[w_i]$ is not in $\{\sigma_{(i+1)_1}, \sigma_{(i+1)_2}\}$ and the name of $\mathrm{f}[w_{i+1}]$ is not in $\{\sigma_{i_1}, \sigma_{i_2}\}$.*
3. *The word w_1 is critical.*
4. *For $i > 1$, the word $\mathrm{l}[\delta(w_{i-1})]w_i$ is critical.*

If w has a rightward critical sequence, the subwords w_i are called the *factors* of the sequence. We will write $u_1 = w_1$, $u_i = \mathrm{l}[\delta(u_{i-1})]w_i$ for $i > 1$ and $u_i' = \mathrm{pre}[\delta(u_i)]$. If we have that $\mathrm{l}[\delta(u_k)] = \mathrm{f}[\beta]^{-1}$, then it is a rightward length reducing sequence and

the reduction of w will be the word $\alpha u_1' u_2' \cdots u_k' \mathrm{suf}[\beta]$ obtained as follows:

$$\begin{aligned} w &= \alpha w_1 w_2 \cdots w_k \beta \\ &= \alpha u_1 w_2 \cdots w_k \beta \\ &=_G \alpha \delta(u_1) w_2 \cdots w_k \beta \\ &= \alpha \mathrm{pre}[\delta(w_1)](\mathrm{l}[\delta(w_1)] w_2) \cdots w_k \beta \\ &= \alpha u_1' u_2 \cdots w_k \beta \\ &= \cdots \\ &= \alpha u_1' u_2' \cdots u_k' \mathrm{suf}[\beta]. \end{aligned}$$

We now state some specific results for the RRS when $|\beta| = 1$.

Definition 4.2.6. Let $w \in A^*$ and $g \in A$ and suppose that wg is freely reduced and admits a RRS with factorisation $\alpha w_1 \cdots w_k g$.

We say that such a RRS is *optimal* if it is of shortest length. That is, if $\alpha' w_1' \cdots w_k' g$ is another RRS, then $|w_1 \cdots w_k| \leq |w_1' \cdots w_k'|$.

Proposition 4.2.7. *Suppose that $w \in A^*$ and $g \in A$ and suppose that wg admits an optimal RRS with factorisation $\alpha w_1 \cdots w_k g$. For each $1 \leq l \leq k$, we have:*

1. *The subword u_i has no proper suffix that is critical. That is, u_i is either of the form ${}_p(x, y)\eta(z^{-1}, t^{-1})_n$ with $p > 0$ or of the form ${}_n(x^{-1}, y^{-1})\eta(z, t)_p$ with $n > 0$, where $\{x, y\} = \{z, t\} = \{a_{i_1}, a_{i_2}\}$. This puts some restrictions on η.*
2. *If the reduction $\alpha u_1' \cdots u_k'$ admits another LLS or RRS, then this whole LLS or RRS is contained in $\alpha u_1'$.*

Left lex reducing sequences (LLS)

We now provide results similar to the RRS for leftward critical sequences.

Proposition 4.2.8. *If a word w admits a leftward critical sequence, then w has a factorisation $w = \alpha w_k w_{k-1} \cdots w_1 \beta$ satisfying the following properties.*

1. *Each w_i is a word over the generators σ_{i_1} and σ_{i_2}.*

2. *For each* $1 \leq i < k$*, we have that* $|\{i_1, i_2\} \cap \{(l+1)_1, (i+1)_2\}| = 1$*, the name of* $\mathrm{f}[w_l]$ *is not in* $\{\sigma_{(i+1)_1}, \sigma_{(i+1)_2}\}$ *and the name of* $\mathrm{l}[w_{i+1}]$ *is not in* $\{\sigma_{i_1}, \sigma_{i_2}\}$*.*
3. *The subword* w_1 *is critical.*
4. *For* $i > 1$*, the word* $w_i \mathrm{f}[\delta(w_{i-1})]$ *is critical.*

Again, if w has a leftward critical sequence with factorisation $w = \alpha w_k w_{k-1} \cdots w_1 \beta$, we say that the subwords w_i are *factors* of the sequence. We will write $u_1 = w_1$, $u_i = w_i \mathrm{f}[\delta(u_{i-1})]$ for $i > 1$ and $u'_i = \mathrm{suf}[\delta(u_i)]$. If we have that $\mathrm{f}[\delta(u_k)] <_{lex} \mathrm{f}[u_k]$, then we have a left lex reducing sequence and the reduction of w will be the word $\alpha \delta(u_k) u'_{k-1} \cdots u'_1 \beta$, obtained as follows:

$$
\begin{aligned}
w &= \alpha w_k \cdots w_2 w_1 \beta \\
&= \alpha w_k \cdots w_2 u_1 \beta \\
&=_G \alpha w_k \cdots w_2 \delta(u_1) \beta \\
&= \alpha w_k \cdots (w_2 \mathrm{f}[\delta(w_1)]) \mathrm{suf}[\delta(w_1)] \beta \\
&= \alpha w_k \cdots u_2 u'_1 \beta \\
&= \cdots \\
&= \alpha \delta(u_k) u'_{k-1} \cdots u'_1 \beta.
\end{aligned}
$$

Definition 4.2.9. Suppose that $w \in A^*$ and $g \in A$ and suppose that wg admits a LLS with factorisation $\alpha w_k \cdots w_1$.

We say that a LLS is *optimal* if it is of longest length. That is, if $\alpha w_k \cdots w_1$ is optimal and $\alpha' w'_k \cdots w'_1$ is another LLS, then $|w_k \cdots w_1| \geq |w'_k \cdots w'_1|$.

Proposition 4.2.10. *Suppose that* $w \in A^*$ *and* $g \in A$ *and suppose that* wg *admits an optimal LLS with factorisation* $\alpha w_k \cdots w_1$*, where* $\mathrm{l}[w_1] = g$*. For each* $1 \leq i \leq k$*, we have:*

1. *The subword* u_i *has no proper prefix that is critical. That is,* u_i *is either of the form* ${}_p(x, y)\eta(z^{-1}, t^{-1})_n$ *with* $n > 0$ *or of the form* ${}_n(x^{-1}, y^{-1})\eta(z, t)_p$ *with* $p > 0$*, where* $\{x, y\} = \{z, t\} = \{a_{i_1}, a_{i_2}\}$*. This puts some restriction on* η*.*
2. *If the reduction* $\alpha \delta(u_k) u'_{k-1} \cdots u'_1$ *admits another LLS or RRS, then this whole LLS or RRS is contained in* $\alpha \delta(u_k)$*.*

Necessity of large type for the Artin groups

Remark 4.2.11. The proofs of statement 2 in Propositions 4.2.7 and 4.2.10 require the hypothesis that the Artin group is of large type. If we allow some $m_{i,j} = 2$, then the reduction might allow for more LLS or RRS to be contained in $u'_2 \cdots u'_k$, in the case of a RRS, or in $u'_{k-1} \cdots u'_1$ in the case of an LLS, as the following example shows.

Example 4.2.12. Let G be the Braid group on 4 generators. We will denote the generators by a, b, c, d, with the relations $aba = bab$, $bcb = cbc$, $cdc = dcd$, and $ac = ca$, $bd = db$, $ad = da$. We choose the following lexical order : $a < a^{-1} < b < b^{-1} < c < c^{-1} < d < d^{-1}$ on our alphabet.

Consider the word $w = dcb^{-1}acd^{-1}$ and $g = c^{-1}$. The word wg admits the following LLS:

$$\begin{aligned} dcb^{-1}a(cd^{-1}c^{-1}) &=_G dcb^{-1}(ad^{-1})c^{-1}d \\ &=_G dc(b^{-1}d^{-1})ac^{-1}d \\ &=_G (dcd^{-1})b^{-1}ac^{-1}d \\ &=_G c^{-1}dcb^{-1}ac^{-1}d, \end{aligned}$$

where the critical subwords u_i are in parenthesis. We observe that the reduction $c^{-1}dcb^{-1}ac^{-1}d$ admits another LLS:

$$\begin{aligned} c^{-1}dcb^{-1}(ac^{-1})d &=_G c^{-1}d(cb^{-1}c^{-1})ad \\ &=_G c^{-1}(db^{-1})c^{-1}bad \\ &=_G c^{-1}b^{-1}dc^{-1}bad. \end{aligned}$$

As such, we see from Example 4.2.12 that if the Artin group is not of Large type, Propositions 4.2.7(2) and 4.2.10(2) are not always true. These statements, in turns, are needed for the proof of Proposition 4.2.15. Consequently, the key map ρ of the algorithm, below, will not be well-defined for Artin groups not of large of type.

The normal form

With all these results, we can finally give the normal form and a recursive method to reduce a given word into its shortlex representative.

Definition 4.2.13. Let $\mathcal{A}_n$ be an Artin group of large type. Let $\mathcal{R}$ be the set of all words that do not admit any left lex reducing sequences or any right length reducing sequences.

The proof that $\mathcal{R}$ is the set of normal forms, given in [HR11], is done by showing that a recursive function $\rho : A^* \to \mathcal{R}$ is well defined. This map ρ is the map such that, given any word w, $\rho(w)$ is the shortlex minimal representative of w. We define ρ in this section and we give, in Section 4.3, an explicit algorithm to compute it.

Definition 4.2.14. Let $w \in \mathcal{R}$ and $g \in A$ and suppose wg does not admit any free reduction. If wg admits an optimal RRS, define $\rho_1(wg)$ as the reduction. If wg admits an optimal LLS, define $\rho_2(wg)$ as the reduction.

Theorem 4.2.15. *Let $w \in \mathcal{R}$, $g \in A$ and suppose that wg is freely reduced and $wg \notin \mathcal{R}$, then the application of exactly one (optimal) RRS or, if wg does not admit a RRS, one (optimal) LLS on wg yields the unique shortlex minimal representative for wg.*

Definition 4.2.16. Define the map $\rho : A^* \to \mathcal{R}$ as follows:

$$
\begin{array}{ll}
\rho(ug) = \rho(\rho(u)g) & \text{if } u \in A^* \text{ and } g \in A\ . \\
\rho(w) = w & \text{if } w \in \mathcal{R} \\
\rho(wg) = \mathrm{pre}[w] & \text{if } w \in \mathcal{R},\ g \in A \text{ and } wg \text{ is not freely reduced} \\
\rho(wg) = \rho_1(wg) & \text{if } w \in \mathcal{R},\ g \in A,\ wg \text{ is freely reduced and } \rho_1(wg) \text{ exists.} \\
\rho(wg) = \rho_2(wg) & \text{if } w \in \mathcal{R},\ g \in A,\ wg \text{ is freely reduced and } \rho_2(wg) \text{ exists} \\
 & \text{and } \rho_1(wg) \text{ does not.}
\end{array}
$$

Proposition 4.2.17. *Let $w \in \mathcal{R}$ and $g \in A$. Then $\rho(wgg^{-1}) = w$ and*

$$\rho(w_{\ m_{i,j}}(a_i, a_j)) = \rho(w_{\ m_{i,j}}(a_j, a_i)).$$

The fact that this map is well defined follows from Theorem 4.2.15, Proposition 4.2.17 as well as from the fact that any prefix of a shortlex minimal representative is also a shortlex representative. Proposition 4.2.17 completes the results needed that two equivalent words in $\mathcal{A}_n$ yield the same shortlex minimal representative through ρ.

With this results, ρ is well defined and, as such, we have that the set $\mathcal{R}$ is in bijection with $\mathcal{A}_n$, since the shortlex minimal representative of a word is unique. We thus have the following theorem.

Theorem 4.2.18. *The set $\mathcal{R}$ is the set of shortlex minimal representatives for $\mathcal{A}_n$.*

4.3 The algorithms

In this section, we build the necessary algorithms to compute the shortlex normal form of an arbitrary word in $\mathcal{A}_n$. The main algorithm, Algorithm 3, computes ρ recursively to output the shortlex normal form of a word. It calls on two auxiliary algorithms to compute an optimal RRS (Algorithm 4) and an optimal LLS (Algorithm 5), if required. We will give the pseudocode for these algorithms here and a Java implementation is included in Appendix B.

The algorithm for the normal form (see Algorithm 3) simply implements the recursive definition of ρ given in Definition 4.2.16. Given an arbitrary word $w = \sigma_{i_i} \cdots \sigma_{i_n}$, we define the shortlex minimal words $w_k = \rho(\sigma_{i_1} \cdots \sigma_{i_k}) = \rho(\rho(w_{k-1})\sigma_{i_k})$. The algorithm is then as follows: for $k = 1$ to n, we read σ_{i_k} and compute $w_k = \rho(w_{k-1}\sigma_{i_k})$ as in Definition 4.2.16.

The complexity of Algorithm 3 depends on that of the algorithms for computing ρ_1 and ρ_2. Before presenting Algorithm 4, which computes ρ_1, we require a few results

Algorithm 3 Main algorithm for the shortlex normal form

Input: Word $w = \sigma_{i_1} \cdots \sigma_{i_n}$ in $\mathcal{A}_n$.
Output: The shortlex minimal representative of w.

1: $u \leftarrow \sigma_{i_1}$
2: **for** $k = 2$ to n **do**
3: **if** $\mathrm{l}[u] = \sigma_{i_k}^{-1}$ **then**
4: $u \leftarrow \mathrm{pre}[u]$
5: **else if** $\rho_1(u\sigma_{i_k})$ exists **then**
6: $u \leftarrow \rho_1(ua_{i_k})$
7: **else if** $\rho_2(u\sigma_{i_k})$ exists **then**
8: $u \leftarrow \rho_2(u\sigma_{i_k})$
9: **else**
10: $u \leftarrow u\sigma_{i_k}$
11: **end if**
12: **end for**
13: **return** u, the shortlex normal form of w.

on the factors of a RRS. This proposition is derived from results in [HR11].

Proposition 4.3.1. *Let $w \in \mathcal{R}$ and let $g \in A$. Suppose that wg is freely reduced and that it admits a RRS with factorization $wg = \alpha w_1 w_2 \cdots w_k g$ and suppose that the name of g is a. Then we have the following:*

1. *w_k is a word over a and b for some generator $b \in S$.*
2. *The name of $\mathrm{l}[w_k]$ is b.*
3. *If $g = a$, then w_k is either of the form ${}_{p-1}(x, y)\xi(a^{-1}, b^{-1})_n$ or of the form ${}_{m-1}(x, y)\xi$, with $\{x, y\} = \{z, t\} = \{a, b\}$ and $\xi \in \{a, b\}^*$.*
4. *If $g = a^{-1}$, then w_k is either of the form ${}_{n-1}(x^{-1}, y^{-1})\xi(a, b)_p$ or of the form ${}_{m-1}(x^{-1}, y^{-1})\xi$, with $\{x, y\} = \{a, b\}$ and $\xi \in \{a, b\}^*$.*
5. *There is exactly one letter $h \in \{a, b, a^{-1}, b^{-1}\}$ such that hw_k is critical. Furthermore, $\alpha w_1 w_2 \cdots w_{k-1} h^{-1}$ is also a RRS.*

Proof: Recall that if wg admits an RRS with factorization $wg = \alpha w_1 w_2 \cdots w_k g$, then $\mathrm{l}[\delta(u_k)] = g^{-1}$, giving us that u_k and w_k are words involving g or g^{-1} and some other letter b, proving (1). Furthermore, Proposition 4.1.12(3) give us that the letters $\mathrm{l}[u_k] = \mathrm{l}[w_k]$ and $\mathrm{l}[\delta(u_k)]$ have different names. Since $\mathrm{l}[\delta(u_k)] = g^{-1}$, we have that the name of $\mathrm{l}[w_k]$ is b, proving (2).

For (3), we have that $\mathrm{l}[\delta(u_k)] = a$ from (2). Since u_k is critical, the definition of δ and Proposition 4.1.12(4 and 5) yields that u_k has to be of the form ${}_p(x, y)\xi(a^{-1}, b^{-1})_n$,

${}_m(x^{-1}, y^{-1})\xi$ or $\xi(a^{-1}, b^{-1})_m$. From Proposition 4.2.7, we have that u_k cannot be of the form $\xi(a^{-1}, b^{-1})_m$ since it contains a proper suffix that is critical. As such, $w_k = \mathrm{suf}[u_k]$ has to be either of the form ${}_{p-1}(x, y)\xi(a^{-1}, b^{-1})_n$ or of the form ${}_{m-1}(x, y)\xi$. The proof of (4) is similar.

Finally, the previous argument for (3) and (4) give us that $h = \mathrm{f}[u_k]$ is totally determined by w_k in the RRS. Furthermore, since we have that $u_k = \mathrm{l}[\delta(u_{k-1})]w_k$, then $h = \mathrm{l}[\delta(u_{k-1})]$ and it follows that $\alpha w_1 w_2 \cdots w_{k-1} h^{-1}$ is a factorization for a RRS.

∎

This proposition is crucial to an efficient implementation of our algorithm for the RRS as it lets us build an RRS starting from the end of w. It also provides us with various criteria to determine when wg does not admit a RRS, such as w_i not having the *right form*, that is the forms in Proposition 4.3.1(3) and (4) given g, or being such that $p(w_i) + n(w_i) < m - 1$.

The algorithm for computing, if it exists, $\rho_1(wg)$, where $w = a_{i_1} \cdots a_{i_k} \in \mathcal{R}$, $g \in A$ and wg is freely reduced, is as follows. We first look at the letter a_{i_k}. If a_{i_k} and g have the same name (Line 4), we stop and wg does not admit a RRS. If the names are different, we read w from the right until we have either read a critical word in the right form (Line 29), read a word such that the required form cannot be obtained (Line 31) or until we read a letter with a new name (Line 16).

In the case where we read a word that cannot yield a critical word of the right form (for example, if the first alternating subword read is $(a, b)_3$ and we read a subword of the form $(x, y)_4$, $\{x, y\} = \{a, b\}$, in an unsigned word or if we need a positive critical subword and we read a negative letter), we stop and wg does not admit any RRS. We do the same thing if we read a letter with a new name and the subword read up to that point is not of the required form.

In the case where we read a new letter and the subword w_1 read up to this

point is of the right form and not critical, we compute $v_1 = pre[\delta(hw_1)]$, where h is the letter needed to have hw_1 critical. We then repeat the algorithm on $w'h^{-1} = \sigma_{i_1} \cdots \sigma_{i_{k-|w_1|}} h^{-1}$ until we read a critical subword of the right form or until we discover that there exists no RRS.

If at step j we read a critical subword of the right form, we compute $v_j = \delta(w_j)$ and we have that the RRS is $\alpha v_j v_{j-1} \cdots v_1$, with α being the prefix of w that has yet to be read by the algorithm. Finally, if at some point we read all of w without finding a critical subword, the algorithm stops and we have that wg admits no RRS.

The algorithm for computing ρ_1 has complexity $O(|w|)$. Indeed, during the algorithm we read at most $|w|$ letters and computing δ can be done in linear time. We note that each letter read by the algorithm is never read more than twice: once by the while loop and once when computing δ. As such, we the complexity is proportional to $2|w|$, giving us a complexity of $O(|w|)$. The algorithm for computing ρ_2 is quite similar and, as we shall see, also has a complexity of $O(|w|)$.

Before giving the algorithm to compute ρ_2, we provide a result analogous to Proposition 4.3.1 for the factors of a LLS. The proof proceeds in the same fashion and we do not include it, but we use the fact that the u_i do not admit proper critical prefix.

Proposition 4.3.2. *Let $w \in W$ and let $g \in A$ in the Artin group $\mathcal{A}_n$. Suppose that wg is free reduced and that it admits a LLS with factorization $wg = \alpha w_k \cdots w_2 w_1$ and suppose that the name of g is a. Then we have the following:*

1. *w_1 is a word over a and b for some generator b.*
2. *If $g = a$, then w_1 is either of the form ${}_n(x^{-1}, y^{-1})\xi(b,a)_p$ or of the form $\xi(b,a)_m$, with $\{x,y\} = \{a,b\}$ and where ξ does not have any subword of the form $(a,b)_p$ or $(b,a)_p$.*
3. *If $g = a^{-1}$, then w_1 is either of the form ${}_p(x,y)\xi(b^{-1},a^{-1})_n$ or of the form $\xi(b^{-1},a^{-1})_n$, with $\{x,y\} = \{a,b\}$ and where ξ does not have any subword of the form $(b^{-1},a^{-1})_n$ or $(a^{-1},b^{-1})_n$.*

The algorithm for finding if wg admits a LLS is more intuitive than the algorithm

Algorithm 4 Algorithm for the RRS

Input: Shortlex minimal word $w = \sigma_{i_1} \cdots \sigma_{i_k}$ in $\mathcal{A}_n$ and $g \in A$.
Output: $\rho_1(wg)$ if it exists or "No RRS".

1: word $u \leftarrow 1$, RRSdone $\leftarrow$ false, $j \leftarrow k$
2: letter $a \leftarrow$ the name of g
3: letter $b \leftarrow$ the name of a_{i_j}
4: **if** $a = b$ **then**
5:    **return** No RRS.
6: **end if**
7: **while** $j \geq 1$ and RRSdone = false **do**
8:    Read w from the right one letter at a time. The subword read this way is w_s.
9:    **if** Letter σ_{i_1} is read from w and $a_{i_1} \in \{a, a^{-1}, b, b^{-1}\}$ **then**
10:       $w_s \leftarrow \sigma_{i_1} w_s$, $j \leftarrow 0$
11:       **if** w_s is critical and of the right form **then**
12:          $u \leftarrow pre[\delta(w_s)]u$, RRSdone $\leftarrow$ true.
13:       **else**
14:          **return** No RRS
15:       **end if**
16:    **else if** The letter σ_{i_t} is read and $\sigma_{i_t} \notin \{a, a^{-1}, b, b^{-1}\}$ **then**
17:       $j \leftarrow t$
18:       **if** $p(w_s) + n(w_s) = m - 1$ and w_s is of the right form **then**
19:          Put h as the letter needed such that hw_s is critical.
20:          $u \leftarrow \text{pre}[\delta(hw_s)]u$, $g \leftarrow h^{-1}$
21:          letter $a \leftarrow$ the name of g
22:          letter $b \leftarrow$ the name of a_{i_j}
23:          $w_s \leftarrow 1$
24:       **else**
25:          **return** No RRS.
26:       **end if**
27:    **else if** The letter σ_{i_t} is read and $\sigma_{i_t} \in \{a, a^{-1}, b, b^{-1}\}$ **then**
28:       $w_s \leftarrow \sigma_{i_t} w_s$
29:       **if** w_s is critical and of the right form **then**
30:          $j \leftarrow t - 1$, $u \leftarrow \text{pre}[\delta(w_s)]u$, RRSdone $\leftarrow$ true.
31:       **else if** w_s is not of the right form anymore **then**
32:          **return** No RRS.
33:       **else**
34:          Keep on reading
35:       **end if**
36:    **end if**
37: **end while**
38: **if** $j > 0$ **then**
39:    $u \leftarrow \sigma_{i_1} \cdots \sigma_{i_j} u$
40: **end if**
41: **return** u, the shortlex normal form of wg.

for a RRS, since the sequence of δ-moves start from the right of the word wg. The complexity of designing the algorithm is in the part that we are looking for the *longest* lex reducing sequence and that we have fewer criteria to stop the algorithm compared to the algorithm for RRS.

The algorithm works as follows. We start with a shortlex minimal word $w = a_{i_1} \cdots a_{i_k}$ and letter $g \in A$. We read wg from the right until the current 2-generator subword being read, w_s, is critical or a letter with a new name is read or if w_s is of the wrong form, that is if w_s cannot be as in Proposition 4.3.2(2) or (3). We let v be the leftward critical sequence computed up to the start of w_s and we let u be the longest LLS computed up to reading the current letter.

If $\mathrm{f}[\delta(w_s)] <_{lex} \mathrm{f}[w_s]$, our longest LLS is now $u = \delta(w_s)v$ and we keep on reading wg. If $f[w_s] <_{lex} \mathrm{f}[\delta(w_s)]$, then we keep on reading. If a letter with a new name is read at letter a_{i_j} and the subword w_s is critical, our leftward critical sequence is now $v' = \mathrm{suf}[\delta(w_s)]v$ and we continue to search for a LLS in $a_{i_1} \cdots a_{i_j} f[\delta(w_s)]$.

If a letter with a new name is read and w_s is not critical, if w_s cannot be a critical word of the right form anymore (as in Proposition 4.3.2(2) or (3)) or if there is no more letter to read, then we stop the algorithm. At this point, if we have some non-empty word u, then the output is the optimal LLS given by αu, where α is the prefix of w that has not been read. If u is empty, we output the original word wg, since wg does not admit a LLS.

We choose to return wg in the case wg admits no LLS rather than create a fail condition for ease of implementation of the main algorithm. Indeed, in such a setting, if wg is free reduced and admits no RRS, then the shortlex normal form of wg is simply the output of Algorithm 5.

We note that we compute δ at most twice in each subword w_s if we wait until we read a new letter or a fail condition before computing it. As such, we see that we have a complexity of $O(|w|)$, since we read at most $|w|$ letters.

As such, Algorithm 3 has a complexity of $O(|w|^2)$, since we read $|w|$ once and at

Algorithm 5 Algorithm for the LLS

Input: Shortlex minimal word $w = \sigma_{i_1} \cdots \sigma_{i_k}$ in $\mathcal{A}_n$ and $g \in A$.
Output: $\rho_2(wg)$ if wg admits an LLS or wg if no LLS are found.

```
1:  u ← 1 { u is the current optimal LLS }
2:  v ← 1 { v is the longest critical sequence up to this subword}
3:  j ← k
4:  w_s ← 1
5:  LLSdone = false
6:  while j ≥ 1 and LLSdone = false do
7:     Read w from the right one letter at a time. The 2-generator subword read this
       way is w_s.
8:     Letter a ← the name of g
9:     Letter b ← the name of the first letter with a name different from a
10:    if Letter σ_{i_t} is read then
11:       if σ_{i_t} ∈ {a, a^{-1}, b, b^{-1}} then
12:          if σ_{i_t} w_s cannot be of the right form then
13:             LLSdone ← true
14:          else if σ_{i_t} w_s is critical then
15:             w_s ← σ_{i_t} w_s
16:             if f[δ(σ_{i_t} w_s)] <_{lex} σ_{i_t} then
17:                u ← δ(σ_{i_t} w_s) v
18:             end if
19:          else
20:             w_s ← σ_{i_t} w_s
21:          end if
22:       else
23:          if w_s is critical then
24:             v ← suf[δ(w_s)] v
25:             Letter a ← name of f[δ(w_s)]
26:             Letter b ← name of σ_{i_t}
27:             w_s ← σ_{i_t} f[δ(w_s)]
28:          else
29:             LLSdone ← true
30:          end if
31:       end if
32:    end if
33: end while
34: if u = 1 then
35:    w' ← wg
36: else
37:    w' = σ_{i_1} ··· σ_{i_{k-(|u|-1)}} u
38: end if
39: return w', the shortlex normal form of wg.
```

each letter read, we have a complexity of $O(|w|)$ to compute ρ_1 and ρ_2. This normal form is thus computable in quadratic time with regards to the length.

Chapter 5

Application of Normal Forms to Cryptography

One of the goals of this thesis is to evalutate the possible use of some Artin groups in cryptography. As such, in this chapter we provide some background in non-commutative group cryptography. We will begin by defining general terms, some hard problems found in groups. We then provide some group-based cryptographic protocols and we argue that some Artin groups are viable platforms. We conclude this chapter, in Section 5.3, by looking at how the normal form presented in Chapter 4 behaves on random words and under conjugacy. Our analysis provides evidence that the Artin groups of large type are not secure under the current conjugacy-based schemes. The first three sections are based on [MSU08].

We will use the standard notation for cryptography throughout the chapter: m is a plaintext message, $c = E_{k_1}(m)$ is the cyphertext obtained by the encryption E_{k_1} using the key k_1. The decryption function is the map D_{k_2} such that $m = D_{k_2}(c)$. If $k_1 = k_2$, then we say that it is a *symmetric* or private key encryption, while if $k_1 \neq k_2$ we say that it is an *asymmetric* or public key encryption.

Before we begin, we provide an informal list of properties a group G should have

to be considered a suitable platform for cryptography. These are based on the list given in [MSU08].

1. The group should be well known and the problem should either be well studied in this group or be reducible to another well studied problem.
2. G should have a fast (quadratic or linear) solution to the word problem or an easily computable (quadradic or linear time) normal form.
3. The problem considered should be hard in G. That is, it should not have a subexponential-time solution by a deterministic algorithm.
4. G should have an exponential growth, that is, the number of elements of any given length n should be exponential in n.

These properties are quite straightforward, but we provide some justification. Property 1 is necessary for any group to be used in a real application, because if G has not been studied well enough, the security of the cryptosystem might be compromised by some property of G that was overlooked. Property 2 is in general useful, since a fast solution to the word problem is necessary in many protocols and a normal form would allow both parties to easily extract the same message or key.

In practice, property 3 might be impossible to prove, but sufficient study of the problem is often accepted as being convincing enough. We note that even if the problem is considered hard, some normal forms might provide a faster solution. As such, in addition to a hard problem, any normal forms in which the problem is easy should also be hard to compute. An example is the prime factorisation problem in the integers: the unique prime factorisation normal form solves this problem in a trivial way, but this normal form is hard to compute.

Finally, property 4 is necessary to prevent "brute force" attacks. If G does not have exponential growth, neither will our key space and trying all the possible keys become a viable solution to any hard problem.

As we have seen throughout this thesis, Artin groups satisfy most of these properties. Artin groups of finite type have been well studied while Artin groups of large type have recently received more attention. In Chapters 3 and 4 we showed that

these groups had an easily computable normal form, giving us both properties 1 and 2. Also, all Artin groups we considered have exponential growth, since if $m_{i,j} \geq 3$ for at least one pair i, j the subgroup generated by σ_i^2, σ_j^2 is a free group. This yields that such an Artin group has a least $2^{\frac{n}{2}}$ elements containing n letters, giving us that these Artin groups have an exponential growth. The only property left to consider is property 3. In section 5.3 we provide a negative result, that is, that Artin groups do not satisfy this property for the conjugacy search problem defined below.

5.1 Hard problems and group requirements

In all forms of cryptography, with public or private key, the security of an encryption is based on the computational difficulty of recovering the original message, m, from the cyphertext $c = E_{k_1}(m)$, when the decryption key k_2 is unknown. In general, an encryption scheme is considered secure if it is computationally infeasible to recover m from c without knowing k_2.

As such, cryptography is based on *hard problems*, that is, problems that are computationally infeasible to solve in the general case and for large parameters. One such problem is the discrete logarithm problem in finite cyclic groups, where given $c = b^a \mod p$, b and p, we need to determine a. We will present some group-theoretic problems that have been suggested for non-commutative group cryptography.

Definition 5.1.1. Let G be a group. A *decision problem* is: given the element g and property p, determine if g has property p.

A *search problem* is: given a property p, find an element $g \in G$ satisfying p.

Definition 5.1.2. A problem is said to be *solvable* if there exists a deterministic algorithm that provides a solution in a finite number steps for all the possible inputs. It is said to be *unsolvable* otherwise.

A problem is said to be *computationally infeasible* if it cannot be solved in reasonable time when the adversary is polynomially bounded in computational power and time.

The discrete logarithm problem is an example of a search problem. The first problem we present for groups is the *word problem*.

Definition 5.1.3. Let G be a group. The *word problem* in G is: given an element $g \in G$, determine if $g = 1_G$.

If a group G has a solvable word problem, then we can also solve the equality problem: given $g, h \in G$, determine if $g = h$. This can be done by checking if $gh^{-1} = 1$. If we wish to use a group G as a cryptographic platform, it is essential that G has a (fast) solution to the word problem. Indeed, it is required in various protocols to be able to verify if two elements are equal.

The word problem is generally solved by giving an algorithm to obtain a unique normal form for each element in G. Finding a normal form for a finitely presented non-abelian group G is, as we have seen, generally not a trivial task. Normal forms are of particular interest in cryptography because they can help hide some mechanisms of the encryption.

We will now present some problems that have been suggested for non-commutative cryptography because they are generally hard to solve and yet simple to state.

Definition 5.1.4. Let G be a group. The *conjugacy problem* is: given $g, h \in G$, determine if there exists $x \in G$ such that $g = h^x = x^{-1}hx$.

The *conjugacy search problem* (CSP) is: given $g, h \in G$ two conjugate elements, find one $x \in G$ such that $g = h^x$.

The *subgroup CSP* is: given $g, h \in G$ two conjugate elements and a subgroup A of G, find one $x \in A$ such that $g = h^x$, provided at least one such x exists.

Definition 5.1.5. Let G be a group and let A, B be subgroups of G. The *decomposition search problem* (DSP) is: given $g, h \in G$, find two elements $a \in A$ and $b \in B$ such that $agb = h$, provided that at least one such pair exists.

The DSP is a generalization of the conjugacy search problem. We shall see in Section 5.2 that some protocols based on the CSP can be attacked by instead solving the decomposition problem. We note that the DSP is easier to solve than the CSP, since the CSP is a special case of the DSP where we require $a = b^{-1}$.

Definition 5.1.6. Let G be a group and let $a_1, \ldots, a_n \in G$. The *membership search problem* is: given x, find, provided it exists, an expression for x in terms of $a_1, \ldots, a_n$.

5.2 Some group-based cryptographic protocols

In this section, we present some public key cryptographic protocols that can be used over non-commutative groups, provided that they are suitable platforms. We will give the protocol and explain what an attacker can do to break it. We work under the standard assumption that the only hidden information is the set of secret keys.

Given a group G and a pair of subgroups A, B, we will say that A, B are *commuting subgroups* if $ab = ba$ for all $a \in A$ and $b \in B$.

The first protocol was given by Ko, Lee et. al. in [KLC+00] and the braid group was the suggested platform. This protocol is quite similar to the Diffie-Helmann key exchange protocol.

Protocol 5.2.1. Let G be a group, let $g \in G$ be a public element and let A, B be two public commuting subgroups of G. The protocol is as follows:

1. Alice picks a random private $a \in A$ and sends g^a to Bob.
2. Bob picks a random private $b \in B$ and sends g^b to Alice.
3. Alice computes $(g^b)^a = g^{ba} = g^{ab}$ and Bob computes $(g^a)^b = g^{ab}$.

The common secret key is g^{ab}.

The attacker has access to g, g^a and g^b. If he can find either an $x \in A$ such that $g^x = g^a$ or a $y \in B$ such that $g^y = g^b$, he can reconstruct the secret key. If one such x or y is found, the key can be obtained by $(g^b)^x = (g^x)^b = (g^a)^b$, since b and x commute. As such, it appears that the attacker needs to solve the CSP restricted to a subgroup, but in fact is suffices to solve the decomposition search problem. Indeed, if the attacker can find some $a_1, a_2 \in A$ such that $a_1 g a_2 = g^a$, then he can compute $a_1 g^b a_2 = b^{-1}(a_1 g a_2) b = b^{-1}(g^a) b = g^{ab}$.

This scheme thus requires the group to have a hard decomposition search problem to be secure and not only a hard CSP.

The next protocol is another key exchange protocol that is a generalisation of Protocol 5.2.1, since it is based on the decomposition search problem. It can be found in [MSU08].

Protocol 5.2.2. Let G be a group, let $g \in G$ be a public element and let A, B be two commuting subgroups of G. The protocol is as follows:

1. Alice picks random private $a_1 \in A$, $b_2 \in B$ and sends $a_1 g b_2$ to Bob.
2. Bob picks a random private $a_2 \in A$, $b_1 \in B$ and sends $b_1 g a_2$ to Alice.
3. Alice computes $a_1 b_1 g a_2 b_2$ and Bob computes $b_1 a_1 g b_2 a_2 = a_1 b_1 g a_2 b_2$.

The common secret key is $a_1 b_1 g a_2 b_2$.

As in Protocol 5.2.1, an attacker needs to solve the decomposition search problem to recover the secret key. This twist appears to be more secure, since the DSP is done in two subgroups instead of only one subgroup.

The next generalization increases the security by hiding part of the subgroups used. Recall that the centralizer of an element $x \in G$ is defined to be the subgroup $C_G(x) = \{g \in G | gx = xg\}$.

Protocol 5.2.3. Let G be a group, let $g \in G$ be public. The protocol is as follows:

1. Alice picks random private $a_1 \in G$, computes $C_G(a_1)$ and publishes a subgroup $A = \langle \alpha_1, \ldots, \alpha_k \rangle \subset C_G(a_1)$.

2. Bob picks random private $b_2 \in G$, computes $C_G(b_2)$ and publishes a subgroup $B = \langle \beta_1, \ldots, \beta_m \rangle \subset C_G(b_2)$.
3. Alice chooses a random $a_2 \in \langle \beta_1, \ldots, \beta_m \rangle$ and sends $a_1 g a_2$ to Bob.
4. Bob chooses a random $b_1 \in \langle \alpha_1, \ldots, \alpha_k \rangle$ and sends $b_1 g b_2$ to Alice.
5. Alice computes $a_1 b_1 g b_2 a_2$ and Bob computes $b_1 a_1 g a_2 b_2 = a_1 b_1 g b_2 a_2$.

The common secret key is $a_1 b_1 g b_2 a_2$, which Bob can obtain since $a_i b_i = b_i a_i$ by construction.

The security of this protocol is two-fold. An attacker has to solve the DSP to find an $x \in C_G(A)$ and a $y \in B$ such that $xgy = a_1 g a_2$. Note that it might not be the case that $a_1 \in A$. It is in general a highly non-trivial task to compute the centralizer of a subgroup, increasing the security of this protocol. The main caveat of Protocol 5.2.3 is that we need to be able to efficiently compute the centralizer, or a subgroup of the centralizer, of any given element in G to have any viable implementation.

The last protocol we present is the Anshel-Anshel-Goldfeld protocol that has first been given in [AAG99]. It differs from the previous protocol in that it does not require the use of any commuting subgroups and can thus easily be used with any non-commutative group.

Protocol 5.2.4. Let G be a group and let $a_1, \ldots, a_k, b_1, \ldots, b_m \in G$ be some elements. The protocol is as follows:

1. Alice picks a (random) private $x \in A = \langle a_1, \ldots, a_k \rangle$ as an expression of the a_i's, which we write as $x = x(a_1, \ldots, a_k)$. Alice then sends $b_1^x, \ldots, b_m^x$ to Bob.
2. Bob picks a (random) private $y \in B = \langle b_1, \ldots, b_m \rangle$ as an expression of the b_i's, which we write as $y = y(b_1, \ldots, b_m)$. Bob then sends $a_1^y, \ldots, a_k^y$ to Alice.
3. Alice computes $x^{-1} \cdot x(a_1^y, \ldots, a_k^y) = x^{-1} x^y = x^{-1} y^{-1} x y$.
4. Bob computes $(y^{-1} \cdot y(b_1^x, \ldots, b_m^x))^{-1} = (y^{-1} y^x)^{-1} = (y^{-1} x^{-1} y x)^{-1} = x^{-1} y^{-1} x y$.

The secret key is then $k = x^{-1} y^{-1} x y$.

Suppose an adversary wants to recover the secret key, by finding an $x' \in A$ such that $b_i^{x'} = b_i^x$ for all i and a $y' \in B$ such that $a_j^{y'} = a_j^y$ for all j. He then needs to know that $x'^{-1} y'^{-1} x' y' = x^{-1} y^{-1} x y$ to recover the secret key.

As such, an attacker has two options: to solve either simultaneous CSP followed by the membership problem or to solve the subgroup CSP. If we solve the CSP and we find an $x' \in G$ such that $b_i^{x'} = b_i^x$ for all i and a y' such that $a_i^{y'} = a_i^y$ for all i, then we have to ensure that $x'^{-1}y'^{-1}x'y' = x^{-1}y^{-1}xy$.

Indeed, the elements $x' = c_b x$ and $y' = c_a y$, where $c_a \in C_G(A)$ and $c_b \in C_G(B)$, are possible solutions to the simultaneous CSP. In this case, we have that $x'^{-1}y'^{-1}xy = x^{-1}y^{-1}xy$. A viable way to find such a solution is to require that $x' \in A$ and $y' \in B$. This forces the attacker to solve the membership problem, which is unsolvable in any groups containing a cartesian product of two free group as a subgroup (see [Mih58]). The other option to obtain the key is thus to solve the harder subgroup CSP to find an $x' \in A$ and a $y' \in B$.

5.3 The CSP in Artin groups of large type

In this section, we consider the CSP in Artin groups of large type, but we also provide some background on the CSP for Artin groups of finite type.

In Artin groups of finite type

In the braid group, early studies were expecting the CSP to be hard, but some techniques based on invariants of the conjugacy classes are proving it wrong for all Artin group of finite type, see [FGM03]. These techniques can solve the CSP in a time proportional to kl^2n^4, where l is the length of the word w, n is the number of Artin generators and k is the size of a particular subset of the conjugacy class of w. Although k seems to be exponential in the number of generators, empirical results lead the authors to conjecture that, for most randomly chosen elements, their algorithm solves the CSP in time polynomial in both the length and the number of generators. It is believed that only a subset of braids are secure for cryptography, but

this set has yet to be determined. The results in [FGM03] were tested in the braid groups, but the algorithms can be used in any Garside groups, which includes all the Artin groups of finite type.

In Artin groups of large type

We now proceed with the Artin groups of large type. As in Chapter 4, we write $w = v$ if w and v are equal as words and we write $w =_G v$ if they are equal as group elements. Also, to simplify our experimentation, we only considered Artin groups where $m_{i,j} = m \geq 3$ for all pairs $i \neq j$. We will denote these groups by $\mathcal{A}_n(m)$.

When we say that a word w of length k is *randomly generated*, we mean that w is uniformly chosen in A^k, where $A = S' \cup S'^{-1}$ and $S' = \{\sigma_1, \ldots, \sigma_n\}$. We then generate words of length k by choosing k letters at random from A, following a uniform distribution.

Our first test on the normal form checks how much reduction generally occurs between a random word w and its shortlex reduction u. This was done by generating a large sample of random words w and by considering their free reduction v and shortlex reduction u. We then computed the ratio of free reduction, given by $\frac{|w|-|v|}{|w|}$, and the ratio of RRS reduction, given by $\frac{|v|-|u|}{|w|}$, for each word and took the mean of our sample.

In Table 5.1, we present these means in various groups $\mathcal{A}_n(m)$, where the sample was 1000 random words of length 2000. As we can see, a random word w will generally see its length reduced by $\frac{1}{n}$ due to free reduction. This is to be expected, since the probability of generating a random subword of the form $\sigma_i^{\pm 1}\sigma_i^{\mp 1}$ is $\frac{1}{2n}$. Together with the fact that free reduction involves an even number of letters, we expect the length to decrease by $\frac{1}{n}$ when free reducing.

Another interesting result we obtain from Table 5.1 is that, as we increase the value of m, the length reduction due to a RRS in a randomly generated, freely reduced

Ratio (in %) of length reduction				
n	m	Free reduction	RRS reduction	Total Reduction
3	3	33.27	10.88	44.15
	4	33.33	2.68	36.01
	5	33.27	0.69	33.96
4	3	25.08	5.80	30.88
	4	24.97	1.02	25.99
	5	25.00	0.19	25.19
5	3	20.00	3.57	23.57
	4	19.94	0.49	20.43
	5	19.97	0.07	20.04

Table 5.1: Ratio of reduction of a randomly generated word by the normal form. Computed on 1000 words of length 2000. The sample variance for the total reduction is between 3.3×10^{-4} and 1.7×10^{-4}, while it is between 1.4×10^{-4} and 1×10^{-6} for the RRS reduction. Both sample variances decrease as n and m increase.

word falls drastically. This follows from the fact that the defining relations are longer and thus the probability of generating a RRS is lower. We did not count the LLS in this first test because they do not affect the length of normal form.

Both of these results point towards the idea that, as n and m increase, a random freely reduced word is, with high probability, a shortlex minimal representative. This is consistent with the fact that, as m tends to infinity, $\mathcal{A}_n(m)$ is closer to being a free group, where all freely reduced words are shortlex minimal representatives.

Knowing by how much a randomly generated word is reduced by the algorithm, we can generate random shortlex minimal representatives of any given length k. It is done in the following fashion in $\mathcal{A}_n(m)$:

1. Let $k' \geq \lfloor k(1+\frac{1}{n}) \rfloor + 1$, where $\lfloor\cdot\rfloor$ is the floor function.
2. Choose a random word w of length k' and compute its shortlex minimal representative v.
3. If $|v| < k$, return to step 2.

4. If $|v| \geq k$, let u be the prefix of length k of v.
5. The randomly generated shortlex minimal representative of length k is u.

We are assured that u is a shortlex minimal representative since it is a prefix of another shortlex minimal representative.

With this process to randomly generate a shortlex minimal representative, we can evaluate the security of the conjugacy problem in Artin groups of large type. This was done by generating a pair of shortlex minimal words u, v and computing the normal form w of uv. We then looked at the maximal prefix u_1 of u that is a prefix of w and the maximal suffix v_2 of v that is also a suffix of w. By computing the ratios $\frac{|u_1|}{|u|}$ and $\frac{|v_2|}{|v|}$ over a large sample of shortlex minimal words, we can determine, in general, how much of the words u and v remain unchanged in the normal form of uv.

Our observations can be found in Tables 5.2 and 5.3, where our samples were 200 words in each group with both u and v having the same length. As we increase the length of u and v, we see that only a small suffix of u and a small prefix of v are involved in any reduction by the algorithm. We observe the same thing as the values of m and n increase.

Conjecture 5.3.1. *In any Artin group of large type, the DSP has an easy solution in general.*

Algorithm 5.3.2. *Recall that the DSP is the following problem: given g and g', find $a \in A$, $b \in B$ such that $agb =_G g'$, provided such a pair exists. Let w and w' be the shortlex minimal representatives of g and g' respectively.*

Write $w = w_1 w_2 w_3$ for some subword w_2. Suppose w_2 can be found as a subword in the middle of w', that is, we can find part of w that was not changed by the algorithm when computing the normal form of awb. Write $w' = \alpha w_2 \beta$, then we have that

$$\alpha w_1^{-1} =_G a$$

For the ratio of u in the normal form of uv				
n	m	Length		
		10	20	50
3	3	0.8165	0.905	0.9618
	4	0.907	0.9577	0.9796
	5	0.952	0.967	0.9895
4	3	0.9165	0.9685	0.9828
	4	0.966	0.9867	0.9945
	5	0.977	0.9898	0.9935
5	3	0.966	0.9812	0.9893
	4	0.98	0.991	0.9957
	5	0.9835	0.9942	0.9973

Table 5.2: The ratio of the longest prefix of u found in the normal form of uv in $\mathcal{A}_n(m)$. The sample was of size 200 in each group. The sample variance for length 10 ranges between 0.05 and 0.002, while it is between 0.003 and 6×10^{-5} for length 50. The sample variances decrease as n and m increase, and as the length increases.

and

$$w_2^{-1}\beta =_G b.$$

By computing the normal forms of αw_1^{-1} *and* $w_2^{-1}\beta$*, we recover the normal forms of* a *and* b*, respectively.*

From the data in Tables 5.2 and 5.3, we see that, in general, at least 60% of w remains unchanged in awb. It follows that we can then find a subword w_2 of w that is also a subword of w', allowing us to solve the DSP. We expect Algorithm 5.3.2 to work with high probability. When it fails, one can try again with a different subword w_2.

This solution to the DSP only requires one to match a substring of w to a substring in the middle of w' and to compute two normal forms. This makes it a fast

For the ratio of v in the normal form of uv				
n	m	Length		
		10	20	50
3	3	0.7875	0.8888	0.961
	4	0.8945	0.9515	0.9769
	5	0.933	0.9657	0.9875
4	3	0.926	0.957	0.9853
	4	0.9675	0.983	0.9937
	5	0.973	0.988	0.9944
5	3	0.969	0.9805	0.9923
	4	0.9795	0.991	0.9962
	5	0.987	0.9942	0.9974

Table 5.3: The ratio of the longest suffix of v found in the normal form of uv in $\mathcal{A}_n(m)$. The sample was of size 200 in each group. The sample variance for length 10 were in between 0.06 and 0.002, while it is between 0.006 and 1×10^{-4} for length 50. The sample variances decrease as n and m increase, and as the length increases.

solution, even if the process is repeated a number of times with different subwords.

Example 5.3.3. In this example, we give an example of how to solve the CSP in $\mathcal{A}_4(3)$ using the method of Algorithm 5.3.2. We will use $a, b, c, \ldots$ for the generators of $\mathcal{A}$ and we will denote by $A, B, C, \ldots$ the respective inverses of $a, b, c, \ldots$. That is, we have $A = a^{-1}$.

Given $w = BaDDAbA$ and $w' = AdABBaaDDAAbaDa$, we find u such that $uwu^{-1} =_G w'$. We consider $w = B\underline{aDDA}bA$, where $w_2 = aDDA$ is the underlined subword. We observe that w_2 is a subword in the middle of w'

$$w' = AdABBa\underline{aDDA}AbaDa,$$

so we put $\alpha = AdABBa =_G uw_1 = uB$ and $\beta = AbaDa =_G w_2v = bAv$. Solving for u and v^{-1}, we can verify if $v =_G u^{-1}$. Computing their normal form, which we did

using our Java implementation, we obtain

$$u = AdABBa(B)^{-1} = AdABBab =_G AdbAA$$

and

$$v = (bA)^{-1}AbaDa = aBAbaDa =_G aaBDa.$$

As it can be seen, we have that $u = v^{-1}$, solving the CSP. We note that these are the exact elements u, u^{-1} used to compute w' in this example.

The previous example shows that even finding a short subword of length 4 is sufficient to solve the CSP. We note that the CSP might be easier to solve than the DSP. Indeed, while solving the CSP, we know if we correctly matched a subword of w to a subword of w' by checking if $u = v^{-1}$. In the case where $u \neq v^{-1}$, we can redo the process with another subword.

We also note that when working with longer words, we can in general find longer subwords of w in w'. This allow for an increased probability of finding the right α and β used in the algorithm. This points to the insecurity of any cryptographic scheme based on the CSP and DSP in the Artin groups of large type.

Conjecture 5.3.4. *In any Artin group of large type, the CSP has an easy solution in general.*

To prove Conjectures 5.3.1 and 5.3.4, one would need to provide a more detailed version of Algorithm 5.3.2 and prove that it recovers u and v from $w' =_G uwv$ with a high probability for any triple u, v, w in any given Artin group of large type. The use of a brute force attack in parallel when working with short words w and w' might also provide a better rate of sucess.

Although results tend to show that the CSP is insecure in the Artin groups of finite and large type, other combinatorial problems have yet to be thoroughly studied in these groups. As such, the existence of an easily computable normal form is a first step in assessing the hardness of new combinatorial problems.

Appendix A

Meet Algorithms

This section contains the algorithms we designed to compute the meet of two reduced elements in the groups $\mathcal{A}(B_n)$ and $\mathcal{A}(D_n)$. They are written in a recursive form, so to compute the meet, we input vectors such that $s = 1$ and $t = n$.

Algorithm 6 Recursive algorithm for left meet in $\mathcal{A}(B_n)$

Input: $A = (a_s, \ldots, a_t)$, $B = (b_s, \ldots, b_t)$ reduced elements.
Output: The left meet $C = A \wedge_L B$.

```
 1: C ← (1, 2, ..., n) {C = 1 in A(A_{n−1})}
 2: m ← ⌊(s+t)/2⌋
 3: Create empty queue Q = {}
 4: Compute C_1 = (a_s, ..., a_m) ∧_L (b_s, ..., b_m)
 5: Compute C_2 = (a_{m+1}, ..., a_t) ∧_L (b_{m+1}, ..., b_t)
 6: C ← C_1 C_2
 7: A' ← C^{−1} A
 8: B' ← C^{−1} B
 9: if A'(m + 1) < A'(m) and B'(m + 1) < B'(m) then
10:    Enqueue σ_m in Q
11: end if
12: while Q is not empty do
13:    Remove σ_i from the start of Q
14:    C ← C σ_i
15:    A' ← σ_i^{−1} A'
16:    B' ← σ_i^{−1} B'
17:    if i > s and A'(i) < A'(i − 1) and B'(i) < B'(i − 1) then
18:       Enqueue σ_{i−1} in Q
19:    end if
20:    if i + 2 ≤ t) and A'(i + 2) < A'(i + 1) and B'(i + 2) < B'(i + 1) then
21:       Enqueue σ_{i+1} in Q
22:    end if
23:    if i = 1 and A'(1) < 0 and B'(1) < 0 then
24:       Enqueue σ_0 in Q
25:    end if
26: end while
27: return The left meet C.
```

Algorithm 7 Recursive algorithm for left meet in $\mathcal{A}(D_n)$

Input: $A = (a_s, \ldots, a_t)$, $B = (b_s, \ldots, b_t)$ reduced elements.
Output: The left meet $C = A \wedge_L B$.

1: $C \leftarrow (1, 2, \ldots, n)$, $m \leftarrow \left\lfloor \frac{s+t}{2} \right\rfloor$
2: Create empty queue $Q = \{\}$
3: Compute $C_1 = (a_s, \ldots, a_m) \wedge_L (b_s, \ldots, b_m)$
4: Compute $C_2 = (a_{m+1}, \ldots, a_t) \wedge_L (b_{m+1}, \ldots, b_t)$
5: $C \leftarrow C_1C_2$
6: $A' \leftarrow C^{-1}A$
7: $B' \leftarrow C^{-1}B$
8: **if** $A'(m+1) < A'(m)$ and $B'(m+1) < B'(m)$ **then**
9:     Enqueue σ_m in Q
10: **end if**
11: **while** Q is not empty **do**
12:     Remove s from the start of Q
13:     $C \leftarrow Cs$
14:     $A' \leftarrow s^{-1}A'$
15:     $B' \leftarrow s^{-1}B'$
16:     **if** $s = \sigma_j$ **then**
17:         $i \leftarrow j$
18:     **end if**
19:     **if** $s = \sigma_0\sigma_1$ **then**
20:         $i \leftarrow 1$
21:     **end if**
22:     **if** $i > s$ and $i > 2$ and $A'(i) < A'(i-1)$ and $B'(i) < B'(i-1)$ **then**
23:         Enqueue σ_{i-1} in Q
24:     **end if**
25:     **if** $i + 2 \leq t$) and $A'(i+2) < A'(i+1)$ and $B'(i+2) < B'(i+1)$ **then**
26:         Enqueue σ_{i+1} in Q
27:     **end if**
28:     **if** $i = 2$ **then**
29:         **if** $A'(2) < A'(1)$ and $B'(2) < B'(1)$ and $A'(1) + A'(2) < 0$ and $B'(1) < B'(2) < 0$ **then**
30:             Enqueue $s = \sigma_0\sigma_1$ in Q
31:         **else if** $A'(2) < A'(1)$ and $B'(2) < B'(1)$ **then**
32:             Enqueue σ_1 in Q
33:         **else if** $A'(1) + A'(2) < 0$ and $B'(1) < B'(2) < 0$ **then**
34:             Enqueue σ_0 in Q
35:         **end if**
36:     **end if**
37: **end while**
38: **return** The left meet C.

Appendix B

Java Implementation of the Algorithm

In this appendix, we include the various Java classes written for the implementation of the normal form in Artin groups of large type. This implementation was designed for testing purposes and is as such not foolproof. If wrong values are passed to the algorithms, it will simply crash instead of sending an error. The purpose being to have a functional implementation of the shortlex normal form algorithm, the code is not optimal, but the algorithm still is quadratic.

We separate the classes into sections depending on their uses.

B.1 Basic object classes

In this section we include the classes that allow for some basic function. The class "NFCoxeterGroup" encodes a Coxeter system.

```
/*
 * This class defines a Coxeter systems variables.
 * It also provide methods to get the various
 * values of "m".
 *
 * These methods can be modified to another scheme
 * of "letter" instead of using characters.
 * For example, one could modify this class as
 * well as the other classes for the normal form to
 * use numbers, where the name is the absolute value
 * and where the inverses are the negative integers.
 *
 * This would require change in the subwords as
 * well such as having a "word" be a sequence of
 * integers, either in a list or in a string that
 * one would parse.
```

```
 *
 * The current implementation is based on latin
 * letters. The name of a letter is its lower case
 * representation while the inverses are the
 * capital letters.
 */

/* Author : Renaud Brien
 * Created : 20/02/12
 * Last update: 20/02/12
 */

import java.util.*;

public class NFCoxeterGroup {

 private int[][] coxMatrix;
 private int rank;

 /* Makes a Coxeter system where
  * all relations are of size m */
 public NFCoxeterGroup ( int rank, int m ) {

  this.rank = rank;
  coxMatrix = new int[rank][rank];

  for(int i = 0 ; i < rank ; i++) {
   coxMatrix[i][i] = 1;
   for (int j = (i+1) ; j < rank ; j++) {
    coxMatrix[i][j] = m;
    coxMatrix[j][i] = m;
   }
  }
 }

 /* Make a Coxeter System with
  * relations of variying length */

 /* arg array must be
  * at least of length "rank * (rank-1) / 2"
  * and all values must be at least 2. */
 public NFCoxeterGroup ( int rank, int[] arg ) {

  this.rank = rank;
  coxMatrix = new int[rank][rank];
  int k = 0;

  for(int i = 0 ; i < rank ; i++) {
   coxMatrix[i][i] = 1;
   for (int j = (i+1) ; j < rank ; j++) {
    coxMatrix[i][j] = arg[k];
    coxMatrix[j][i] = arg[k];
    k++;
   }
  }
 }

 public int[][] getMatrix() {
  return coxMatrix;
 }

 public void printMatrix() {
  for( int i = 0 ; i < rank ; i++ ) {
   System.out.println(
        Arrays.toString(coxMatrix[i]));
  }
 }

 public int getM (int a, int b) {
  return coxMatrix[a][b];
 }

 public int getM (char a, char b) {
  int i = Character.toLowerCase(a)-97;
  int j = Character.toLowerCase(b)-97;
  return coxMatrix[i][j];
 }

}
```

The class “NormalFromLetter” provide functions to work with letters. This class allows for varied implementation of the letters.

```
/*
 * This class provides some necessary functions
 * to compare if two letters  have the same name
 * as well as methods to return the name and sign
 * of a given letter.
 *
 * These methods can be modified to another scheme
 * of "letter" instead of using characters.
 * For example, one could modify this class as
 * well as the other classes for the normal form
 * to use numbers, where the name is the absolute
 * value and where the inverses are the negative
 * integers.
 *
 * This would require change in the subwords as
 * well such as having a "word" be a sequence of
 * integers, either in a list or in a string that
 * one would parse.
 *
 * The current implementation is based on latin
 * letters. The name of a letter is its lower case
 * representation while the inverses are the
 * capital letters.
 */

/* Author : Renaud Brien
 * Created : 19/02/12
 * Last update: 20/02/12
 */

import java.util.*;

public class NormFormLetter {

 public static boolean sameName( Character a,
                                 Character b) {

  /*if ( (a - b == 32) || (a-b == -32) )
   return true;
  else
   return false;
  */

  if (Character.toLowerCase(a) ==
        Character.toLowerCase(b))
   return true;
  else
   return false;
 }

 /* Returns 1 if positive, -1 if negative. */
 public static int sign ( Character a ) {
  if ( a == Character.toLowerCase(a) )
   return 1;
  else
   return -1;
 }

 public static char name ( char a ) {
  return Character.toLowerCase(a);
 }

 public static char inverse( char a ) {
  if ( a == Character.toLowerCase(a) )
   return Character.toUpperCase(a);
  else
   return Character.toLowerCase(a);
 }

}
```

The class “NormalFormSubword” provides functions to save subwords and to compute δ moves. We make the remark that, while writing the code, we used the notation of [HR11], so the “deltaMove” function is the application of τ and the “tauMove” function is the application of δ.

```
/*
 * This class let us create a "subword" object containing
 * all the necessary information to realize the normal
 * algorithms. It stores all the information needed to
 * process a subword whitout knowing what is the final
 * form of the subword. It also contains the method
 * to compute the "tau-move" of a subword.
 *
 * These methods can be modified to another scheme of
 * "letter" instead of using characters. For example,
 * one could modify this class as well as the other
 * classes for the normal form to use numbers, where the
 * name is the absolute value and where the inverses
 * are the negative integers.
 *
 * This would require change in the subwords as well
 * such as having a "word" be a sequence of integers,
 * either in a list or in a string that one would parse.
 *
 * The current implementation is based on latin letters.
 * The name of a letter is its lower case representation
 * while the inverses are the capital letters.
 */

/* Author : Renaud Brien
 * Created : 19/02/12
 * Last update: 20/02/12
 */

import java.util.*;

public class NormFormSubword {

 public char letter1;
 public char letter2;

 public char maxLetter;

 public int sign;
 public boolean critical;

 public int p;
 public int n;
 public int m;

 public int firstP, firstN;
 public int leftP, leftN;

 public String subword = new String();
 public String tauSubword = "";

 public boolean needPCrit = false;
 public boolean needNCrit = false;
 public boolean needPNCrit = false;
 public boolean needNPCrit = false;

 public int critType;
 /* Crit types :
  * 0 = (x,y)m
  * 1 = m(x,y) eta
  * 2 = eta (x,y)m
  * 3 = p(x,y) eta (Z,T)n
  * 4 = n(X,Y) eta (z,t)p
  */

 /* CritType 1 is used only in the RRS
  * CritType 2 is used only in the LLS
  */

 public int etaStart, etaEnd = -1;
 public int rightAltStart, rightAltEnd = -1;
 public int leftAltStart, leftAltEnd = -1;

 public NormFormSubword () {
  this.p = 0;
  this.n = 0;
  critical = false;
  critType = -1;
 }

 public void getOrder() {
  if (letter1 > letter2) {
   maxLetter = letter1;
  }
  else {
   maxLetter = letter2;
  }
 }

 public void add( char a ) {
  subword = a + subword;
 }

 private char deltaMove ( char a ) {
  char b = a;
  if ((m % 2) == 1) {
   if (a == Character.toLowerCase(letter1) ) {
    b = Character.toLowerCase(letter2);
   }
   if (a == Character.toLowerCase(letter2) ) {
    b = Character.toLowerCase(letter1);
   }
   if (a == Character.toUpperCase(letter1) ) {
    b = Character.toUpperCase(letter2);
   }
   if (a == Character.toUpperCase(letter2) ) {
    b = Character.toUpperCase(letter1);
   }
  }
  return b;
 }

 public void tauMove () {

  tauSubword = "";
  char x,y,z,t;
  /* (x,y)m */
  if (critType == 0) {
   t = subword.charAt(0);
   z = subword.charAt(1);
   for (int i=0 ; i < m ; i++) {
    if (i%2 == 0)
     tauSubword = tauSubword + z;
    else
     tauSubword = tauSubword + t;
   }
  }
  /* m(x,y) eta */
  if (critType == 1) {
   z = subword.charAt(subword.length() - 1);
   etaStart = m;
   etaEnd = subword.length() - 1;
   if (sign == -1) {
    if (z == Character.toUpperCase(letter1)) {
     t = Character.toUpperCase(letter2);
    }
    else {
     t = Character.toUpperCase(letter1);
    }
   }
```

```
  else {
   if (z == Character.toLowerCase(letter1)) {
    t = Character.toLowerCase(letter2);
   }
   else {
    t = Character.toLowerCase(letter1);
   }
  }

  for (int i=0 ; i < m ; i++) {
   if (i%2 == 0)
    tauSubword = t + tauSubword;
   else
    tauSubword = z + tauSubword;
  }

  for (int i = etaEnd; i >= etaStart ; i-- ) {
   tauSubword = deltaMove( subword.charAt(i) )
               + tauSubword;
  }
 }
 /* eta (x,y)m */
 if (critType == 2) {
  z = subword.charAt(0);
  etaStart = 0;
  etaEnd = subword.length() - m - 1;
  if (sign == -1) {
   if (z == Character.toUpperCase(letter1))
    t = Character.toUpperCase(letter2);
   else
    t = Character.toUpperCase(letter1);
  }
  else {
   if (z == Character.toLowerCase(letter1))
    t = Character.toLowerCase(letter2);
   else
    t = Character.toLowerCase(letter1);
  }

  for (int i=0 ; i<m ; i++) {
   if (i%2 == 0)
    tauSubword = tauSubword + t;
   else
    tauSubword = tauSubword + z;
  }

  for (int i = etaStart ; i <= etaEnd ; i++) {
   tauSubword = tauSubword +
     deltaMove( subword.charAt(i) );
  }
 }
 /* p(x,y) eta (Z,T)n */
 if (critType == 3) {
  int lastL = subword.charAt(subword.length()-1;
  x = Character.toUpperCase(subword.charAt(0));
  t = Character.toLowerCase(lastL));

  if ( x == Character.toUpperCase(letter1) ) {
   y = Character.toUpperCase(letter2);
  }
  else {
   y = Character.toUpperCase(letter1);
  }

  if ( t == letter1 ) {
   z = letter2;
  }
  else {
   z = letter1;
  }

  etaStart = p;
  etaEnd = subword.length() - n - 1;

  /* (t,z)p */
  for (int i=0 ; i < p ; i++) {
   if (i%2 == 0)
    tauSubword = z + tauSubword;
   else
    tauSubword = t + tauSubword;
  }

  /* delta(eta) (t,z)p */
  if (etaEnd >= etaStart) {
   for (int i = etaEnd ; i >= etaStart ; i--) {
    tauSubword = deltaMove( subword.charAt(i) )
                + tauSubword;
   }
  }

  String tempSubword = "";

  /* n(Y,X) */
  for (int i=0 ; i < n ; i++) {
   if (i%2 == 0)
    tempSubword = tempSubword + y;
   else
    tempSubword = tempSubword + x;
  }

  /* n(Y,X) delta(eta) (t,z)p */
  tauSubword = tempSubword + tauSubword;

 }

 /* n(X,Y) eta (z,t)p */
 if (critType == 4) {
  int lastL = subword.charAt(subword.length()-1;
  x = Character.toLowerCase(subword.charAt(0));
  t = Character.toUpperCase(lastL));

  if ( x == letter1 ) {
   y = letter2;
  }
  else {
   y = letter1;
  }

  if ( t == Character.toUpperCase(letter1) ) {
   z = Character.toUpperCase(letter2);
  }
  else {
   z = Character.toUpperCase(letter1);
  }
  etaStart = n;
  etaEnd = subword.length() - p - 1;

  /* (T,Z)n */
  for (int i=0 ; i < n ; i++) {
   if (i%2 == 0)
    tauSubword = z + tauSubword;
   else
    tauSubword = t + tauSubword;
  }

  /* delta(eta) (t,z)p */
  if (etaEnd >= etaStart) {
   for (int i = etaEnd ; i >= etaStart ; i--) {
    tauSubword = deltaMove( subword.charAt(i) )
                + tauSubword;
   }
  }

  String tempSubword = "";

  /* p(y,x) */
  for (int i=0 ; i < p ; i++) {
   if (i%2 == 0)
    tempSubword = tempSubword + y;
   else
    tempSubword = tempSubword + x;
  }

  /* n(Y,X) delta(eta) (t,z)p */
  tauSubword = tempSubword + tauSubword;
 }
}

public static void main (String[] arg) {

 NormFormSubword sw = new NormFormSubword();
 sw.m = 3;
 sw.letter1 = 'c';
 sw.letter2 = 'b';
 sw.critType = 4;
 sw.n = 2;
 sw.p = 1;
 sw.subword = "BCCbb";
```

```
  sw.tauMove();
  System.out.println( "EtaStart = " + sw.etaStart +
                 " , EtaEnd = " + sw.etaEnd );
  System.out.println( sw.tauSubword );
 }
}
```

B.2 Main algorithm

The class “SLexAlg” is the implementation of Algorithm 3.

```
/*
 * This class is the implementation of the Algorithm to
 * obtain the ShortLex normal form.
 *
 * It outputs the ShortLex representative of the input word.
 */

/* Author : Renaud Brien
 * Created : 20/02/12
 * Last update: 20/02/12
 */

import java.util.*;

public class SLexAlg {

 public static String slexAlg ( String word,
                              NFCoxeterGroup cGroup ) {

  int n = word.length();
  String w = "";

  if (word == null || word.equals("") )
   return word;

  w = w + word.charAt(0);

  for (int i = 1 ; i < n ; i++) {
   char g = word.charAt(i);

   if ( w.length() == 0 ){
    w = w + g;
   }
   else {
    char lastOfW = w.charAt(w.length()-1);

    if ( (NormFormLetter.sameName(lastOfW, g))
       && ( NormFormLetter.sign(g)
           != NormFormLetter.sign(lastOfW) ) ) {
     w = w.substring( 0, w.length()-1 );
    }
    else {
     String tmp;
     tmp = RRSAlg.algRRS( w, g , cGroup);

     if (tmp != null) {
      w = tmp;
     }
     else {
      w = LLSAlg.algLLS( w, g, cGroup );
     }
    }
   }
  }
  return w;
 }

 public static void main ( String[] arg ) {
  NFCoxeterGroup cGroup = new NFCoxeterGroup( 4, 3);

/*int[] array = { 3, 2, 2,
                    3, 2,
                       3 };
  cGroup = new NFCoxeterGroup( 4, array );
*/
  String redString;
  String w = arg[0];

  //System.out.println( "Start with: " + w );

  redString = slexAlg(w, cGroup);

  System.out.println("Starting word is " + w);
  System.out.println("Reduction is     " +
                    redString);
 }
}
```

B.3 RRS algorithm

The class “RRSAlg” is the implementation of Algorithm 4.

```
/*
 * This class is the implementation of the
 * Right Length Reducing Sequence.
 *
 * Input: slex reduced word "w" and letter "g".
 * It outputs:
 *   a null string if there are no RRS,
 *   the slex form of wg if there is a RRS.
 */

/* Author : Renaud Brien
 * Created : 20/02/12
 * Last update: 28/02/12
 */

import java.util.*;

public class RRSAlg {

 private static int sign( char a ) {
```

```
 return NormFormLetter.sign( a );
}

private static char name( char a ) {
 return NormFormLetter.name( a );
}

private static boolean sameName(char a, char b) {
 return NormFormLetter.sameName(a,b);
}

public static String algRRS ( String w,
                              char startingG,
                      NFCoxeterGroup cGroup ) {

 boolean valid = true;
 boolean done = false;
 NormFormSubword currentSWord;
 String redWord = "";
 boolean subEnd, firstAltWord;

 char g = startingG;
 char currl, prev;

 int k = w.length() - 1;

 currentSWord = new NormFormSubword();

 while ( (valid == true) && (k >= 0)
         && (done == false) ) {
  int p = 0;
  int n = 0;
  int firstP = 0;
  int firstN = 0;
  int tempP = 0;
  int tempN = 0;

  if (sameName(w.charAt(k), g)){
   valid = false;
   return null;
  }

  currl = w.charAt(k);
  prev = g;

  currentSWord.letter1 = name(g);
  currentSWord.letter2 = name(currl);
  currentSWord.m = cGroup.getM( currentSWord.letter1,
                               currentSWord.letter2);
  currentSWord.sign = sign( currl );

  if (sign(currl) == -1) {
   tempN = 1;
   n = 1;
   if ( sign(currl) == sign(g)) {
    currentSWord.needPNCrit = true;
   }
   else {
    currentSWord.needNCrit = true;
   }
  }
  else {
   tempP = 1;
   p = 1;
   if ( sign(currl) == sign(g)) {
    currentSWord.needNPCrit = true;
   }
   else {
    currentSWord.needPCrit = true;
   }
  }

  currentSWord.add( currl );

  k--;

  subEnd = false;
  firstAltWord = true;

  while ( k >= 0 && valid == true &&
          subEnd == false) {
   prev = currl;
   currl = w.charAt(k);

/* Begin section if letters have same
 * name and thus same sign,
 * since the word is freely reduced*/
if ( sameName(prev,currl) ) {
 if (firstAltWord == true) {
  if (sign(prev) == 1) {
   firstAltWord = false;
   firstP = tempP;
   if (tempP > p)
    p = tempP;
   if (p == currentSWord.m) {
    currentSWord.critType = 0;
    currentSWord.critical = true;
    currentSWord.p = firstP;
    subEnd = true;
    done = true;
    if (currentSWord.needPCrit == false)
     valid = false;
    }
    else{
     currentSWord.critType = 1;
     currentSWord.add( currl );
    }
    tempP = 1;
    tempN = 0;
   }
   else if (sign(prev) == -1) {
    firstAltWord = false;
    firstN = tempN;
    if (tempN > n)
     n = tempN;
    if (n == currentSWord.m) {
     currentSWord.critType = 0;
     currentSWord.critical = true;
     currentSWord.n = firstN;
     subEnd = true;
     done = true;
     if (currentSWord.needNCrit == false)
       valid = false;
    }
    else{
     currentSWord.critType = 1;
     currentSWord.add( currl );
    }
    tempN = 1;
    tempP = 0;
   }
  }
  else {
   currentSWord.add( currl );
   if (sign(prev) == 1) {
    if (tempP > p)
     p = tempP;
    tempP = 1;
    tempN = 0;
   }
   else {
    if (tempN > n)
     n = tempN;
    tempN = 1;
    tempP = 0;
   }
  }
 }
 /* End section if letters have same name */

 /* Begin section with different name */
 else {
  /* New letter */
  if (name(currl) != currentSWord.letter1
   && name(currl) != currentSWord.letter2) {

   subEnd = true;
   if (firstAltWord == true) {
    if (sign(prev) == 1) {
     firstP = p;
     n = 0;
     currentSWord.critType = 0;
     firstAltWord = false;
    }
    else {
     firstN = n;
     p = 0;
     currentSWord.critType = 0;
```

```
  firstAltWord = false;
 }
}
if (p + n < currentSWord.m - 1) {
 currentSWord.critical = false;
 valid = false;
}
else /* if (p + n == currentSWord.m - 1) */ {
     /* Critical type 1 : m(x,y) eta */
 if (currentSWord.critType <= 1) {
  if (sign(prev) == 1) {
   if (tempP == currentSWord.m-1
       && currentSWord.needPCrit == true) {
    valid = true;
    currentSWord.p = p + 1;
    currentSWord.n = n;
    if ( prev == currentSWord.letter2 ) {
     currentSWord.add( currentSWord.letter1 );
     g = Character.toUpperCase(
                   currentSWord.letter1);
    }
    else {
     currentSWord.add( currentSWord.letter2 );
     g = Character.toUpperCase(
                   currentSWord.letter2);
    }
    currentSWord.critical = true;
   }
   else if (firstP == currentSWord.m-1
          && currentSWord.needNPCrit == true) {
    valid = true;
    currentSWord.critType = 4;
    currentSWord.p = firstP;
    currentSWord.n = n+1;
    if ( prev == currentSWord.letter1 ) {
     currentSWord.add(
       Character.toUpperCase(
                   currentSWord.letter2) );
     g = currentSWord.letter2;
    }
    else if ( prev == currentSWord.letter2 ) {
     currentSWord.add(
       Character.toUpperCase(
                  currentSWord.letter1) );
     g = currentSWord.letter1;
    }
    currentSWord.critical = true;
   }
   else {
    valid = false;
   }
  }
  else if (sign(prev) == -1) {
   if (tempN == currentSWord.m-1
       && currentSWord.needNCrit == true) {
    valid = true;
    currentSWord.p = p;
    currentSWord.n = n+1;
    if (prev == Character.toUpperCase(
                    currentSWord.letter1)){
     currentSWord.add(
           Character.toUpperCase(
                   currentSWord.letter2) );
     g = currentSWord.letter2;
    }
    else {
     currentSWord.add(
       Character.toUpperCase(
                   currentSWord.letter1) );
     g = currentSWord.letter1;
    }
    currentSWord.critical = true;
   }
   else if (firstN == currentSWord.m-1
     && currentSWord.needPNCrit == true) {
    valid = true;
    currentSWord.p = p+1;
    currentSWord.n = firstN;
    currentSWord.critType = 3;
    if (prev == Character.toUpperCase(
                     currentSWord.letter1)){
     currentSWord.add(currentSWord.letter2 );
     g = Character.toUpperCase(
                   currentSWord.letter2);
     }
     else {
      currentSWord.add( currentSWord.letter1 );
      g = Character.toUpperCase(
                      currentSWord.letter1);
     }
     currentSWord.critical = true;
    }
    else {
     valid = false;
    }
   }
  }
  /* Critical Type 3 : p(x,y) eta (Z,T)n */
  else if ( currentSWord.critType == 3 ){
   if (tempP == currentSWord.m - n - 1) {
    valid = true;
    currentSWord.p = p+1;
    currentSWord.n = n;
    if ( prev == currentSWord.letter1 ) {
     currentSWord.add( currentSWord.letter2 );
     g = Character.toUpperCase(
                     currentSWord.letter2);
    }
    else {
     currentSWord.add( currentSWord.letter1 );
     g = Character.toUpperCase(
                     currentSWord.letter1);
    }
    currentSWord.critical = true;
   }
   else {
    valid = false;
   }
  }
  /* Critical type 4 : n(X,Y) eta (z,t)p */
  else {
   if (tempN == currentSWord.m - p - 1) {
    valid = true;
    currentSWord.p = p;
    currentSWord.n = n+1;
    if (prev == Character.toUpperCase(
                      currentSWord.letter1)){
     currentSWord.add(
       Character.toUpperCase(
                      currentSWord.letter2) );
     g = currentSWord.letter2;
    }
    else {
     currentSWord.add(
       Character.toUpperCase(
                    currentSWord.letter1) );
     g = currentSWord.letter1;
    }
    currentSWord.critical = true;
   }
   else {
    valid = false;
   }
  }
 }
 k++;
}
/* Same Letters (different names) */
else {
 /* Same sign */
 if (sign(prev) == sign(currl)) {
  if (sign(prev) == 1) {
   tempP++;
   if (tempP > p)
    p = tempP;
   currentSWord.add(currl);
   if (tempP == currentSWord.m) {
    if (firstAltWord == true ) {
     firstP = tempP;
     firstAltWord = false;
     currentSWord.critType = 0;
    }
    if ( currentSWord.needPCrit == true ) {
     valid = true;
     subEnd = true;
     currentSWord.critical = true;
     done = true;
    }
```

```
      else {
       valid = false;
       subEnd = true;
      }
     }
     if ( currentSWord.critType ==  4) {
      if (p > firstP) {
       valid = false;
       subEnd = true;
      }
     }
     else if ( currentSWord.critType == 3 ) {
      if ( tempP + firstN == currentSWord.m ) {
       valid = true;
       subEnd = true;
       done = true;
       currentSWord.n = firstN;
       currentSWord.p = tempP;
       currentSWord.critical = true;
      }
     }
    }
    else /* if (sign(prev) == -1) */ {
     tempN++;
     if (tempN > n)
      n = tempN;
     currentSWord.add(currl);
     if (tempN == currentSWord.m) {
      if (firstAltWord == true ) {
       firstN = tempN;
       firstAltWord = false;
       currentSWord.critType = 0;
      }
     if ( currentSWord.needNCrit == true ) {
      valid = true;
      subEnd = true;
      currentSWord.critical = true;
      done = true;
     }
     else {
      valid = false;
      subEnd = true;
     }
    }
    if ( currentSWord.critType ==  3) {
     if (n > firstN) {
      valid = false;
      subEnd = true;
     }
    }
    else if ( currentSWord.critType == 4 ) {
     if ( tempN + firstP == currentSWord.m ) {
      valid = true;
      subEnd = true;
      done = true;
      currentSWord.n = tempN;
      currentSWord.p = firstP;
      currentSWord.critical = true;
     }
    }
   }
  }
  /* Different signs */
  else {
   if (sign(prev) == 1) {
    tempN = 1;
    tempP = 0;
    if (firstAltWord == true) {
     firstP = p;
     firstAltWord = false;
     if ( currentSWord.needNPCrit == false ) {
      valid = false;
      subEnd = false;
     }
     else {
      currentSWord.critType = 4;
     }
    }
    else if ( currentSWord.critType < 3 ) {
     if (currentSWord.needNPCrit == false ) {
      valid = false;
      subEnd = false;
     }
     else {
      currentSWord.critType = 4;
      if (p > firstP) {
       valid = false;
       subEnd = true;
      }
     }
    }
    if ( currentSWord.critType == 4 && valid) {
     if (firstP == currentSWord.m - 1) {
      n = tempN;
      currentSWord.add(currl);
      valid = true;
      subEnd = true;
      done = true;
      currentSWord.critical = true;
      currentSWord.p = firstP;
      currentSWord.n = 1;
     }
     else {
      currentSWord.add(currl);
      if (tempN > n)
       n = tempN;
      valid = true;
      subEnd = false;
     }
    }
   }
   else if ( sign(prev) == -1 ) {
    tempP = 1;
    tempN = 0;
    if (firstAltWord == true) {
     firstN = n;
     firstAltWord = false;
     if ( currentSWord.needPNCrit == false ) {
      valid = false;
      subEnd = false;
     }
     else {
      currentSWord.critType = 3;
     }
    }
    else if ( currentSWord.critType < 3 ) {
     if ( currentSWord.needPNCrit == false ) {
      valid = false;
      subEnd = false;
     }
     else {
      currentSWord.critType = 3;
      if (n > firstN) {
       valid = false;
       subEnd = true;
      }
     }
    }
    if ( currentSWord.critType == 3 && valid) {
     if (firstN == currentSWord.m - 1) {
      p = tempP;
      currentSWord.add(currl);
      valid = true;
      subEnd = true;
      done = true;
      currentSWord.critical = true;
      currentSWord.p = 1;
      currentSWord.n = firstN;
     }
     else {
      if (tempP > p)
       p = tempP;
      currentSWord.add(currl);
      valid = true;
      subEnd = false;
     }
    }
   }
  }
 }  /* End section with different names */
 k--;
 if (done)
  k++;
} /* End subword "while" loop */

if (k < 0 && done == false) {
 valid = false;
```

```
    }

    if (valid && currentSWord.critical) {
     currentSWord.tauMove();
     String temp = currentSWord.tauSubword;

     redWord = currentSWord.tauSubword.substring( 0 ,
       currentSWord.tauSubword.length() - 1) + redWord;
     currentSWord = new NormFormSubword();
    }

  }
  /*End main loop.
   * Exits if not valid RRS or if "done".*/

  if ( done ) {
   if ( k >= 0) {
    redWord = w.substring( 0 , k ) + redWord;
   }
  }
  if (valid) {
   return redWord;
  }
  else {
   return null;
  }
 }

 public static void main ( String[] arg ) {
  NFCoxeterGroup cGroup = new NFCoxeterGroup( 4, 3);

  String rrsString;
  String w = arg[0];
  char g = arg[1].charAt(0);

  rrsString = algRRS( w, g, cGroup);

//  System.out.println("First g is " + g);
//  System.out.println("Starting word is " + w);
  System.out.println("Reduction is     "
                     + rrsString);
 }

}
```

B.4 LLS algorithm

The class “LLSAlg” is the implementation of Algorithm 5.

```
/*
 * This class is the implementation of the
 * Left Lex Reducing Sequence.
 *
 * Input: slex reduced word w and letter g.
 * It outputs the lex reduced form of wg,
 * if there is a LLS
 * and the untouched wg, if there is no LLS.
 */

/* Author : Renaud Brien
 * Created : 20/02/12
 * Last update: 4/03/12
 */

import java.util.*;

public class LLSAlg {

 private static int sign( char a ) {
  return NormFormLetter.sign( a );
 }

 private static char name( char a ) {
  return NormFormLetter.name( a );
 }

 private static boolean sameName(char a, char b) {
  return NormFormLetter.sameName(a,b);
 }

 public static String algLLS ( String w,
                         char startingG,
                 NFCoxeterGroup cGroup ) {

  boolean valid = true;
  boolean done = false;
  NormFormSubword currentSWord;
  String lexWord = "";
  boolean subEnd, firstAltWord;
  boolean missingLetter;

  String best = "";
  int bestK = -1;
  String longest = "";
  int longestK = -1;
  String subBest;
  int subBestK;
  int subBestLength;

  int wordEnd;

  char g = startingG;
  char currl, prev;

  int k = w.length() - 1;

  currentSWord = new NormFormSubword();

  while ( (valid == true) && (k >= 0)
         && (done == false) ) {

   int p = 0;
   int n = 0;
   int firstP = 0;
   int firstN = 0;
   int tempP = 0;
   int tempN = 0;

   subBest = "";
   subBestK = -1;
   subBestLength = 0;

   missingLetter = true;

   currl = g;

   currentSWord.letter1 = name(currl);
   currentSWord.add(currl);

   subEnd = false;
   firstAltWord = true;

   if (sign(currl) == 1) {
    currentSWord.sign = 1;
    tempP = 1;
    p = 1;
   }
   else {
    currentSWord.sign = -1;
    tempN = 1;
    n = 1;
   }
```

```
while ( k >= 0 && valid == true
        && subEnd == false) {

 prev = currl;
 currl = w.charAt(k);

 if ( missingLetter == true
      && sameName(prev, currl) == false) {
  currentSWord.letter2 = name(currl);
  currentSWord.m = cGroup.getM( currentSWord.letter1,
                                currentSWord.letter2);
  currentSWord.getOrder();
  missingLetter = false;
 }

 /* Begin section if letters have
  * same name and thus same sign,
  * since the word is freely reduced*/
 if ( sameName(prev,currl) ) {
  if (firstAltWord == true) {
   if (sign(prev) == 1) {
    firstP = tempP;
    if (tempP > p)
     p = tempP;
     if (p == currentSWord.m) {
       currentSWord.critType = 2;
       currentSWord.critical = true;
       currentSWord.p = firstP;
       currentSWord.needPCrit = true;
     }
     else {
      currentSWord.critType = 4;
      currentSWord.p = firstP;
      currentSWord.firstP = firstP;
     }
     currentSWord.add( currl );
     tempP = 1;
     tempN = 0;
     firstAltWord = false;
    }
    else if (sign(prev) == -1) {
     firstN = tempN;
     if (tempN > n)
      n = tempN;
     if (n == currentSWord.m) {
      currentSWord.critType = 2;
      currentSWord.critical = true;
      currentSWord.n = firstN;
      currentSWord.needPCrit = true;
     }
     else{
      currentSWord.critType = 3;
      currentSWord.n = firstN;
      currentSWord.firstN = firstN;
     }
     currentSWord.add( currl );
     tempN = 1;
     tempP = 0;
     firstAltWord = false;
    }
   }
   else {
    currentSWord.add( currl );
    if (sign(prev) == 1) {
     if (tempP > p)
      p = tempP;
     tempP = 1;
     tempN = 0;
     if (currentSWord.critType == 4 ||
        ( currentSWord.critType == 3
         && firstN < currentSWord.m - 1 ) ) {
       currentSWord.critical = false;
     }
    }
    else {
     if (tempN > n)
      n = tempN;
     tempN = 1;
     tempP = 0;
     if (currentSWord.critType == 3 ||
        ( currentSWord.critType == 4
         && firstP < currentSWord.m - 1 ) ) {
      currentSWord.critical = false;
     }
    }
   }
  }
 }
 /* End section if letters have same name */

 /* Begin section if letters have
  * different names */
 else {
  /* New letter */
  if (name(currl) != currentSWord.letter1
       && name(currl) != currentSWord.letter2) {
   k++;
   subEnd = true;
  }
  /* Different name, Same two letters */
  else {
   /* Same Sign */
   if (sign(currl) == sign(prev)) {
    if (sign(currl) == 1) {
     tempP++;
     if (tempP > p)
      p = tempP;

     if (firstAltWord) {
      if( tempP == currentSWord.m ){
       currentSWord.add(currl);
       firstP = tempP;
       currentSWord.critType = 2;
       currentSWord.p = firstP;
       currentSWord.firstP = firstP;
       currentSWord.critical = true;
       firstAltWord = false;
      }
     else {
      currentSWord.add(currl);
     }
    }
    else {
     if (currentSWord.critType == 2) {
      if (tempP >= currentSWord.m) {
       k++;
       subEnd = true;
       done = true;
      }
      else {
       currentSWord.add(currl);
      }
     }
     else if (currentSWord.critType == 4) {
      if ( tempP > firstP ){
       k++;
       subEnd = true;
       done = true;
      }
      else {
       currentSWord.add(currl);
      }
     }
     else /* if critType == 3 */ {
      if (tempP + firstN == currentSWord.m){
       currentSWord.add(currl);
       currentSWord.critical = true;
      }
      else if (tempP < currentSWord.m - firstN){
       currentSWord.add(currl);
       currentSWord.critical = false;
      }
      else {
       /*if (tempP > currentSWord.m - firstN)*/
       k++;
       subEnd = true;
       done = true;
      }
     }
    }
   }
   /* sign == -1 */
   else {
    tempN++;
    if (tempN > n)
     n = tempN;
    if (firstAltWord) {
     if( tempN == currentSWord.m ){
      currentSWord.add(currl);
```

```
     firstN = tempN;
     currentSWord.critType = 2;
     currentSWord.n = firstN;
     currentSWord.firstN = firstN;
     currentSWord.critical = true;
     firstAltWord = false;
    }
    else {
     currentSWord.add(currl);
    }
   }
   else {
    if (currentSWord.critType == 2) {
     if (tempN >= currentSWord.m) {
      k++;
      subEnd = true;
      done = true;
     }
     else {
      currentSWord.add(currl);
     }
    }
    else if (currentSWord.critType == 3) {
     if ( tempN > firstN ){
      k++;
      subEnd = true;
      done = true;
     }
     else {
      currentSWord.add(currl);
     }
    }
    else /* if critType == 4 */{
     if (tempN + firstP == currentSWord.m){
      currentSWord.add(currl);
      currentSWord.critical = true;
     }
     else if (tempN < currentSWord.m - firstP){
      currentSWord.add(currl);
      currentSWord.critical = false;
     }
     else {
      /* if (tempN > currentSWord.m - firstP) */
      k++;
      subEnd = true;
      done = true;
     }
    }
   }
  }
 }
 /* Different signs */
 else {
  if (sign(prev) == 1) {
   if (firstAltWord) {
    firstP = tempP;
    if (firstP == currentSWord.m) {
     currentSWord.critical = true;
     currentSWord.critType = 2;
     currentSWord.p = firstP;
     currentSWord.firstP = firstP;
     subEnd = true;
     done = true;
     k++;
    }
    else {
     currentSWord.critType = 4;
     currentSWord.add(currl);
     currentSWord.p = firstP;
     currentSWord.firstP = firstP;

     if (firstP == currentSWord.m - 1) {
      currentSWord.critical = true;
     }
    }
    firstAltWord = false;
   }
   else {
    if (currentSWord.critType == 2) {
     subEnd = true;
     done = true;
     k++;
    }
    else if (currentSWord.critType == 3) {
     currentSWord.critical = false;
     currentSWord.add(currl);
    }
    else /* if critType == 4 */ {
     currentSWord.add(currl);
     if (firstP == currentSWord.m - 1) {
      currentSWord.critical = true;
     }
    }
   }
   tempP = 0;
   tempN = 1;
  }
  else {
   if (firstAltWord) {
    firstN = tempN;
    if (firstN == currentSWord.m) {
     currentSWord.critical = true;
     currentSWord.critType = 2;
     currentSWord.n = firstN;
     currentSWord.firstN = firstN;
     subEnd = true;
     done = true;
     k++;
    }
    else {
     currentSWord.critType = 3;
     currentSWord.add(currl);
     currentSWord.n = firstN;
     currentSWord.firstN = firstN;

     if (firstN == currentSWord.m - 1) {
      currentSWord.p = 1;
      currentSWord.critical = true;
     }
    }
    firstAltWord = false;
   }
   else {
    if (currentSWord.critType == 2) {
     subEnd = true;
     done = true;
     k++;
    }
    else if (currentSWord.critType == 4) {
     currentSWord.critical = false;
     currentSWord.add(currl);
    }
    else /* if critType == 3 */ {
     currentSWord.add(currl);
     if (firstN == currentSWord.m - 1) {
      currentSWord.critical = true;
     }
    }
   }
   tempP = 1;
   tempN = 0;
  }
 }
}
}
/* End section if letters have different names */

if ( currentSWord.critical ) {
 if (name(currentSWord.subword.charAt(0)) ==
       name(currentSWord.maxLetter) ){
  subBestK = k;
  subBestLength = currentSWord.subword.length();
 }
}

k--;
} /* End of subword "while" loop */

if (subBestK != -1) {
 if (subBestK == k+1) {
  if (currentSWord.critType == 3) {
   currentSWord.n = currentSWord.firstN;
   currentSWord.p =
   currentSWord.m - currentSWord.firstN;
  }
  else if (currentSWord.critType == 4) {
   currentSWord.p = currentSWord.firstP;
   currentSWord.n =
```

```
      currentSWord.m - currentSWord.firstP;
  }
  currentSWord.tauMove();
  subBest = currentSWord.tauSubword;

  if (longest.length() == 0) {
   best = subBest;
  }
  else {
   best = subBest + longest.substring(1);
  }
  longest = best;
  g = longest.charAt(0);
  bestK = subBestK;
  longestK = subBestK;
 }
 else {
  if (currentSWord.critical) {
   if (currentSWord.critType == 3) {
    currentSWord.n = currentSWord.firstN;
    currentSWord.p =
        currentSWord.m - currentSWord.firstN;
   }
   else if (currentSWord.critType == 4) {
    currentSWord.p = currentSWord.firstP;
    currentSWord.n =
        currentSWord.m - currentSWord.firstP;
   }
   currentSWord.tauMove();
   String tempLongest = longest;

   if (longest.length() == 0) {
    longest = currentSWord.tauSubword;
   }
   else {
    longest = currentSWord.tauSubword
         + longest.substring(1);
   }
   longestK = k+1;
   g = longest.charAt(0);

   int subBestStart = currentSWord.subword.length()
                          - subBestLength;
   currentSWord.subword =
      currentSWord.subword.substring( subBestStart );
   currentSWord.critical = true;
   currentSWord.tauMove();
   bestK = subBestK;

   if (tempLongest.length() > 0) {
    best = currentSWord.tauSubword
          + tempLongest.substring(1);
   }
   else {
    best = currentSWord.tauSubword;
   }
  }
  else{
   done = true;

   if (currentSWord.critType == 3) {
    currentSWord.n = currentSWord.firstN;
    currentSWord.p =
      currentSWord.m - currentSWord.firstN;
   }
   else if (currentSWord.critType == 4) {
    currentSWord.p = currentSWord.firstP;
    currentSWord.n =
      currentSWord.m - currentSWord.firstP;
   }

   int subBestStart =
     currentSWord.subword.length() - subBestLength;
   currentSWord.subword =
     currentSWord.subword.substring( subBestStart );
   currentSWord.critical = true;
   currentSWord.tauMove();
    bestK = subBestK;

    if (longest.length() > 0) {
     best = currentSWord.tauSubword
           + longest.substring(1);
    }
    else {
     best = currentSWord.tauSubword;
    }
   }
  }
 }
 else {
  if ( currentSWord.critical == true ) {
   if (currentSWord.critType == 3) {
    currentSWord.n = currentSWord.firstN;
    currentSWord.p =
      currentSWord.m - currentSWord.firstN;
   }
   else if (currentSWord.critType == 4) {
    currentSWord.p = currentSWord.firstP;
    currentSWord.n =
      currentSWord.m - currentSWord.firstP;
   }

   currentSWord.tauMove();

   if (longest.length() == 0) {
    longest = currentSWord.tauSubword;
   }
   else {
    longest = currentSWord.tauSubword
         + longest.substring(1);
   }
   longestK = k+1;
   g = longest.charAt(0);
  }
  else {
   done = true;
  }
 }

 currentSWord = new NormFormSubword();
 } /* End of main loop.
   * Exits if no valid LLS or if "done". */

 if ( done || k < 0 ) {
  if ( bestK >= 0) {
   lexWord = w.substring( 0 , bestK ) + best;
  }
 }
 if (bestK > -1) {
  return lexWord;
 }
 else {
  return (w+startingG);
 }
}

public static void main ( String[] arg ) {
 NFCoxeterGroup cGroup = new NFCoxeterGroup( 4, 3);

 String llsString;

 String w = arg[0];
 char g = arg[1].charAt(0);

 llsString = algLLS( w, g, cGroup);

 System.out.println("Starting word wg is "
                     + w + g);
 System.out.println("Reduction is        "
                     + llsString);
}

}
```

B.5 Testing classes

We include here some of the classes we used to run tests on the algorithms. The first class was used to compute Table 5.1.

```
/*
 * This class is to run tests to reduce random words with
 * the slex algorithm
 *
 * It prints the word and the slex reduction from the
 * algorithm.
 */

/* Author : Renaud Brien
 * Created : 04/03/12
 * Last update: 11/03/12
 */

import java.util.*;

public class SLexTest {

  private static String freeRed( String word ) {
    int n = word.length();
    String w = "";

    if (word == null || word.equals("") )
      return word;

    w = w + word.charAt(0);

    for (int i = 1 ; i < n ; i++) {

      char g = word.charAt(i);

      if ( w.length() == 0 ){
        w = w + g;
      }
      else {
        char lastOfW = w.charAt(w.length()-1);

        if ( (NormFormLetter.sameName(lastOfW, g))
              && (NormFormLetter.sign(g) !=
                  NormFormLetter.sign(lastOfW) ) ) {

          w = w.substring( 0, w.length()-1 );
        }
        else {
          w = w + g;
        }
      }
    }
    return w;
  }

  /* arg has to look like : n m lenght k */
  /* 0 < n < 27, m > 2, length >= 0, k > 0 */
  public static void main ( String[] arg ) {

    int n = Integer.parseInt(arg[0]);
    int m = Integer.parseInt(arg[1]);
    int length = Integer.parseInt(arg[2]);
    int k = Integer.parseInt(arg[3]);

    NFCoxeterGroup cGroup = new NFCoxeterGroup( n, m);

    String s;
    String red, red2, fRed;
    int totalFreeRed = 0, totalAlgRed = 0,
        totalRed = 0;
    int test = 0;
    int l;
    int totalFreeRed2 = 0, totalAlgRed2 = 0,
        totalRed2 = 0;

    for (int i = 0 ; i < k ; i++) {
      s = "";
      red = "";
      red2 = "";
      fRed = "";
      for (int j = 0 ; j < length ; j++) {
        l = (int) (Math.random()*(2*n) );
        if( l % 2 == 0) {
          s = s + (char)(97 + (l/2));
        }
        else {
          s = s + (char)(65 + ((l-1)/2));
        }
      }

/* Test 1 : Words
 * Uncomment this section to get the standard test.
 * It prints the word and its reduction with the
 * length of the words */

/*
      test = 1;
      red = SLexAlg.slexAlg( s , cGroup);

      System.out.println( "   " + s + "  " + i +
                          "\n-> " + red );
      System.out.println( "Start length = " + s.length()
             + " , Reduced length = " + red.length());
      //System.out.println( "Well reduced = " + equal );
/*  */

/* Test 2 : Lengths, with words
 * Uncomment this section to get information on how
 * much non-free reduction is done by the algorithm.
 * It prints the word, its free reduction and the
 * reduction with the length of each words */

/*
      test = 2;
      fRed = freeRed(s);
      red = SLexAlg.slexAlg( fRed , cGroup);

      System.out.println( "   " + s + "  " + i +
                      "\n=> " + fRed + "\n-> " + red );
      System.out.println( "Start length = " + s.length()
            + " , Free Reduced length = " + fRed.length()
            + " , Reduced length = " + red.length() );
/*  */

/* Test 3 : Lengths
 * Uncomment this section to get information on how
 * much non-free reduction is done by the algorithm,
 * whithout printing the words. It prints the length
 * of the starting word, the free reduction and the
 * reduction. It also prints the total length reduction,
 * the length lost by free reduction and that lost
 * by the RRSAlg.*/

/*
      test = 3;
      fRed = freeRed(s);
      red = SLexAlg.slexAlg( fRed , cGroup);

      int freeRed = s.length() - fRed.length();
      int algRed = fRed.length() - red.length();
      int totalRed = s.length() - red.length();

      System.out.println( s.length() + "  " +
                fRed.length() + "  " + red.length());
      System.out.println( totalRed + "  " +
                freeRed + "  " + algRed + "\n");
//      System.out.println( "Start length = " + s.length() +
//            "\nFree Reduced length = " + fRed.length()
//          + "\nReduced length = " + red.length() +"\n");
```

```
/*  */

/* Test 4 : Succesive reduction test
 * Uncomment this part to run the test to see
 * if the reduction works well. It works by passing the
 * reduced word to the algoritm again and it prints
 * the words that fails the equality test. */

/*
      test = 4;
      red = SLexAlg.slexAlg( s , cGroup);
      red2 = SLexAlg.slexAlg( red, cGroup);
      boolean equal = red.equals(red2);
      if ( equal == false ){
       System.out.println( "   " + s + "  " + i);
       System.out.println("\n-> " + red );
       System.out.println( "=> " + red2);
       System.out.println( "Start length = " + s.length()
               + " , Reduced length = " + red.length() );
      }
/* */

/* Test 5 : Lengths Statistics
 * Uncomment this section to get statistics on free
 * and non-free reduction done by the algorithm, whithout
 * printing the words. */

/*  */

      test = 5;
      fRed = freeRed(s);
      red = SLexAlg.slexAlg( fRed , cGroup);

      int freeRed = s.length() - fRed.length();
      int algRed = fRed.length() - red.length();
      int bothRed = s.length() - red.length();

      totalFreeRed = totalFreeRed + freeRed;
      totalAlgRed = totalAlgRed + algRed;
      totalRed = totalRed + bothRed;

      totalFreeRed2 = totalFreeRed2 + (freeRed*freeRed);
      totalAlgRed2 = totalAlgRed2 + (algRed * algRed);
      totalRed2 = totalRed2 + (bothRed * bothRed);

//      System.out.println( s.length() + "  "
//             + fRed.length() + "  " + red.length());
//      System.out.println( bothRed + "  " + freeRed +
//                      "  " + algRed + "\n");
//      System.out.println( "Start length = " + s.length()
//           + "\nFree Reduced length = " + fRed.length()
//           + "\nReduced length = " + red.length() +"\n");

/*  */
    }
    if (test == 5) {
     double statFreeRed = ((double)totalFreeRed/
                             (k*length))*100;
     double statAlgRed = ((double)totalAlgRed/
                            (k*length))*100;
     double statTotalRed = ((double)totalRed/
                              (k*length))*100;

     double varFreeRed = ((((double)totalFreeRed2/
                              (length*length))/k) -
            (double)(statFreeRed*statFreeRed))*(k/(k-1));

     double varAlgRed = ((((double)totalAlgRed2/
                             (length*length))/k) -
            (double)(statAlgRed*statAlgRed))*(k/(k-1));

     double varTotalRed = ((((double)totalRed2/
                               (length*length))/k) -
           double)(statTotalRed*statTotalRed))*(k/(k-1));

     System.out.println( "Word length = " + length
                        + " , k = " + k );

//     System.out.println( totalRed + "  " +
//                totalFreeRed + "  " + totalAlgRed + "\n");

  System.out.println("Free reduction reduces length by about : "
                       + statFreeRed + " with Variance "
                       + varFreeRed);
  System.out.println("RRS reduction reduces length by about : "
                       + statAlgRed + " with Variance "
                       + varAlgRed );
  System.out.println("Total lenght is reduced by about : "
                       + statTotalRed + " with Variance "
                       + varTotalRed );
    }

  }

}
```

This class was used to build Tables 5.2 and 5.3

```
/*
 * This class is used to look at how much
 * "scrambling" is done on when applied to
 * a product of two reduced words.
 */

/* Author : Renaud Brien
 * Created : 12/03/12
 * Last update: 04/03/12
 */

public class SLexTest4 {

  private static String freeRed( String word ){
    int n = word.length();
    String w = "";

    if (word == null || word.equals("") )
      return word;

    w = w + word.charAt(0);

    for (int i = 1 ; i < n ; i++) {

      char g = word.charAt(i);

      if ( w.length() == 0 ){
        w = w + g;
      }
      else {
        char lastOfW = w.charAt(w.length()-1);

        if ( (NormFormLetter.sameName(lastOfW, g))
              && ( NormFormLetter.sign(g)
                     != NormFormLetter.sign(lastOfW))){

          w = w.substring( 0, w.length()-1 );
        }
        else {
          w = w + g;
        }
      }
    }
    return w;
  }

  public static String randWord ( int n, int length ) {

    String s = "";
    int l;

    for (int j = 0 ; j < length ; j++) {
        l = (int) (Math.random()*(2*n) );
        if( l % 2 == 0) {
          s = s + (char)(97 + (l/2));
        }
```

```
        else {
          s = s + (char)(65 + ((l-1)/2));
        }
      }
    return s;
  }

  /* arg has to look like : n m lenght k */
  /* 0 < n < 27, m > 2, length >= 0, k > 0
   * length is the wanted minimal length
   * of the reduced words.*/
  public static void main ( String[] arg ) {

    int nb = Integer.parseInt(arg[0]);
    int m = Integer.parseInt(arg[1]);
    int length = Integer.parseInt(arg[2]);
    int k = Integer.parseInt(arg[3]);

    double leftRatio = 0;
    double rightRatio = 0;

    double leftRatio2 = 0;
    double rightRatio2 = 0;

    int length2 = length + (int)(length * (0.5)) + 1;

    NFCoxeterGroup cGroup = new NFCoxeterGroup( nb, m);

    String w, s, ws;
    int l;
    int sameStartFor = 0, sameEndFor = 0;


    for (int j = 0 ; j < k ; j++) {
      sameStartFor = 0;
      sameEndFor = 0;
      w = "";
      s = "";
      ws = "";

      /* Making w */
      do {
        String rand = SLexTest4.randWord(nb, length2);
        w = SLexAlg.slexAlg(rand,cGroup);
      } while ( w.length() < length );

      w = w.substring(0,length);

      /* Making s */
      do {
        String rand = SLexTest4.randWord(nb, length2);
        s = SLexAlg.slexAlg(rand,cGroup);
      } while ( s.length() < length );

      s = s.substring(0,length);

      ws = w + s;
      ws = SLexAlg.slexAlg(ws, cGroup);

      int i = 0;

      while( i < w.length() && i < ws.length()
              && w.charAt(i) == ws.charAt(i) ){
        sameStartFor++;
        i++;
      }

      i = 0;
      int sEnd = s.length() - 1;
      int wsEnd = ws.length() - 1;
      while( i < s.length() && i < ws.length()
              && s.charAt(sEnd) == ws.charAt(wsEnd) ){
        sameEndFor++;
        i++;
        sEnd--;
        wsEnd--;
      }

/*      System.out.println( "   " + w + " "
                          + s + "\n-> " + ws );
      System.out.println( "Lengths:  " + w.length()
                          + "  " + s.length() + "  "
                          + ws.length());
      System.out.println( "Same for: "
                          + sameStartFor + "  "
                          + sameEndFor + "\n");
/* */

      double currentRatioR =
        ((double)sameStartFor / (double)w.length());
      double currentRatioL =
        ((double)sameEndFor / (double)s.length());

      rightRatio = rightRatio
        + currentRatioR;
      leftRatio = leftRatio
        + currentRatioL;

      rightRatio2 = rightRatio2
        + (currentRatioR*currentRatioR);
      leftRatio2 = leftRatio2
        + (currentRatioL*currentRatioL);

    }

    rightRatio = rightRatio / k;
    leftRatio = leftRatio / k;

    double varRight =
      ((rightRatio2/k) -
       (rightRatio*rightRatio))*(double)(k/(k-1));
    double varLeft =
      ((leftRatio2/k) -
       (leftRatio*leftRatio))*(double)(k/(k-1));

    System.out.println( "Ratio of untouched w in ws = "
                        + rightRatio
                        + " with Variance " + varRight);
    System.out.println( "Ratio of untouched s in ws = "
                        + leftRatio
                        + " with Variance " + varLeft);

  }

}
```

This class was used to obtain examples on the conjugacy problem

```
/*
 * This class is used to look at how much
 * "scrambling" is done on when applied to a
 * product of tree reduced words.
 * We try to find as much as we can from awb,
 * knowing only w.
 */

/* arg for main has to look like : n m lenght k */

/* Author : Renaud Brien
 * Created : 12/03/12
 * Last update: 04/03/12
 */

public class SLexTest5 {

  private static String freeRed( String word ) {
    int n = word.length();
    String w = "";

    if (word == null || word.equals("") )
      return word;

    w = w + word.charAt(0);
```

```
  for (int i = 1 ; i < n ; i++) {

    char g = word.charAt(i);

    if ( w.length() == 0 ){
      w = w + g;
    }
    else {
      char lastOfW = w.charAt(w.length()-1);

      if ( (NormFormLetter.sameName(lastOfW, g))
            && ( NormFormLetter.sign(g) !=
                NormFormLetter.sign(lastOfW) ) ) {

        w = w.substring( 0, w.length()-1 );
      }
      else {
        w = w + g;
      }
    }
  }
  return w;
}

public static String randWord ( int n, int length ) {

  String s = "";
  int l;

  for (int j = 0 ; j < length ; j++) {
      l = (int) (Math.random()*(2*n) );
      if( l % 2 == 0) {
        s = s + (char)(97 + (l/2));
      }
      else {
        s = s + (char)(65 + ((l-1)/2));
      }
    }
  return s;
}

public static String randRedWord( int n, int length,
                              NFCoxeterGroup cGroup ){

  int length2 = length + (int)(length *
                        ((double)1/(double)n)) + 1;
  String w = "";

  do {
    String rand = SLexTest5.randWord(n, length2);
    w = SLexAlg.slexAlg(rand,cGroup);
  } while ( w.length() < length );

  return w;
}

public static void attackGame (int number, int mValue,
                    int lengthValue, int sampleSize ){

  int nb = number;
  int m = mValue;
  int length = lengthValue;
  int k = sampleSize;

  int length2 = length + (int)(length *
                      ((double)1/(double)nb)) + 1;

  NFCoxeterGroup cGroup = new NFCoxeterGroup( nb, m);

  String w, a, b, awb;
  int l;
  int sameStartFor = 0, sameEndFor = 0;

  for (int j = 0 ; j < k ; j++) {
    sameStartFor = 0;
    sameEndFor = 0;
    w = "";
    a = "";
    b = "";
    awb = "";

    /* Making w */
    do {
      String rand = SLexTest5.randWord(nb, length2);
      w = SLexAlg.slexAlg(rand,cGroup);
    } while ( w.length() < length );

    /* Making a */
    do {
      String rand = SLexTest5.randWord(nb, length2);
      a = SLexAlg.slexAlg(rand,cGroup);
    } while ( a.length() < length );

    /* Making b */
    do {
      String rand = SLexTest5.randWord(nb, length2);
      b = SLexAlg.slexAlg(rand,cGroup);
    } while ( b.length() < length );

    awb = a + w + b;
    awb = SLexAlg.slexAlg(awb, cGroup);

    String random = SLexTest5.randRedWord( nb,
                              awb.length(), cGroup );
    random = random.substring(0,awb.length());

    double coin = Math.random();

    if (coin < 0.5) {
      System.out.println( " awb " );
      System.out.println( "\n\n\n\n\n\n\n\n\n\n\n\n" );
      System.out.println( "w =   " + w
                        + "\nawb = " + awb );
    }
    else {
      System.out.println( " ran " );
      System.out.println( "\n\n\n\n\n\n\n\n\n\n\n\n" );
      System.out.println( "w =   " + w
                        + "\nawb = " + random );
    }
  }
}

/* arg has to look like : n m lenght k */
/* 0 < n < 27, m > 2, length >= 0, k > 0
 * length is the wanted minimal length
 * of the reduced words.*/
public static void main ( String[] arg ) {

  int nb = Integer.parseInt(arg[0]);
  int m = Integer.parseInt(arg[1]);
  int length = Integer.parseInt(arg[2]);
  int k = Integer.parseInt(arg[3]);

  int length2 = length + (int)(length *
                  ((double)1/(double)nb)) + 1;

  NFCoxeterGroup cGroup = new NFCoxeterGroup( nb, m);

  String w, a, b, awb;
  int l;
  int sameStartFor = 0, sameEndFor = 0;


  for (int j = 0 ; j < k ; j++) {
    sameStartFor = 0;
    sameEndFor = 0;
    w = "";
    a = "";
    b = "";
    awb = "";

    /* Making w */
    do {
      String rand = SLexTest5.randWord(nb, length2);
      w = SLexAlg.slexAlg(rand,cGroup);
    } while ( w.length() < length );

    /* Making a */
    do {
      String rand = SLexTest5.randWord(nb, length2);
      a = SLexAlg.slexAlg(rand,cGroup);
    } while ( a.length() < length );

    /* Making b */ /*
    do {
      String rand = SLexTest5.randWord(nb, length2);
      b = SLexAlg.slexAlg(rand,cGroup);
```

```
} while ( b.length() < length );

/* Making a^{-1} */
b = "";
for(int i = 0; i< a.length() ; i++){
  b = NormFormLetter.inverse(a.charAt(i)) + b;
}

awb = a + w + b;
awb = SLexAlg.slexAlg(awb, cGroup);

int i = 0;

/*
while( i < w.length() && i < ws.length()
       && w.charAt(i) == ws.charAt(i) ){
  sameStartFor++;
  i++;
}

i = 0;
int sEnd = s.length() - 1;
int wsEnd = ws.length() - 1;
while( i < s.length() && i < ws.length()
       && s.charAt(sEnd) == ws.charAt(wsEnd) ){
  sameEndFor++;
  i++;
  sEnd--;
  wsEnd--;
}

*/
System.out.println( "a = " + a + "\nb = " + b );
System.out.println( "\n\n\n\n\n\n\n\n\n\n\n\n" );

System.out.println( "w =   " + w
                    + "\nawb = " + awb );
//System.out.println( "Lengths:  " + w.length()
//        + "  " + s.length()
//        + "  " + ws.length());
//System.out.println( "Same for: " + sameStartFor
//        + "  " + sameEndFor + "\n");

    }
  }

}
```

Bibliography

[AAG99] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.

[All02] Daniel Allcock. Braid pictures for Artin groups. *Trans. Amer. Math. Soc.*, 354(9):3455–3474 (electronic), 2002.

[Art47] E. Artin. Theory of braids. *Ann. of Math. (2)*, 48:101–126, 1947.

[BB05] Anders Björner and Francesco Brenti. *Combinatorics of Coxeter groups*, volume 231 of *Graduate Texts in Mathematics*. Springer, New York, 2005.

[Cho48] Wei-Liang Chow. On the algebraical braid group. *Ann. of Math. (2)*, 49:654–658, 1948.

[CKL+01] Jae Choon Cha, Ki Hyoung Ko, Sang Jin Lee, Jae Woo Han, and Jung Hee Cheon. An efficient implementation of braid groups. In *Advances in cryptology—ASIACRYPT 2001 (Gold Coast)*, volume 2248 of *Lecture Notes in Comput. Sci.*, pages 144–156. Springer, Berlin, 2001.

[ECH+92] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups.* Jones and Bartlett Publishers, Boston, MA, 1992.

[ERM94] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.

[FGM03] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and Garside groups. *J. Algebra*, 266(1):112–132, 2003.

[Gar69] F. A. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser. (2)*, 20:235–254, 1969.

[HR11] Derek F. Holt and Sarah Rees. Artin groups of large type are shortlex automatic with regular geodesics. *Proc. London Math. Soc.*, page 27, 2011.

[Hum90] James E. Humphreys. *Reflection groups and Coxeter groups*, volume 29 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1990.

[KLC+00] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-Sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000 (Santa Barbara, CA)*, volume 1880 of *Lecture Notes in Comput. Sci.*, pages 166–183. Springer, Berlin, 2000.

[KT08] Christian Kassel and Vladimir Turaev. *Braid groups*, volume 247 of *Graduate Texts in Mathematics*. Springer, New York, 2008. With the graphical assistance of Olivier Dodane.

[Mih58] K. A. Mihaĭlova. The occurrence problem for direct products of groups. *Dokl. Akad. Nauk SSSR*, 119:1103–1105, 1958.

[MM06] Jean Mairesse and Frédéric Mathéus. Growth series for Artin groups of dihedral type. *Internat. J. Algebra Comput.*, 16(6):1087–1107, 2006.

[MSU08] Alexei Myasnikov, Vladimir Shpilrain, and Alexander Ushakov. *Group-based cryptography*. Advanced Courses in Mathematics. CRM Barcelona. Birkhäuser Verlag, Basel, 2008.