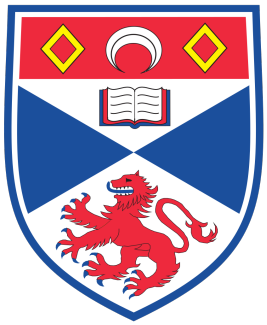
University of St Andrews

Solving Equations over Free Groups

Author:
Daphne Bogosian

Supervisors:
Dr Christopher Jefferson
Dr Markus Pfeiffer
Prof Colva Roney-Dougal

April 12, 2019



Contents

Abstract

This project will focus on finding solutions to linear and quadratic equations over freely presented groups. A group is freely presented if there are no relations over its generators other than those between an element and its inverse. An equation (of elements of a freely presented group) is linear if every variable occurs exactly 0 or 1 times in the equation, and quadratic if every variable occurs exactly 0 or 2 times in the equation. This project will focus on determining whether or not such equations have solutions by representing the equation as a constraint problem and either using a custom solver or an existing one to search for a solution. Extension aims include describing the full, possibly infinite, family of solutions to an equation in a formal language and considering more general equations over a free group.

1 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 15,094 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

2 Introduction

This paper is split into three main sections: a section of mathematical context which explains the background necessary to understand the problem, a section detailing the algorithm for finding solutions to linear equations, and a section describing the constraint solver and its efficacy on quadratic equations. The mathematical background is compiled from various sources — from textbooks on abstract algebra [10] [12] papers on the topic of equations over free groups [2] [8] [9] [7]. The algorithm detailed in section 5 and the constraint solver in section 6 are completely original to this paper and were developed with the assistance of Dr Markus Pfeiffer and Dr Christopher Jefferson, respectively, at the University of St Andrews.

There are numerous sources of literature on equations over free groups proving that the problem is solvable for certain classes of equations [2], that the solvability of certain types of problems is NP-Complete [7], and that the size of a solution is bounded [9]. But there are few examples of this literature being practically applied to software that can find solutions to specific problems.

This project focuses on linear and quadratic equations. Quadratics appear in various areas of mathematics such as JSJ-decompositions of groups, group actions of dynamical systems, and algrebraic geometry over groups [9] though none of these applications are the focus of this paper. There are fewer obvious applications of linear equations, but they are simpler to solve and serve as a baseline to which the constraint solver can be compared. Solutions to general equations over free groups have applications in finding isomorphisms between finitely presented groups. [1]

Section 5 contains the proof that every linear equation has a solution, comprehensively describes an algorithm that finds the full set of interesting solutions to a linear equation, and then proves that this algorithm is correct. Section 6 describes the constraints that model a solvable equation in a constraint solver, and proves that a quadratic equation has a solution if and only if a constraint solver can find an assignment of values satisfying these constraints. Both the algorithm and the constraint solver are fully implemented, as a GAP package and a Python package respectively, and can be run using the documentation provided in section 7.

3 Requirement Specification

The following goals were outlined at the time of conception of this project.

1 Produce a report that clearly defines the mathematical background of the problem and accurately and understandably describes the process of building the solution and how it works.

2 Produce software that takes in human-readable mathematical linear equations over a freely presented group and outputs the corresponding constraint model.

3 Produce software that takes in human-readable mathematical quadratic equations over a freely presented group and outputs the corresponding constraint model.

4 Empirically and theoretically analyze of the time complexity of the algorithms used in the aforementioned software.

Extensions/optional aims:

5 Create empirical evidence that improvements to the solver, using properties of freely presented groups and the equations, has improved the time complexity of the constraint modelling algorithm.

6 Produce software that generalizes the set of solutions to an equation into a formal language.

Note that the linear equation solver outlined in section 5 is additional to the above objectives, and that the single constraint solver described in section 6 satisfies both objectives 2 and 3.

4 Context

This section describes mathematical concepts surrounding the topic of this paper: equations over freely presented groups. This includes an in-depth introduction to free groups, presentations, and equations over the former. The definition of a group is integral to these topics.

Definition 4.1. A *group* is a nonempty set G together with a binary operation $*$ that together satisfy the following properties:

- *Closure:* $a * b \in G$ for all $a, b \in G$.
- *Associativity:* $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$.
- *Identity:* there exists an element 1_G in G such that $a * 1_G = 1_G * a = a$ for all $a \in G$.
- *Inverses:* for every element $a \in G$, there exists an element $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = 1_G$. [12]

To understand free groups and freely presented groups, the reader must first be familiar with groups. It is assumed the reader is already acquainted with the definition of a group, as well as with associated concepts, such as homomorphisms, isomorphisms, images, kernels, subgroups, normal subgroups, generators, and related results. See *An Introduction to the Theory of Groups* by Rotman [12] or *A Course in the Theory of Groups* by Robinson [10] for background in these areas.

4.1 Free Groups

This project is concerned with equations over freely presented groups. However, in order to define what it means to be *freely presented* it is first necessary to define what it means for a group to be *free*. In this section we will define freeness, but first we must introduce the necessary terminology.

Definition 4.2. An *alphabet* is a finite set A. The elements of A are called *letters*. Define a set A^{-1} such that $A^{-1} = \{a^{-1} : a \in A\}$. Call the element a^{-1} the *inverse* of a. A sequence of elements from $A \cup A^{-1}$ is called a *word* over A. The empty word (or empty sequence of letters) will be denoted ϵ. A word in A is said to be reduced if it does not contain any two consecutive letters of the form aa^{-1} or $a^{-1}a$ for any $a \in A$. [12]

Letters will be represented with lowercase letters from the Latin alphabet. Words, reduced or otherwise, will be represented with lowercase letters from the Greek alphabet. The operation of appending a sequence of letters β to the end of a sequence of letters α will be called the *concatenation* of α with β and will be denoted $\alpha\beta$. The operation of replacing a sequence of letters of the form aa^{-1} or $a^{-1}a$ for any a in an alphabet A with the empty sequence ϵ will be called *reduction*. The operation of repeated reduction of a word α until there are no subsequences of letters left to be reduced will be denoted by $[\alpha]$.

Lemma 4.1. *Let A be an alphabet and let α be a word over A. Then $[\alpha]$ is unique.*

Proof. For a word α to have different possible reductions, the order in which each aa^{-1} or $a^{-1}a$ sequence is replaced with ϵ must change the result of the overall reduction (as ultimately all such pairs must be removed for the word to be reduced). There are two cases in which there is a choice in the order of the reduction:

Case 1: the word has the form $a_1a_1^{-1}b_1b_2 \dots b_na_2a_2^{-1}$ where $b_1 \neq a_1$ and $b_n \neq a_2^{-1}$. Then either $a_1a_1^{-1}$ or $a_2a_2^{-1}$ can be replaced with ϵ first. Whichever occurs first, and the other second, the end result is still the same: $b_1 \dots b_n$.

Case 2: the word has the form $b_1b_2 \dots aa^{-1}a \dots b_n$. Then there are two options: replace aa^{-1} with ϵ or replace $a^{-1}a$ with ϵ. Both the former and the latter option result in the reduced word $b_1b_2 \dots a \dots b_n$, so again the order the reduction is performed in does not change the resulting reduced form. □

Lemma 4.2. *Let α and β be words over an alphabet A. Then $[[\alpha][\beta]] = [\alpha\beta]$.*

Proof. Reducing the word $[\alpha][\beta]$ is one way of reducing the word $\alpha\beta$. Since by lemma 4.1, the reduction of any word is unique, $[\alpha][\beta] = [\alpha\beta]$. □

Lemma 4.3. *Let α, β, γ, and δ be words over an alphabet A such that $[\alpha] = [\gamma]$ and $[\beta] = [\delta]$. Then $[\alpha\beta] = [\gamma\delta]$, i.e. the operation of concatenation followed by repeated reduction is well defined.*

Proof. Since $[\alpha] = [\gamma]$ and $[\beta] = [\delta]$, it follows that $[\alpha][\beta] = [\gamma][\delta]$. So by Lemma 4.1, $[[\alpha][\beta]] = [[\gamma][\delta]]$ and by Lemma 4.2, applied to both sides of the equation, $[\alpha\beta] = [\gamma\delta]$. □

These concepts prepare us to define the property of *freeness* and prove the existence of a group with such a property.

Definition 4.3. Let F be a group, let X be a non-empty subset of F, and let $\iota : X \to F$ be an injective mapping. Then F is *free with basis X* (with respect to ι) if for every group G and every mapping $f : X \to G$ there exists a unique homomorphism $\phi : F \to G$ such that $\iota\phi = f$. [10]

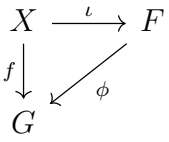


Figure 1: A visual representation of the relationships between X, a free group F with basis X, and an arbitrary group G.

In other words, for all mappings from a set X to a group G, an element of X can be mapped to an element of the group F, which then maps to the same element of G as in the mapping from X to G. So, the elements $(x)\iota$ for every $x \in X$ are a relabeling of the elements of X as elements of F. Then the group F is generated from these elements and for every $x \in X$, $(x)f = ((x)\iota)\phi$.

However, it remains to be proved that such a group F exists.

Theorem 4.1. *The set $F(X)$ of reduced words over an alphabet X under the operation of concatenation followed by repeated reduction is free with basis X and is called the free group with basis X.*

Proof. There are two things to prove: that $F(X)$ is a group and that $F(X)$ is free. We start with the former.

Closure: let $\alpha, \beta \in F(X)$. Then $[\alpha\beta]$ must be a fully reduced word over the alphabet A by the nature of the operation. So $F(X)$ is closed under this operation.

Identity: take any $\alpha \in F(X)$. Then $[\epsilon\alpha] = [\alpha\epsilon] = [\alpha] = \alpha$ since α is by definition already reduced. So the empty word ϵ forms the identity of $F(X)$.

Inverses: take any $\alpha = a_1 a_2 \dots a_n \in F(X)$. Then

$$\begin{aligned}[a_1 a_2 \dots a_n a_n^{-1} a_{n-1}^{-1} \dots a_1^{-1}] &= [a_n^{-1} \dots a_2^{-1} a_1^{-1} a_1 a_2 \dots a_n] \\ &= [\epsilon] = \epsilon.\end{aligned}$$

So $a_n^{-1} \dots a_2^{-1} a_1^{-1}$ is the inverse of the element α.

Associativity: let $\alpha, \beta, \gamma \in F(X)$. Then $[[\alpha\beta][\gamma]] = [\alpha\beta\gamma] = [[\alpha][\beta\gamma]]$ since by Lemma 4.1, the reduction of any given word is unique.

Now we must prove that $F(X)$ is free. Let $\iota : X \to F(X)$ be the inclusion mapping from X into $F(X)$, i.e. the function defined by $(a)\iota = a$ for all $a \in X$. Let $f : X \to G$ be a function from X to any group G. Define a mapping ϕ from $F(X)$ to G by mapping $a_1^{k_1} \dots a_n^{k_n} \to (a_1)f^{k_1} \dots (a_n)f^{k_n}$ where $k_1, \dots, k_n \in \{-1, 1\}$ and $a_1, \dots, a_n \in X$.

If $\alpha = a_1^{k_1} \dots a_n^{k_n}$ and $\beta = b_1^{l_1} \dots b_m^{l_m} \in F(X)$ for some $k_1, \dots, k_n, l_1, \dots, l_m \in \{-1, 1\}$ then

$$\begin{aligned}(\alpha\beta)\phi &= (a_1)f^{k_1} \dots (a_n)f^{k_n}(b_1)f^{l_1} \dots (b_m)f^{l_m} \\ &= (\alpha)\phi(\beta)\phi.\end{aligned}$$

So ϕ is a homomorphism.

What remains is to show that ϕ is unique. Suppose there exists another homomorphism $\gamma : F(X) \to G$ such that $\iota\gamma = f$. By showing that $(\alpha)\phi = (\alpha)\gamma$ for all $\alpha \in F(X)$, we prove that $\gamma = \phi$, and so ϕ is unique.

Now γ is such that $((a)\iota)\gamma = (a)f$ for all $a \in X$. So $\iota\gamma = \iota\phi$ (since $(a)f = (a)\phi$). So for all $a \in \operatorname{im}\iota$ it holds that $(a)\gamma = (a)\phi$. In other words, ϕ and γ map the elements of $\operatorname{im}(\iota)$ to the same values.

Also, $F(X)$ is generated by the image of ι, i.e. $F(X) = \langle \operatorname{im}\iota \rangle$. So γ and ϕ agree on all terms of their domains, i.e. $(\alpha)\phi = (\alpha)\gamma$ for all $\alpha \in F(X)$, and so $\gamma = \phi$. Therefore ϕ is unique and $F(X)$ is free. [10] [12] □

This proves that at least one free group exists for every set X, but it is also possible for other groups besides the group of reduced words on X to be free.

Example 4.1. Let $X = \{a\}$ and consider the group of integers $\mathbb{Z}$ under the operation $+$. Define a map $\iota : X \to \mathbb{Z}$ by $(a)\iota = 1$. Let f be any function from X to a group G, such that $(a)f = g$ for some $g \in G$. Define a function $\phi : \mathbb{Z} \to G$ by $(x)\phi = g^x$ for $x \in \mathbb{Z}$.

We can show that ϕ is a group homomorphism: let $x, y \in \mathbb{Z}$. Then

$$(x + y)\phi = g^{x+y} = g^x g^y = (x)\phi(y)\phi.$$

Also, ϕ is the only group homomorphism such that $((1)\iota)\phi = (1)\phi = (1)f = g$, since if there were some other function $\gamma : \mathbb{Z} \to G$ such that $\iota\gamma = f$, then they would have the same values when evaluated on every element of $\operatorname{im}(\iota) = \{1\}$. However, $\mathbb{Z}$ is generated by $\operatorname{im}(\iota)$, so γ and ϕ must agree on every element of $\mathbb{Z}$ and hence are the same mapping.

Therefore, $\mathbb{Z}$ is free (with respect to ι) with basis X.

Note that the choice of basis is very important: while $\mathbb{Z}$ is free on the basis X given above, it is not necessarily free on other generating sets. This is demonstrated in example 4.6.

But if it is possible for other groups to be free, why is the group of reduced words $F(X)$ defined as *the* free group on X? Because all groups that are free on a specific set X are actually isomorphic. This can be shown directly using definition 4.3 of freeness.

Theorem 4.2. *Up to isomorphism, there is at most one free group on a set X.*

Proof. Suppose that F_1 and F_2 are both groups that are free with basis X with respect to the mappings $\iota_1 : X \to F_1$ and $\iota_2 : X \to F_2$ respectively. Since F_1 is free, there must exist a unique homomorphism $\phi_2 : F_1 \to F_2$ (since F_2 is a group) such that $\iota_1\phi_2 = \iota_2$. Also, since F_2 is free, there must exist a unique homomorphism $\phi_1 : F_2 \to F_1$ such that $\iota_2\phi_1 = \iota_1$. So

$$\iota_1 = \iota_2\phi_1 = \iota_1\phi_2\phi_1$$

Since F_1 is free, the identity mapping id_{F_1} is the unique homomorphism such that $\iota_1 \mathrm{id}_{F_1} = \iota_1$. Hence $\phi_2\phi_1 = \mathrm{id}_{F_1}$. Following a symmetric argument for F_2, we can show that $\phi_1\phi_2 = \mathrm{id}_{F_2}$, the identity mapping of F_2, and so ϕ_1 and ϕ_2 are mutual inverses, and also must be isomorphisms. [10] □

Since there can only be one free group on any given set X, and the group of reduced words $F(X)$ is always free on X, to show that any group G is free it suffices to simply show that it is isomorphic to a group of reduced words over an alphabet.

Example 4.2. Again, consider $(\mathbb{Z}, +)$. Let $F(X)$ be the group of reduced words over the set $X = \{a\}$. So $F(X)$ is generated by a. Also, $\mathbb{Z}$ is generated by 1. It is possible to show that $\mathbb{Z}$ is free by showing that it is isomorphic to $F(X)$.

Define a function $f : \mathbb{Z} \to F(X)$ by $x \mapsto a^x$ (note $a^0 = \epsilon$). Then, if $x, y \in \mathbb{Z}$

$$(x + y)f = a^{x+y} = a^x a^y = (x)f(y)f$$

And so f is a homomorphism. Now choose $x, y \in \mathbb{Z}$ such that $(x)f = (y)f$. Then

$$a^x = a^y \implies a^{x-y} = \epsilon \implies x - y = 0 \implies x = y$$

since there does not exist an $n \in \mathbb{Z} \setminus \{0\}$ such that $\alpha^n = \epsilon$ in $F(X)$. So f is injective. Also, every element of $F(X)$ has the form a^n where $n \in \mathbb{Z}$, so f is also surjective. Therefore, f is an isomorphism.

Before the next example, we must first define a group action.

Definition 4.4. If G is a group and X is a set, then a group action . of G on X is a function $. : G \times X \to G$ that satisfies the following two axioms:

- $x.1_G = x$ for all $x \in X$.
- $x.(gh) = (x.g).h$ for all $g, h \in G$ and $x \in X$. [12]

A group G acts on a set X if there exists a group action of G on X. In other words, the group G permutes the elements of X, with the identity of G mapping any value x to itself.

Example 4.3. Consider the group generated by the following integer matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, \; B = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$$

The group generated by A and B under the operation of matrix multiplication forms a subgroup of $SL(2, \mathbb{Z})$, the group of 2 by 2 matrices over $\mathbb{Z}$ with determinant 1. This group is, in fact, free. To show this we must find an isomorphism ϕ from a free group of some basis to the group $\langle A, B \rangle$.

Let $X = \{a, b\}$ and let ι be the inclusion mapping from X to $F(X)$. Define a function $\phi : F(X) \to \langle A, B \rangle$ by mapping $a \mapsto A$ and $b \mapsto B$. If this function is an isomorphism, then $\langle A, B \rangle$ is free.

Now ϕ is a surjective homomorphism by the way it is defined to map onto $\langle A, B \rangle$ since $a^{k_1} b^{l_1} \dots a^{k_n} b^{l_n}$ under ϕ maps to the arbitrary matrix $A^{k_1} B^{l_1} \dots A^{k_n} B^{l_n}$ generated by A and B. All that remains is to show it is injective.

ϕ is injective if and only if $\ker \phi = \{\epsilon\}$. So for any nontrivial word $\alpha \in F(X)$ we must show that $(\alpha)\phi \neq I$, the 2 by 2 identity matrix and identity element of $\langle A, B \rangle$.

We will first establish useful facts about $\langle A \rangle$ and $\langle B \rangle$, and the way arbitrary elements of each affect arbitrary elements of certain disjoint subsets of $\mathbb{Z}^2$. Then, we take $M = (\alpha)\phi$ to be an arbitrary element of $\langle A, B \rangle$ and K to be an arbitrary element of one of these disjoint subsets. By showing that the product M with K sends K from one subset to a different disjoint subset, we prove that $M \neq I$, and therefore ϕ is an isomorphism.

First note that matrix multiplication of a 1 by 2 matrix with elements from $\mathbb{Z}^2$ with a 2 by 2 matrix from $\langle A, B \rangle$ is a group action. Since both $\langle A \rangle$ and $\langle B \rangle$ are subgroups of $\langle A, B \rangle$, they also act on $\mathbb{Z}^2$ under the same operation.

Next we show by two induction arguments, one each for the cases $n > 0$ and $n < 0$, that for $n \in \mathbb{Z}$,

$$A^n = \begin{bmatrix} 1 & 2n \\ 0 & 1 \end{bmatrix} \text{ and } B^n = \begin{bmatrix} 1 & 0 \\ 2n & 1 \end{bmatrix}.$$

By induction on n, for when $n > 0$: start with a base case of $n = 1$. Then it is easy to show that the condition holds for both A and B with $n = 1$. Now assume it holds for $n = k$ and consider the case when $n = k + 1$.

$$A^{k+1} = A^k * A = \begin{bmatrix} 1 & 2k \\ 0 & 1 \end{bmatrix} A = \begin{bmatrix} 1 & 2 + 2k \\ 0 & 1 \end{bmatrix}.$$

So the condition holds for all $n \in \mathbb{N}$. To show it holds also for negative powers, we prove by induction that

$$A^{-n} = \begin{bmatrix} 1 & -2n \\ 0 & 1 \end{bmatrix} \text{ and } B^{-n} = \begin{bmatrix} 1 & 0 \\ -2n & 1 \end{bmatrix}.$$

Notice that

$$\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

So $A^{-1} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ and the condition holds when $n = 1$. Assume it holds for $n = k$ and then consider the $n = k + 1$ case.

$$A^{-(k+1)} = A^{-k}A^{-1} = \begin{bmatrix} 1 & -2k \\ 0 & 1 \end{bmatrix} A^{-1} = \begin{bmatrix} 1 & -2 - 2k \\ 0 & 1 \end{bmatrix}$$

So this condition holds for all $n \in \mathbb{N}$. For both induction arguments, an identical process suffices to prove the symmetric result for B.

Since every element of $\langle A \rangle$ is a power of A and every element of $\langle B \rangle$ is a power of B, it follows that A^n and B^n represent arbitrary elements of $\langle A \rangle$ and $\langle B \rangle$ respectively, for some $n \in \mathbb{Z}$.

This information can be used to prove that elements of $\langle A \rangle$ and $\langle B \rangle$ act on elements from one subset of $\mathbb{Z}^2$ by mapping them to a different, disjoint subset of $\mathbb{Z}^2$.

Let X_1 and X_2 be subsets of $\mathbb{Z}^2$ such that

$$X_1 = \left\{ \begin{bmatrix} x & y \end{bmatrix} \in \mathbb{Z}^2 \colon |x| > |y| \right\} \text{ and } X_2 = \left\{ \begin{bmatrix} x & y \end{bmatrix} \in \mathbb{Z}^2 \colon |x| < |y| \right\}.$$

So X_1 and X_2 are disjoint subsets of $\mathbb{Z}^2$. For an arbitrary nontrivial element $A^n \in \langle A \rangle$ and an arbitrary element $\begin{bmatrix} x & y \end{bmatrix} \in X_1$,

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 2n \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} x & 2xn + y \end{bmatrix} \in X_2$$

since it follows from $|y| < |x|$ that

$$\begin{aligned} |2xn + y| &\geq |2n||x| - |y| && \text{(Triangle Inequality)} \\ &\geq 2|x| - |y| \\ &= |x| + (|x| - |y|) \\ &> |x|. && \text{(since } |y| < |x|) \end{aligned}$$

A symmetric argument can be used to show that for an arbitrary nontrivial element $B^n \in \langle B \rangle$ and an arbitrary element $\begin{bmatrix} x & y \end{bmatrix} \in X_2$,

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 2n & 1 \end{bmatrix} = \begin{bmatrix} x + 2yn & y \end{bmatrix} \in X_1$$

So nontrivial elements of $\langle A \rangle$ send elements of X_1 to X_2 and nontrivial elements of $\langle B \rangle$ send elements of X_2 to X_1. Now if any nontrivial $K \in \langle A, B \rangle$ maps elements from X_2 to X_1 and elements of X_1 to X_2, then it could not possibly be the identity matrix. This would imply that the α in $F(X)$ that maps to K under ϕ is not in $\ker \phi$. Since K and hence α are arbitrary, we would have that $\ker \phi = \{\epsilon\}$ and therefore ϕ is injective (and hence an isomorphism).

So we just need to show that any product of arbitrary elements in $\langle A \rangle$ or $\langle B \rangle$ maps an element of either X_1 or X_2 into the other. To do this we must first consider the different types of arbitrary nontrivial elements in $F(X)$.

Observe that any nontrivial word α in $F(X)$ can be written as an alternating sequence of powers of as and bs. For instance, the word $aaabbb$ can be written a^3b^3. We can then partition the words of $F(X)$ into four types: words that begin with an a term and end with an a term, words that begin with an a term and end with a b term, words that begin with a b term and end with a b term, and words that begin with a b term and end with an a term:

$$\begin{aligned} &a^{k_1}b^{k_2}a^{k_3}\dots b^{k_{n-1}}a^{k_n} \\ &a^{k_1}b^{k_2}a^{k_3}\dots a^{k_{n-1}}b^{k_n} \\ &b^{k_1}b^{k_2}a^{k_3}\dots a^{k_{n-1}}b^{k_n} \\ &b^{k_1}b^{k_2}a^{k_3}\dots b^{k_{n-1}}a^{k_n} \end{aligned}$$

Since ϕ is surjective, applying ϕ to each of these four forms gives the four possible cases for the form of $(\alpha)\phi$:

$$\begin{aligned} &A^{k_1}B^{k_2}A^{k_3}\dots B^{k_{n-1}}A^{k_n} \\ &A^{k_1}B^{k_2}A^{k_3}\dots A^{k_{n-1}}B^{k_n} \\ &B^{k_1}A^{k_2}B^{k_3}\dots A^{k_{n-1}}B^{k_n} \\ &B^{k_1}A^{k_2}B^{k_3}\dots B^{k_{n-1}}A^{k_n} \end{aligned}$$

We will now show that an arbitrary element from each of these four cases sends either an element of X_1 into X_2 or an element of X_2 into X_1, implying in each case that the product $(\alpha)\phi$ cannot be the identity I. Let $k_1, \ldots, k_n \in \mathbb{Z} \setminus \{0\}$.

Case 1: $(\alpha)\phi = A^{k_1}B^{k_2}A^{k_3}\ldots B^{k_{n-1}}A^{k_n}$. Then, for some elements $x_1, x_3, x_5, \ldots, x_n \in X_1$ and elements $y_2, y_4, y_6, \ldots, y_{n-1} \in X_2$,

$$\begin{aligned} x_1((\alpha)\phi) &= (x_1)(A^{k_1}B^{k_2}\ldots A^{k_n}) \\ &= (y_2)(B^{k_2}\ldots A^{k_n}) \\ &\vdots \\ &= (x_n)(A^{k_n}) \in X_2 \end{aligned}$$

So $(\alpha)\phi$ sends elements from X_1 to X_2. Since X_1 and X_2 are disjoint, $(\alpha)\phi$ cannot be the identity.

Case 2: $(\alpha)\phi = B^{k_1}A^{k_2}B^{k_3}\ldots A^{k_{n-1}}B^{k_n}$. By a symmetric argument to that of case 1, for an arbitrary element $y \in X_2$, $(y)(B^{k_1}A^{k_1}\ldots B^{k_n}) \in X_1$.

Case 3: $(\alpha)\phi = A^{k_1}B^{k_2}A^{k_3}\ldots A^{k_{n-1}}B^{k_n}$. Choose an element $A' \in \langle A\rangle$ such that $A' \neq I$ and $A' \neq A^{-k_1}$. Then

$$A'(\alpha)\phi A'^{-1} = A'A^{k_1}B^{k_2}\ldots B^{k_n}A'^{-1} \neq I$$

since it is of the form described in case 1. Therefore, $(\alpha)\phi$ is not the identity element.

Case 4: $(\alpha)\phi = B^{k_1}A^{k_2}B^{k_3}\ldots B^{k_{n-1}}A^{k_n}$. A symmetric argument to that of case 3 gives the result that $(\alpha)\phi \neq I$.

Therefore, there is no product of elements from $\langle A\rangle$ and $\langle B\rangle$ that is equal to the identity, and so $\ker\phi = \{\epsilon\}$ and ϕ is an isomorphism.

The group generated by A and B in Example 4.3 is called the Sanov Subgroup. The proof of its freeness was derived from the proof of a more general result, called the Ping Pong Lemma. [3]

Free groups are important since, as a consequence of the following result, it is possible to express any group in terms of a free group on some basis.

Theorem 4.3. *Every group is a homomorphic image of a free group.*

Proof. Let G be a group generated by the set X, and let $f : X \to G$ be the function defined by $(x)f = x$ for all $x \in X$. Similarly, let $\iota : X \to F(X)$ be an inclusion mapping. Then $\langle \operatorname{im}\iota\rangle = F(X)$ and $\langle \operatorname{im} f\rangle = G$. Since both f and and ι are inclusion mappings from X, f can be extended to a homomorphism $\phi : F(X) \to G$ defined by $(x)\phi = x$ for all $x \in \operatorname{im}\iota$. The generators $x \in X$ are in the image of ϕ, so it is surjective, and therefore $\operatorname{im}\phi = G$. [10] □

To find a free group given a group G, take a generating set of G and take an alphabet X with the same size as the generating set. Then the group of reduced words over this alphabet $F(X)$ is free and G can be expressed as the homomorphic image of a mapping from $F(X)$. When the mapping ϕ from a free group $F(X)$ to a group G is not just surjective, but also injective, then every element from $F(X)$ maps to exactly one element of G, which implies there does not exist a product of non-trivial elements of G that cancels to the identity. This means that F and G are isomorphic, so G is also free on the same basis as F.

The remainder of this section will consist of other useful results about free groups.

Lemma 4.4. *For $w \in F(X)$, where $F(X)$ is free, if $w^n = \epsilon$ then $w = \epsilon$, for any $n \neq 0$.*

Proof. By induction on the length of a nontrivial word $\alpha \in F(X)$. This proof requires two base cases, since the inductive step increases the length of the word by two. If $|\alpha| = 1$ then α^n must be reduced, and so $\alpha^n \neq \epsilon$. If $|\alpha| = 2$ then, writing α as ab, a and b must not be inverses, so $\alpha^n \neq \epsilon$.

Assume the condition holds for words of length k. Let α_{k+2} be an element of $F(X)$ such that $|\alpha_{k+2}| = k + 2$. Then we can write $\alpha_{k+2} = b_1\alpha_k b_2$ for some nontrivial $b_1, b_2 \in X$ and a word α_k of length k. Write α_k as $a_1 a_2 \dots a_k$. Since $|\alpha_{k+2}| = |\alpha_k| + 2$, b_1 and a_1 cannot be inverses and b_2 and a_k cannot be inverses. So we have two cases in the reduction of α_{k+2}^n:

Case 1: b_1 and b_2 are not mutual inverses. Then $\alpha_{k+2}^n \neq \epsilon$ and the condition holds.

Case 2: b_1 and b_2 are inverses of each other. Then

$$\begin{aligned}\alpha_{k+2}^n &= (b_1 a_1 \dots a_k b_2)(b_1 a_1 \dots a_k b_2) \dots (b_1 a_1 \dots a_k b_2) \\ &= (b_1 a_1 \dots a_k)(a_1 \dots a_k) \dots (a_1 \dots a_k b_2) \\ &\neq \epsilon\end{aligned}$$

since $(a_1 \dots a_k)(a_1 \dots a_k) \dots (a_1 \dots a_k) = \alpha_k^n \neq \epsilon$. □

Lemma 4.5. *Let F be a free group with basis X where $|X| > 1$. Then the set of elements in F that commute with all elements of F is precisely $\{\epsilon\}$.*

Proof. Let $x_1, \dots, x_n$, with $n \geq 2$ (since $|X| > 1$) be the generators of F. Let α be a nontrivial element of F such that for all elements $\beta \in F$, $\alpha\beta = \beta\alpha$. Without loss of generality, let $\alpha = a_1^{\gamma_1} \dots a_n^{\gamma_n}$ where the a_i are letters from X and $\gamma_1, \dots, \gamma_n \in \{-1, 1\}$. Let a_k be an element of X that is not equal to $a_1^{\gamma_1}$ or $a_n^{-\gamma_2}$ and let $\omega = a_k a_1^{-\gamma_1}$. Since α commutes with all elements of F,

$$\begin{aligned} & \omega\alpha = \alpha\omega \\ \implies \; & a_k a_1^{-\gamma_1} a_1^{\gamma_1} \dots a_n^{\gamma_n} = a_1^{\gamma_1} \dots a_n^{\gamma_n} a_k a_1^{-\gamma_1} \\ \implies \; & a_k a_2^{\gamma_2} \dots a_n^{\gamma_n} = a_1 \dots a_n^{\gamma_n} a_k a_1^{-\gamma_1}\end{aligned}$$

On the left hand side of this equation, there was at least one reduction, but the word on the right hand side is irreducible. So this is a contradiction, and α does not commute with every element of F. □

4.2 Presentations

In section 4.1 we established that every group is the homomorphic image of a free group. In this section, we will demonstrate that it is possible to fully characterize any given group as the homomorphic image of a free group using *presentations*.

Lemma 4.6. *Let G be a group and let X be a generating set of G. Then G is isomorphic to a quotient of the free group $F(X)$.*

Proof. From Theorem 4.3, we know that $\operatorname{im}\phi = G$ for some $\phi : F(X) \to G$. So by the first isomorphism theorem,

$$G = \operatorname{im}\phi \cong F(X)/\ker\phi.$$

[12] □

Presentations are a way of representing the group G as a quotient of $F(X)$.

Definition 4.5. Let $F(X)$ be a free group with basis X and let G be a group. A presentation of G is a surjective homomorphism π from F to G. The elements of $R = \ker\pi$ are called *relators*. [10]

So by the first isomorphism theorem, $F/R \cong \operatorname{im}\pi = G$.

This definition is not very human-readable. Given a surjective homomorphism and the free group, it is not easy to visualize what the elements of G are. From this definition, however, we can create much more useful notation. But first, it is necessary to define the *normal closure* of a set A in a group G.

Proposition 4.1. *Let G be a group and let $X \subseteq G$. The* normal closure *of X in G is the intersection all normal subgroups of G containing X and is itself a normal subgroup of G.*

Proof. Let N_i, for $i \in I$, where I is an index set, be a collection of normal subgroups of a group G. By proving that $\bigcap_{i\in I} N_i$ is a normal subgroup, we prove that the normal closure of X must also be a normal subgroup by definition.

First, we prove that it is a subgroup. Note that $\bigcap_{i\in I} N_i$ is non-empty since each N_i must contain the identity element in order to be a group. Now consider elements $g, h \in \bigcap_{i\in I} N_i$. Both elements are in every subgroup N_i which implies that also $h^{-1} \in \bigcap_{i\in I} N_i$, since groups are closed under inverses, and also that $gh^{-1} \in \bigcap_{i\in I} N_i$ since groups are closed under multiplication. So $\bigcap_{i\in I} N_i$ is a subgroup of G.

Second we show that it is normal. Let $x \in \bigcap_{i\in I} N_i$ be an arbitrary element and let $y \in G$. Since each N_i is normal, and $x \in N_i$ for all i, it follows that $yxy^{-1} \in \bigcap_{i\in I} N_i$. So the intersection of all N_i is also normal. [10] □

Since the closure of X is the intersection of all normal subgroups of G that contain X, it must be the smallest normal subgroup which contains X in G. The normal closure of X in G will be denoted $\langle\langle X\rangle\rangle$.

Let $F(X)$ be a free group with basis X, let $\pi : F(X) \to G$ be a presentation of a group G. Then $(X)\pi$ is the set of generators for G. Also let S be any subset of $F(X)$ such that $\ker\pi = \langle\langle S\rangle\rangle$. Then S is a set of words in $F(X)$ which map to the identity under π, and the remainder of $\ker\pi$ can be derived by taking the normal closure of S in $F(X)$. Then π together with X and S define a set of generators and a set of relators. [10] This completely characterizes (a group isomorphic to) G and is denoted

$$G = \langle X \mid S\rangle.$$

This notation will be used to denote a presentation, and is in fact not only an equivalent definition, but the more standard one.

In $\langle X \mid S\rangle$, the elements in S are certain words that, when written in terms of the generators of G, equate to the identity. These are called the defining relators of G. Any other relators (i.e. words that equate to the identity when written out of generators of G) of the presentation are elements of the normal closure of S in $F(X)$. So they are any product of conjugated elements of S in $F(X)$ with elements of $F(X)$.

Note that the set of defining relators is not unique for a given presentation π. Two different subsets S_1 and S_2 may both generate the same normal closure. In other words, it is possible, for some X, that the group defined by $\langle X \mid S_1\rangle$ is the same as the group defined by $\langle X \mid S_2\rangle$.

Definition 4.6. Let G be a group with generating set X_1 and let $\langle X_2 \mid S\rangle$ be a presentation such that $|X_1| = |X_2|$. Then G *satisfies the relators of* S if there exists a surjective homomorphism $\phi : \langle X_2 \mid S\rangle \to G$ such that $(s)\phi \in \ker\phi$ for all $s \in S$.

Theorem 4.4. *Let $\langle X \mid \omega_1, \ldots, \omega_n\rangle$ be a presentation and let N be the normal closure of $\{\omega_1, \ldots, \omega_n\}$ in $F(X)$. Then the group $G = F(X)/N$*

- *is defined by $\langle X \mid \omega_1, \ldots, \omega_n\rangle$,*
- *is generated by the cosets xN where $x \in X$,*
- *and these generators satisfy the defining relators of the presentation.*

Proof. Let $\pi : F \to G$ be defined by $(f)\pi = fN$ for all $f \in F(X)$. This can be shown to be both surjective and a homomorphism, so π is a presentation. Since N is the normal closure of $\{\omega_1, \ldots, \omega_n\}$ in $F(X)$, the group G is represented by the presentation $\langle X \mid \{\omega_1, \ldots, \omega_n\}\rangle$.

An arbitrary element of G then has the form ωN where $\omega = x_1^{\gamma_1}x_2^{\gamma_2}\ldots x_m^{\gamma_m}$ where $\gamma_1, \ldots, \gamma_m \in \{-1, 1\}$ and $x_1, \ldots, x_m \in X$. So

$$\begin{aligned}\omega N &= (x_1^{\gamma_1}\ldots x_m^{\gamma_m})N \\ &= (x_1N)^{\gamma_1}(x_2N)^{\gamma_2}\ldots(x_mN)^{\gamma_m}\end{aligned}$$

So G is generated by the set $\{xN : x \in X\}$.

Consider the arbitrary defining relator ω_i. Since N is the normal closure of $\{\omega_1, \ldots, \omega_n\}$, the element ω_i must be contained in N. Therefore $\omega_i N = \epsilon N = 1_G$, so G satisfies the defining relations. [12] □

When given a presentation $\langle X \mid S\rangle$, the group G it represents is the "largest" possible group generated by X in which the relations of S are satisfied.

If a group has generators that satisfy the defining relators given in a certain presentation, then it cannot be any bigger than the group defined by the presentation. It is either the same group, and therefore the same size, or, if it is finite, then it is smaller than the one defined by the presentation (as additional relations cannot possibly create more elements). If the group defined by the presentation is infinite, then adding more relators to the presentation does not mean the resulting group is actually smaller: it could still be infinite, but the group with more relators would be a quotient of the original group.

This concept is formalized in von Dyck's theorem.

Theorem 4.5 (von Dyck)**.** *Let $F(X)$ be a free group with basis X. Let G and H be groups with presentations $\gamma : F(X) \to G$ and $\delta : F(X) \to H$ such that each relator of γ is also a relator of δ. Then the function ϕ defined by $(\alpha)\gamma \mapsto (\alpha)\delta$ for every $\alpha \in F(X)$ is a well-defined surjective homomorphism from G to H.*

Proof. Since δ and γ are presentations, they are homomorphisms. So for an arbitrary $\alpha, \beta \in F(X)$ under γ,

$$((\alpha)\gamma(\beta)\gamma)\phi = ((\alpha\beta)\gamma)\phi = (\alpha\beta)\delta = (\alpha)\delta(\beta)\delta$$

So ϕ is a homomorphism.

Since every relator of γ is a relator of δ, $\ker\gamma \subseteq \ker\delta$. Since γ is a presentation, it is surjective. So if $g \in G$ then $(\alpha_1)\gamma = g$ for some $\alpha_1 \in F(X)$. Now if $g = (\alpha_2)\delta$, then $\alpha_2 = \alpha_1\alpha_3$ or $\alpha_2 = \alpha_3\alpha_1$ for some $\alpha_3 \in \ker\gamma$ (without loss of generality, we assume the former). But $\ker\gamma \subseteq \ker\delta$ so $(\alpha_1)\delta = (\alpha_1\alpha_3)\delta = g$ so ϕ is well defined.

Also, since both γ and δ are presentations of G, $\operatorname{im}\phi = \operatorname{im}\delta = G$. So ϕ is also surjective. [10] □

The consequence of this theorem, in terms of presentations given as $\langle X \mid R\rangle$, is that if there exists a group G generated by (a copy of) X that satisfies the defining relators of R, then G is isomorphic to a quotient of the group defined by the presentation $\langle X \mid R\rangle$. In other words, any group with a generating set X that satisfies all of the defining relators in R, is at most as big (which could be infinite) as the group defined by the presentation (which is the largest possible group generated by the set X that satisfies the defining relators in R).

The following are several examples of groups defined by certain presentations.

Example 4.4. The group $\langle X \mid \rangle$ is the group with no defining relators. In other words, $\ker\pi = \{\epsilon\}$ (where $\pi : F(X) \to \langle X \mid \rangle$ is the surjective homomorphism defining this presentation). So π is injective as well as surjective and therefore an isomorphism. So $\langle X \mid \rangle \cong F(X)$. [6]

The group $\langle X \mid \rangle$ is called *freely presented* since it contains no relations. A group being freely presented implies that the group is also free (with a basis of its generating set). A group being free on a basis X implies that it can be written as the presentation $\langle X \mid \rangle$.

Example 4.5. The group G defined by the presentation $\langle x \mid x^n\rangle$ is isomorphic to $(\mathbb{Z}_n, +)$: it is generated by a single element, 1, and $1 + 1 + \cdots + 1$ (n times) is 0, the identity element of $\mathbb{Z}_n$. So $\mathbb{Z}_n$ is generated by a copy of $\{x\}$ and its generator satisfies the relation $x^n = \epsilon$. So $\mathbb{Z}_n$ must be isomorphic to a quotient of G. Also, $|\mathbb{Z}_n| = n$ and $|G| = n$, so they must be isomorphic.

Example 4.6. Consider the group G defined by the presentation $\langle a, b \mid a^3b^{-2}, aba^{-1}b^{-1}\rangle$. The second relator of G indicates that G is commutative since $aba^{-1}b^{-1} = 1_G$ implies that $ab = ba$. So any arbitrary word in G can be written as a^ib^j where $i, j \in \mathbb{Z}$.

Notice that $(\mathbb{Z}, +)$ is generated by the set $\{2, 3\}$ and that $2 + 2 + 2 + (-3) + (-3) = 0$. Define a function $\phi : G \to \mathbb{Z}$ by mapping $a \mapsto 2$ and $b \mapsto 3$. Note

$$(a^3b^{-2})\phi = (a)\phi^3(b)\phi^{-2} = 2 * 3 + 3 * -2 = 6 + (-6) = 0 \text{ and}$$
$$(aba^{-1}b^{-1})\phi = (a)\phi(b)\phi(a^{-1})\phi(b^{-1})\phi = 2 + 3 - 2 - 3 = 0$$

so ϕ respects the relations of G, is well defined, and is a homomorphism. If we prove that ϕ is an isomorphism, i.e. is injective and surjective, then we have proved that $\mathbb{Z}$ is defined by the presentation.

Any $n \in \mathbb{Z}$ can be expressed as $n = 2(-n) + 3(n)$, i.e. $(a^{-n}b^n)\phi = n$. So ϕ is surjective. Now ϕ is injective if and only if $\ker\phi = \{1_G\}$. Let $a^ib^j \in G$ such that $(a^ib^j)\phi = 0$. Then $(a^ib^j)\phi = 2i + 3j = 0$. So either $i = j = 0$, or neither i nor j are zero.

If $i = j = 0$ then $a^ib^j = 1_G$. Otherwise,

$$2i + 3j = 0 = k(2 * 3 + 3 * -2) = 2 * 3k + 3 * -2k$$

for some k. So $i = 3k$ and $j = -2k$. Then $a^ib^j = a^{3k}b^{-2k} = (a^3b^{-2})^k = 1_G^k = 1_G$. So $\ker\phi = 1_G$. and ϕ is an isomorphism.

Notice that in section 4.1 we proved that $\mathbb{Z}$ is free. Yet in the above example, we proved that it is also given by a presentation that is *not* free. These examples demonstrate the importance of the chosen generating set: $\mathbb{Z}$ is free on a set of a single element set, but requires relations on a two element set.

Definition 4.7. A group is *finitely presented* if it has a presentation $\langle X \mid S\rangle$ in which both X and S are finite. [12]

An infinite group can have a finite presentation: $\langle a, b \mid ab\rangle$ has an infinite number of elements ($a, a^2, \ldots, a^n$ and so on) but has a finite number of generators and relations. Since it is difficult to represent infinite sets on a computer, this project will be dealing purely with finite presentations. Specifically, finite, free presentations.

4.3 Equations

Now we have defined free groups and freely presented groups, but what is an *equation* over a freely presented group? Like any ordinary equation, an equation over a freely presented group consists of a set of variables, a set of constants, and an equality relation between two combinations of the constants and variables via different mathematical operations. However, for an equation over a freely presented group F, the constants are all from F and the only operation is the operation defined on the group F (e.g. concatenation followed by repeated reduction).

Definition 4.8. Let A and Y be disjoint alphabets. Then an equation over the free group $F(A)$ is given by a pair of words $\omega_1, \omega_2 \in F(A \cup Y)$ representing the equality

$$\omega_1 = \omega_2 \tag{1}$$

Elements of A and their inverses are called *constants* and elements of Y and their inverses are called *variables*. [2]

In other words, an equation is a relation between two reduced words over an alphabet and a set of variables, and their inverses. Throughout, upper case letters from the Latin alphabet will be used to denote variables, and lower case letters from the Latin alphabet will be used to denote constants. Words over $F(A)$ or words over $F(A \cup Y)$ will be denoted by Greek letters.

It is possible to further simplify this definition.

Theorem 4.6. *Any equation $\omega_1 = \omega_2$ over a group $F(A)$ with a set of variables Y can be expressed as $\omega_3 = \epsilon$ for some $\omega_3 \in F(A \cup Y)$.*

Proof.

$$\omega_1 = \omega_2 \implies \omega_1 \omega_2^{-1} = \epsilon.$$

□

So any equation can be expressed as $\omega = \epsilon$ where $\omega \in F(A \cup Y)$. In a generic equation, it is possible for variables and their inverses to occur multiple times, as it is possible for any sequence of constants to be repeated. However when each variable occurs exactly once, the equation is called *linear*.

Definition 4.9. An equation $\omega = \epsilon$ is called *linear* if every one of the generators from Y occurs in ω exactly once, each time with exponent -1 or $+1$.

A linear equation is an equation where each variable occurs exactly once. Similarly, a quadratic equation is an equation in which each variable occurs exactly twice.

Definition 4.10. An equation $\omega = \epsilon$ is called *quadratic* if every one of the generators from Y occurs exactly twice in ω, each time with exponent -1 or $+1$. [2]

Now that equations have been defined, it is natural to define the solution to an equation.

Definition 4.11. Let $\omega = \epsilon$ be an equation where $\omega \in F(A \cup Y)$. A *solution* to $\omega = \epsilon$ is a homomorphism $S : F(A \cup Y) \to F(A)$ such that

- $(a)S = a$ for all $a \in A$, and
- $(\omega)S = \epsilon$.

For brevity, in a solution, only the mappings of variables will be defined explicitly as the mapping of constants is fixed.

Example 4.7. The equation $X_2 X_1 X_3^{-1} = \epsilon$ has the solution

$$S = \begin{cases} X_1 \mapsto \alpha_1 \\ X_2 \mapsto \alpha_2 \\ X_3 \mapsto \alpha_2 \alpha_1. \end{cases}$$

Note this equation also has many other possible solutions.

Definition 4.12. An endomorphism $F(A \cup Y) \to F(A \cup Y)$ is an A-map if it fixes the elements of A. We can define an equivalence relation between words from $F(A \cup Y)$ by $\omega_1 \equiv \omega_2$ if $\omega_1 = (\omega_2)\phi$ where ϕ is an A-map. [2]

Therefore, an equation is solvable if it is equivalent to the empty word. Additionally, if an equation $\omega = \epsilon$ is solvable, then every other word it is equivalent to is also equivalent to the empty word. So when searching for a solution, we want to express ω in the simplest equivalent form possible.

With the motivation of finding the particular A-maps that will simplify a word ω, we first define certain patterns of subwords that occur in words over $F(A \cup Y)$. We will then show that these subwords can be simplified using certain A-maps.

Definition 4.13. A word $\omega \in F(A \cup Y)$ is *Y-redundant* if, for $X_1, X_2 \in Y$ where $X_1 \neq X_2^{-1}$, it contains one or more subwords of the form $(X_1X_2)^{\pm 1}$, and the variables $X_1^{\pm 1}$ and $X_2^{\pm 1}$ occur in ω only in such subwords. The subword $(X_1X_2)^{\pm 1}$ is called a *Y-redundancy* of ω. [2]

For example, the word $\alpha X_1X_2\beta(X_1X_2)^{-1}\alpha$ is Y-redundant, but the word $\alpha X_1\beta X_2\alpha X_1X_2$ is not.

Definition 4.14. A word $\omega \in F(A \cup Y)$ is *A-redundant* if, for $X \in Y$ and $a \in A$, it contains one or more subwords of the form $(Xa)^{\pm 1}$ and the variable $X^{\pm 1}$ occurs in ω only in such subwords. The word $(Xa)^{\pm 1}$ is called a *A-redundancy* of ω. [2]

The words $(Xa)^3$ and $X_1\alpha X_2\beta X_1\alpha$ are A-redundant. The word $(Xa)^3X$ is not, as the variable X occurs once in a subword that is *not* Xa.

Definition 4.15. A word that is reduced and is neither A-redundant nor Y-redundant is called irredundant. [2]

Lemma 4.7. *Every word ω in $F(A \cup Y)$ is equivalent to an irredundant word.*

Proof. By induction on the number of distinct variables that occur in ω: if ω contains no variables, then by definition ω is irredundant. Now assume that any word ω_{n-1} which contains $n-1$ distinct variables is equivalent to an irredundant word. Suppose ω_n has n variables and is Y-redundant with Y-redundancy X_1X_2. Define an A-map ϕ which maps X_2 to ϵ. Then $(\omega_n)\phi$ has $n-1$ variables and is by assumption equivalent to an irredundant word. So $\omega_n \equiv (\omega)\phi$ is also equivalent to an irredundant word.

By induction on the number of A-redundancies that occur in ω: if ω does not contain any A-redundancies and is *not* irredundant, then it must have a Y-redundancy. By the above inductive argument, it is equivalent to an irredundant word. Now assume that any word ω_{n-1} with $n-1$ A-redundancies is equivalent to an irredundant word. Let ω_n be a word with n A-redundancies, one of which is Xa. Define an A-map ϕ which maps X to Xa^{-1}. Then $(\omega_n)\phi$ has $n-1$ A-redundancies, so by assumption is equivalent to an irredundant word. So $\omega_n \equiv (\omega)\phi$ is also equivalent to an irredundant word. [2] □

We can assume from here on that all equations are irredundant. With the terminology for equations over free groups established, we can turn to the question of solvability of linear and quadratic equations.

5 Linear Equations

This section explores the process of solving linear equations over freely presented groups. It contains a set of useful terms and definitions which are then used to prove that every linear equation has at least one solution, an original algorithm for finding every solution to a given equation, an evaluation of the algorithm's performance, and a proof that the algorithm is correct.

5.1 Mathematical Background

Here, we will establish the mathematical backbone of the algorithm for finding solutions to linear equations over freely presented groups by defining relevant terms and proving results.

Up to ordering of variables, and allowing any α_i to be the empty word,

$$\alpha_1 X_1^{\gamma_1} \alpha_2 X_2^{\gamma_2} \alpha_3 \dots \alpha_{n-1} X_{n-1}^{\gamma_{n-1}} \alpha_n = \epsilon,$$

where $\gamma_1, \dots, \gamma_{n-1} \in \{-1, 1\}$, is a completely generic description of a linear equation since by Theorem 4.6 the right side of the equation can be made to be the empty word. Furthermore, by applying an A-map that maps any variable X_i which occurs in the equation with a negative exponent to its inverse X_i^{-1}, we can further simplify the equation to

$$\alpha_1 X_1 \alpha_2 X_2 \alpha_3 \dots \alpha_{n-1} X_{n-1} \alpha_n = \epsilon. \tag{2}$$

The number of variables is called the *degree* of the equation. In this section, we are concerned with whether solutions to such an equation exist, how they can be characterized if they do, and whether it is possible to design an algorithm for finding all such solutions.

A solvable linear equation can have potentially infinite solutions, formed by taking a solution and padding each variable assignment with letters that cancel with inverses inserted into other variable assignments. Having infinite solutions is problematic to an algorithm that finds one solution at a time, since the algorithm would have to run infinitely to find all of them. Additionally, these "padded" solutions are not interesting as the extra added letters, that differentiate them from the shorter solutions from which they were derived, have nothing to do with the original equation. Instead, we are looking for the "shortest" possible variable substitutions that will result in the equation being solved. This concept is formalized in the following definition.

Definition 5.1. A solution S to a linear equation $\alpha_1 X_1 \alpha_2 X_2 \alpha_3 \dots \alpha_{n-1} X_{n-1} \alpha_n = \epsilon$ is *short* if in no step of the repeated reduction of the word $(\alpha_1 X_1 \alpha_2 X_2 \alpha_3 \dots \alpha_{n-1} X_{n-1} \alpha_n)S$ do letters of $(X_i)S$ reduce with letters of $(X_j)S$, for any $i, j \in \{1, \dots, n\}$.

Example 5.1. Let a, b, c, and d be letters from an alphabet A. The equation

$$(abc)X_1(a^3)X_2 = \epsilon$$

has a short solution

$$S = \begin{cases} X_1 \mapsto c^{-1}b^{-1}a^{-1} \\ X_2 \mapsto a^{-3} \end{cases}.$$

But it also has the solution

$$S = \begin{cases} X_1 \mapsto c^{-1}b^{-1}a^{-1}d \\ X_2 \mapsto a^{-3}d^{-1} \end{cases}.$$

The latter solution, however relies on the d in X_1 cancelling with the d^{-1} of X_2. Therefore it is not short.

Notice that in the above example the length of a short solution is the same as the totalled length of all the constants, i.e.

$$\sum_{i=1}^{n}|(X_i)S| \leq \sum_{i=1}^{n+1}|\alpha_i|. \tag{3}$$

However, this condition is not sufficient for defining a short solution.

Example 5.2. Consider the equation

$$\alpha X_1 X_2 \alpha^{-1} = \epsilon.$$

Setting X_1 and X_2 to ϵ is a short solution to this equation. However, the solution

$$S = \begin{cases} X_1 \mapsto \beta \\ X_2 \mapsto \beta^{-1} \end{cases}.$$

is not short, since the word $\alpha\beta\beta^{-1}\alpha_1^{-1}$ reduces to ϵ only by reducing $\beta = (X_1)S$ with $\beta^{-1} = (X_2)S$. However this solution still satisfies property 3.

Theorem 5.1. *Every linear equation of degree greater than 0 has at least one short solution.*

Proof. Consider the arbitrary linear equation

$$\alpha_1 X_1 \alpha_2 X_2 \alpha_3 \dots \alpha_{n-1} X_{n-1} \alpha_n = \epsilon. \tag{4}$$

Define a homomorphism S which sends all of $X_2, X_3, \dots, X_{n-1}$ to ϵ and acts as the identity on all elements of the alphabet. Then we are left with the equation

$$\alpha_1 X_1 \overline{\alpha} = \epsilon$$

where $\overline{\alpha} = \alpha_2\alpha_3 \dots \alpha_n$. Multiplying on both sides appropriately gives $X_1 = \alpha_1^{-1}\overline{\alpha}^{-1}$. Extending S with this assignment gives the solution

$$S = \begin{cases} X_1 \to \alpha_1^{-1}\overline{\alpha}^{-1} \\ X_2 \to \epsilon \\ \vdots \\ X_n \to \epsilon \end{cases} = \begin{cases} X_1 \to \alpha_1^{-1}\alpha_n^{-1}\alpha_{n-1}^{-1} \dots \alpha_2^{-1} \\ X_2 \to \epsilon \\ \vdots \\ X_n \to \epsilon \end{cases}$$

Since no two $(X_i)S$ reduce with each other in the word $(\alpha_1 X_1 \alpha_2 X_2 \dots \alpha_{n-1} X_{n-1} \alpha_n)S$, S is a short solution to equation 4. Since equation 4 is completely arbitrary, every linear equation has at least one short solution. □

Note that an equation of degree 0 does not necessarily have a solution. Such an equation does not contain any variables, and so is only solvable when the constants reduce to ϵ.

Corollary 5.1. *A linear equation of degree 1 has exactly one short solution.*

Proof. Consider the equation $\alpha_1 X \alpha_2 = \epsilon$. Multiplying on both sides gives $X = \alpha_1^{-1}\alpha_2^{-1}$. □

This result will form the base case of the recursive algorithm that finds the solutions to a linear equation.

Definition 5.2. A triple (u, v, w), where u, v, w are elements from a freely presented group, is singular if v cancels entirely when forming the product uvw. [8]

In other words, the triple (u, v, w) is singular if there exists a u', w', p, and q such that

$$\begin{aligned} u &= u'p, \\ w &= qw', \\ v &= p^{-1}q^{-1}. \end{aligned}$$

From this definition we can show that any equation $\omega \in F(A \cup Y)$ (not necessarily a linear one) must contain at least one singular triple to be solvable.

Lemma 5.1. *If $\omega_1\omega_2\omega_3...\omega_n = \epsilon$ then at least one of the triples*

$$(\epsilon, \omega_1, \omega_2), (\omega_1, \omega_2, \omega_3), \ldots, (\omega_{n-1}, \omega_{n-1}, \omega_n), (\omega_{n-1}, \omega_n, \epsilon)$$

is singular.

Proof. Suppose that $\omega_1, \ldots, \omega_n$ are words such that none of the above triples are singular. Then we may write each ω_i as follows:

$$\begin{aligned} \omega_1 &= \alpha_1\beta_1 \\ \omega_i &= \beta_{i-1}^{-1}\alpha_i\beta_i \quad \text{for } 1 < i < n \\ \omega_n &= \beta_{n-1}^{-1}\alpha_n \end{aligned}$$

where every $\alpha_i\alpha_{i+1}$ is irreducible. Then the word $\omega_1 \ldots \omega_n$ reduces to $\alpha_1\alpha_2 \ldots \alpha_n \neq \epsilon$ and we have a contradiction. [8] □

An equation ω from $F(A \cup Y)$, the free group over constants from a set A and variables from a set Y, is solvable if and only if it is equivalent to ϵ under some A-map ϕ. Then the above lemma tells us that the unreduced word $(\omega)\phi$ must contain at least one singular triple.

However, we also know that for any non-trivial equation $\omega = \epsilon$, we can assume without loss of generality that ω is irredundant. So ω is reduced already, and therefore does not contain any singular triples. Then the triples that arise in $(\omega)\phi$ must result from the substitution of variables from Y with words over the alphabet A. Additionally, in a linear equation we can assume that the A-map ϕ is a short solution and that the equation can be represented in the form given in equation 4. Then a singular triple in a linear equation must have the form (u, v, w), where u and w are constants from the initial equation ω, and v is the image of a variable X_i under ϕ.

From this reasoning, we establish the following theorem:

Theorem 5.2. *Let S be a solution to the equation $\alpha_1X_1\alpha_2X_2\alpha_3 \ldots \alpha_{n-1}X_{n-1}\alpha_n = \epsilon$. Then the unreduced word $\alpha_1((X_1)S)\alpha_2((X_2)S)\alpha_3 \ldots \alpha_{n-1}((X_{n-1})S)\alpha_n$ contains at least one singular triple of the form $(\alpha_i, (X_i)S, \alpha_{i+1})$ where $1 \leq i < n$.*

This tells us that for any solution to a linear equation, at least one of the variables (under the solution mapping) must reduce with at least one of its neighbouring constants. This fact is integral to the algorithm for finding all short solutions to a given linear equation.

5.2 Algorithm

By theorem 5.1, all linear equations of degree greater than 0 have at least one solution. However, an equation could have many possible solutions, not necessarily just one, each of which is characterized by the inverses of the constants. Also, by Corollary 5.1, a linear equation of only one variable has exactly one short solution.

Finding the solutions to a linear equation with two variables is more complicated. In the equation $\alpha_1 X_1 \alpha_2 X_2 \alpha_3 = \epsilon$, the variable X_1 does not have to cancel with the entirety of α_1 or α_2. It does not have to cancel with anything at all: it could be ϵ and X_2 could cancel with the three constants. But once X_1 has been assigned a value that creates a singular triple, the equation reduces to that of a single variable which then only has one possible short solution. So the number of solutions to this equation is given by the number of valid assignments of values to X_1 plus the number of valid assignments of values to X_2.

The number of valid assignments of variable X_i is the number of assignments such that (α, X_i, β) is a singular triple, where α and β are the neighbouring constants of the variable X_i. For each variable chosen to form the singular triple (which by Theorem 5.2 must exist), there are numerous possible assignments, each one of which must be explored. For example, if we have chosen to reduce the triple $\alpha_1 X_1 \alpha_2$, where $\alpha_1 = a_1 \dots a_n$ and $\alpha_2 = b_1 \dots b_m$, then X_1 can be mapped to any element of the sets

$$\begin{aligned} A &= \{\epsilon, a_n^{-1}, a_n^{-1}a_{n-1}^{-1}, a_n^{-1}a_{n-1}^{-1}a_{n-2}^{-1}, \dots, a_n^{-1} \dots a_1^{-1}\}, \\ B &= \{\epsilon, b_1^{-1}, b_2^{-1}b_1^{-1}, \dots, b_m^{-1} \dots b_1^{-1}\}, \text{ or} \\ AB &= \{ab : a \in A, b \in B\}. \end{aligned}$$

With each assignment of X_1, there is only one possible valid assignment for X_2. Then the rest of the solutions are found using the same process where X_2 is assigned first.

This method can be generalized to the n variable case, since no matter how large the equation, at least one of the variables must immediately cancel with its neighbouring constants. So as described above, the algorithm begins by finding the set of possible assignments to one variable, and for each of these assignments, reduces the equation accordingly to have one less variable. The base case is hit when only one variable is left. The algorithm, implemented in GAP [4], is outlined below in algorithm 1. It makes use of the functions outlined in algorithms 2, 3, and 4. The input is a list of constants and a list of variables, where the two are interleaved to form the equation. For instance, the input $[\alpha_1, \alpha_2]$ and $[X_1]$ represents the equation $\alpha_1 X_1 \alpha_2 = \epsilon$. The empty word ϵ counts as a constant, occupies an index in the list of constants, and has a length of 0. A variable must always have neighbouring constants on both sides, even if they are the identity element. If the number of constants and the number of variables don't match, then an error is thrown. In the zero variable/one constant case, the for loop on line 9 will not be entered and the list of solutions will remain empty.

Algorithm 2 creates a list of lengths along each of the neighbouring constants of the given variable. This list determines how much of each constant the ith variable will cancel with.

Algorithm 3 calculates the constant value that is assigned to the ith variable. This value becomes part of the final solution. Algorithm 4 deep copies the list of constants, removing the ith and $i+1$th constants and replacing them with a given constant. This is how the $n-1$ variable equation is formed.

Algorithm 1 Algorithm for finding set of all solutions to a linear equation

```
1: procedure SOLVE(constants, variables)
2:     if Len(variables) ≠ Len(constants) − 1 then
3:         throw Error
4:     end if
5:     if Len(variables) = 1 then
6:         return {variables[0] ↦ constants[0]^−1 ∗ constants[1]^−1}
7:     end if
8:     solutions := []
9:     for i ∈ {0 . . . Len(variables)} do
10:        offsets := GetOffsets(constants[i], constants[i + 1])
11:        for offset ∈ offsets do
12:            varValue := CalculateVarValue(constants[i], constants[i + 1], offset)
13:            replacementValue := constants[i] ∗ varValue ∗ constants[i + 1]
14:            newConstants := InsertVarValue(constants, replacementValue, i)
15:            subSolutions = Solve(newConstants, variables.Remove(i))
16:            for subSolution ∈ subSolutions do
17:                solution := subSolution.Assign(variables[i] ↦ varValue)
18:                solutions.Add(solution)
19:            end for
20:        end for
21:    end for
22:    return solutions
23: end procedure
```

Algorithm 2 Calculating all the possible offsets in neighbouring constants

```
1: procedure GETOFFSETS(constant1, constant2)
2:     offsets := []
3:     for k ∈ {0 . . . Len(constant1)} do
4:         for j ∈ {0 . . . Len(constant2)} do
5:             offsets.Add([k, j])
6:         end for
7:     end for
8:     return offsets
9: end procedure
```

Algorithm 3 Calculating variable assignment value

```
1: procedure CALCULATEVARVALUE(constant1, constant2, offset)
2:     leftSubword := Substring(constants[i], offset[0], Len(constants[i]) − 1)
3:     rightSubword := Substring(constants[i + 1], 0, offset[1])
4:     return leftSubword^−1 ∗ rightSubword^−1
5: end procedure
```

Example 5.3. Let $A = \{a, b\}$ and $Y = \{X_1, X_2\}$. We will use the algorithm to find all solutions to the equation $a^2X_1baX_2 = \epsilon$. The constants are $[a^2, ba, \epsilon]$ and the variables are $[X_1, X_2]$.

1. $i = \epsilon$: consider the assignments of X_1 such that (a^2, X_1, ba) forms a singular triple. The possible assignments for X_1 are $\epsilon, a^{-1}, a^{-2}, b^{-1}, a^{-1}b^{-1}, a^{-2}b^{-1}$, and $a^{-3}b^{-1}$.

   (a) Choose $X_1 = \epsilon$: then the equation becomes $a^2baX_2 = \epsilon$ which has solution $X_2 \mapsto a^{-1}b^{-1}a^{-2}$. Add solution $X_1 \mapsto \epsilon,\ X_2 \mapsto a^{-1}b^{-1}a^{-2}$ to the list of solutions.

   (b) Choose $X_1 = a^{-1}$: then the equation becomes $abaX_2 = \epsilon$ which has solution $X_2 \mapsto a^{-1}b^{-1}a^{-1}$. Add solution $X_1 \mapsto a^{-1},\ X_2 \mapsto a^{-1}b^{-1}a^{-1}$ to the list of solutions.

   (c) Choose $X_1 = a^{-2}$: then the equation becomes $baX_2 = \epsilon$ which has solution $X_2 \mapsto a^{-1}b^{-1}$. Add solution $X_1 \mapsto a^{-2},\ X_2 \mapsto a^{-1}b^{-1}$ to the list of solutions.

   (d) Choose $X_1 = b^{-1}$: then the equation becomes $a^3X_2 = \epsilon$ which has solution $X_2 \mapsto a^{-3}$. Add solution $X_1 \mapsto b^{-1},\ X_2 \mapsto a^{-3}$ to the list of solutions.

   (e) Choose $X_1 = a^{-1}b^{-1}$: then the equation becomes $a^2X_2 = \epsilon$ which has solution $X_2 \mapsto a^{-2}$. Add solution $X_1 \mapsto a^{-1}b^{-1},\ X_2 \mapsto a^{-2}$ to the list of solutions.

   (f) Choose $X_1 = a^{-2}b^{-1}$: then the equation becomes $aX_2 = \epsilon$ which has solution $X_2 \mapsto a^{-1}$. Add solution $X_1 \mapsto a^{-2}b^{-1},\ X_2 \mapsto a^{-1}$ to the list of solutions.

   (g) Choose $X_1 = a^{-3}b^{-1}$: then the equation becomes $X_2 = \epsilon$. Add solution $X_1 \mapsto a^{-3}b^{-1}$, $X_2 \mapsto \epsilon$ to the list of solutions.

2. $i = \epsilon$: so we are considering the assignments of X_2 such that (ba, X_2, ϵ) forms a singular triple. The possible assignments for X_2 are ϵ, a^{-1}, and $a^{-1}b^{-1}$.

   (a) Choose $X_2 = \epsilon$: then the equation becomes $a^2X_1ba = \epsilon$ which has solution $X_1 \mapsto a^{-3}b^{-1}$. Add solution $X_1 \mapsto a^{-3}b^{-1},\ X_2 \mapsto \epsilon$ to the list of solutions.

   (b) Choose $X_2 = a^{-1}$: then the equation becomes $a^2X_1b = \epsilon$ which has solution $X_1 \mapsto a^{-2}b^{-1}$. Add solution $X_1 \mapsto a^{-2}b^{-1},\ X_2 \mapsto a^{-1}$ to the list of solutions.

   (c) Choose $X_2 = a^{-1}b^{-1}$: then the equation becomes $a^3X_1 = \epsilon$ which has solution $X_1 \mapsto a^{-3}$. Add solution $X_1 \mapsto a^{-3},\ X_2 \mapsto a^{-1}b^{-1}$ to the list of solutions.

Notice in the above example that solutions 1(g) and 2(a) are identical, as are solutions 1(f) and 2(b). Finding duplicated solutions means that the same solution space is being explored multiple times which is wasteful, especially for quations with higher degrees. To prevent this, we introduce an optimization which keeps track of what solutions have been explored and doesn't continue the search if part of a solution has been found before. This optimization, and a comparison of performance with and without it, is outlined in section 5.3.

All that remains, besides optimizations, is to prove that this algorithm finds exactly all of the possible short solutions to a given equation.

Algorithm 4 Calculating new constants

```
1: procedure InsertVarValue(constants, replacement, varIndex)
2:     newConstants := []
3:     for i ∈ len(constants) do
4:         if i ≠ varIndex and i ≠ varIndex − 1 and i ≠ varIndex + 1 then
5:             newConstants.add(constants[i])
6:         else if i = varIndex then
7:             newConstants.add(replacement)
8:         end if
9:     end for
10:    return newConstants
11: end procedure
```

Theorem 5.3. *Algorithm 1 finds the full set of short solutions to a given linear equation.*

Proof. By induction on the number of variables in the equation.

When the equation has one variable, it has exactly one short solution. The algorithm will return exactly this solution on line 6.

Assume the theorem holds for an equation of n variables and consider an equation of $n+1$ variables.

By Theorem 5.2, under a solution, one of the variables must cancel with its neighbours. The algorithm iteratively picks each of the variables to be the variable that cancels. For each variable X_i, it finds every possible word that would cancel with some part of the constant before it and the constant after it (including the empty word ϵ). These words constitute all possible assignments for X_i in a short solution such that $(\alpha_i, X_i, \alpha_{i+1})$ is a singular triple.

The algorithm, one by one, substitutes X_i with each of these words. Each of these substitutions results in an n variable equation, which by assumption the algorithm finds all short solutions for. Augment each of these short solutions with the particular assignment of X_i that produced the n variable equation it corresponds to. Since the value of X_i was picked so that it only cancels with its neighbours, and not other variables, the resulting solution to the $n+1$ variable equation is still short. Also, since these are all the possible assignments of X_i, this set of solutions is the full set of short solutions to the equation in which reduction begins with the variable X_i. Since the algorithm repeates this process for every variable, the algorithm finds every possible short solution to the $n+1$ variable equation.

So by induction, the algorithm finds the full set of short solutions to any given linear equation. □

5.3 Complexity Analysis

This algorithm is not efficient. It recurses to a depth of the number of variables, and must explore each possible order in which the variables are assigned a value. Additionally, in each level of recursion, it must perform reasonably complex operations such as calculating the offsets of neighbouring constants, taking inverses, and performing multiplications of different subwords.

There is, however, one simple optimization that ensures no solution is considered twice. Before beginning execution, a global empty list is created to keep track of every variable assignment seen by the algorithm. If any variable is ever given the exact same assignment as at some point earlier in execution, then the recursive call and that assignment of the variable is skipped. A variable receiving the same assignment a second time indicates that the algorithm will be exploring the same solutions a second time. Every time solutions are found, they are added to the global list. While this optimization in practice makes the algorithm faster, it does not affect the average case time complexity.

Let x denote the number of variables in the equation, and assume we have an average case scenario where $x > 1$. Then the algorithm enters the loop at line 9 in algorithm 1 and for each variable, must calculate every possible valid assignment. For simplicity we can assume that every constant has the same length n. Then the function `GetOffsets` has a time complexity of n^2, assuming that the `add` operation can be finished in constant time.

Also, the function returns n^2 elements. For each of these elements the algorithm must construct the assignment for the variable and generate the new equation. While these operations are not negligible, cost-wise, they are small in comparison to the overall cost of the entire algorithm. For simplicity we will call the overall average time it takes for these operations to complete m. Then the algorithm recurses: if $T(x, n)$ is the function for the time complexity of the algorithm, where x is the number of variables, and n is the length of the constants, then the recursive step takes $T(x-1, n)$ time. Lastly an assignment operation and an add operation, which are assumed to be constant in time, are executed for each of the (say l) partial solutions. The resulting equation for the time complexity on an equation with x constants is

$$T(x, n) = x(n^2 + n^2(m + T(x - 1, n) + l)).$$

This equation is difficult to expand, but with some simplification, we can approximate the dominating term. Since m and l are assumed to be relatively trivial, we set them both equal to 1.

$$\begin{aligned} T(x, n) &= xn^2 + xn^2m + xn^2T(x - 1, n) + xn^2l \\ &= 3xn^2 + xn^2T(x - 1, n). \end{aligned}$$

Since the second term is going to be much larger than the first, we discard the first, leaving $T(x) = xn^2T(x - 1, n)$. This recursively expands to $T(x) = x!xn^2$.

This is a simplification. The lengths of the constants in the equation will not always be the same, and in every depth of the recursion, these lengths will change. In reality, the precise time complexity of an equation with n constants is determined by $n + 1$ variables: the individual length of each of the n constants and the number of variables. However, of these $n + 1$ variables, the one that occupies the most dominant term asymptotically in T is the number of variables as it dictates the depth of the recursion.

To back up these assertions, timed experiments were run on the algorithm with randomly generated equations in which one of the two variables (constant size, assumed to be the same for all constants, and number of variables) varied. The experiments were run using the algorithm both with and without the optimization described above. The results are shown in figure 2.

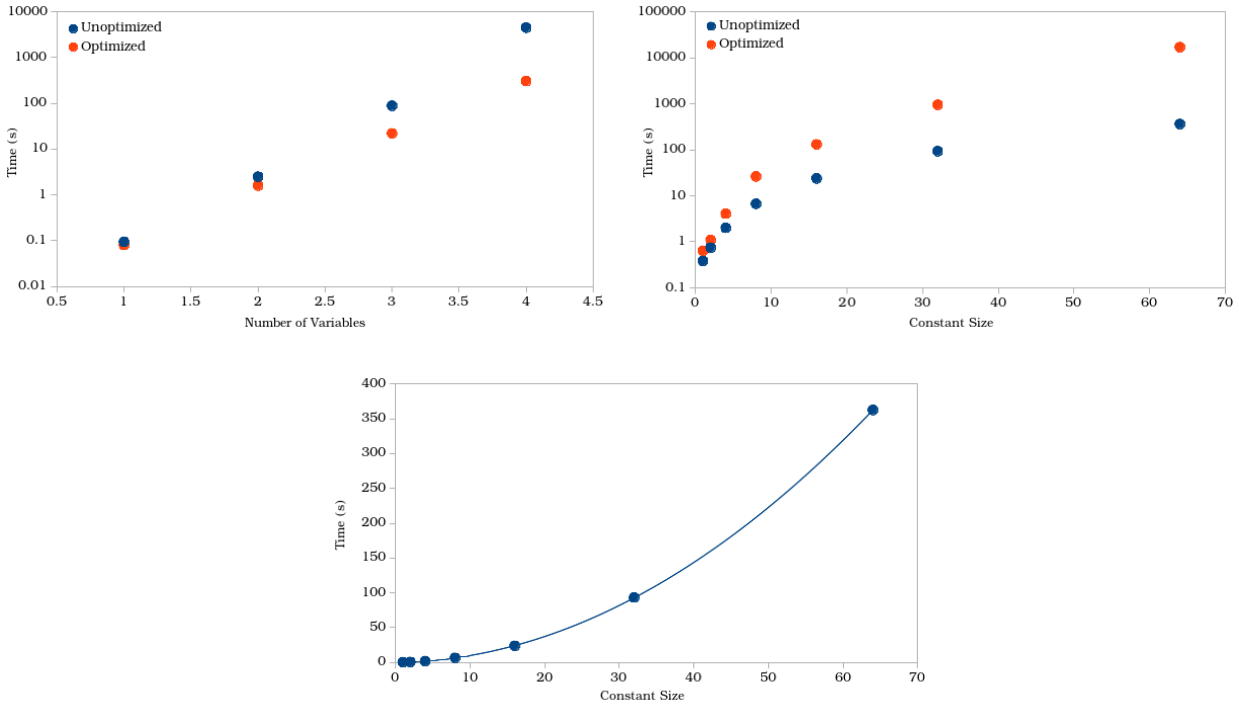


Figure 2: Left: the amount of time taken to find solutions to an equation with a constant size of 3, a varying number of variables, and an alphabet size of 2. Right: the amount of time taken to find solutions to an equation with a varying constant size, 2 variables, and an alphabet size of 2. Bottom: the same data as the unoptimized algorithm from the top right graph, but on a linearly scaled Y-axis and with a quadratic line of best fit. Time in seconds.

These graphs support the theoretical complexity determined above: solve time is exponential with the number of variables and polynomial with the size of the constants (as demonstrated by the bottom graph of figure 2). However, these results also show that the optimization is detrimental for equations with a small numbers of variables, since the cost of checking whether each solution has been seen before outweighs the benefits of pruning the search tree. Additionally, the longer the constants of the equation, the greater the overhead of the optimization since bigger constants means a greater number of solutions to check. In contrast, when there are more variables the benefits of cutting the search short are greater and the time spent checking each solution is more than made up for.

Perhaps unsurprisingly (as it was never mentioned in the theoretical analysis), figure 3 demonstrates that the alphabet size makes no significant difference in both the amount of time it takes to find solutions and the difference in performance of the optimized and unoptimized algorithms.

5.4 Enumerating Solutions

Now we turn to the question of counting the number of solutions to any given linear equation. As in the previous section, the problem is simplified by assuming the length of all

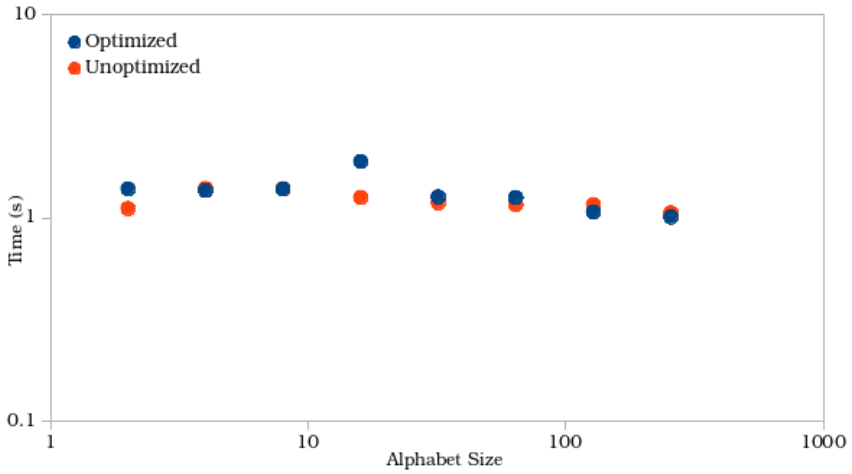


Figure 3: The amount of time taken to find solutions to an equation with a varying alphabet size, 2 variables, and constant size 2. Time in nanoseconds.

constants is the same. Even with this simplification, analytically determining a formula for the number of solutions is complicated. If the constant length is n, each variable can start the reduction process in n^2 different ways, leaving an equation of $n-1$ variables with constants of varying lengths. And so the number of solutions to this shorter equation must be found, and so on, in a recursive method similar to that of determining the time complexity of the algorithm. Instead, it is easier to empirically examine how many solutions the algorithm finds for different sizes of equations.

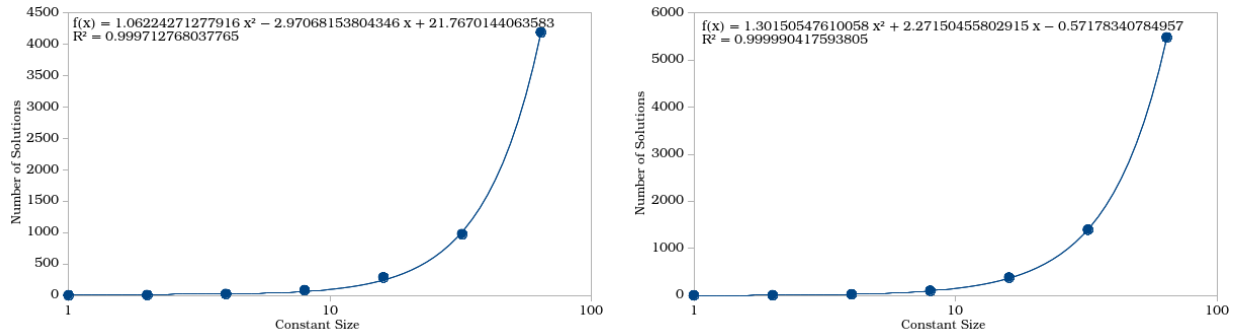


Figure 4: The number of solutions to an equation with a varying constant size, 2 variables (left), 3 variables (right), and an alphabet size of 2. Best fit line and coefficient of determination given in both.

Figure 4 shows that the number of solutions is quadratically related to the size of the constants in the equation. Even when the number of variables in the equation is varied, the relation remains quadratic. This is parallels the relation between constant size and time complexity, but the similarities end here.

The number of variables in the equation does not always have an exponential effect on the number of solutions, and the equation that models the number of solutions depends on the size of the constants. With a constant size of 3, the number of solutions grows logarithmically

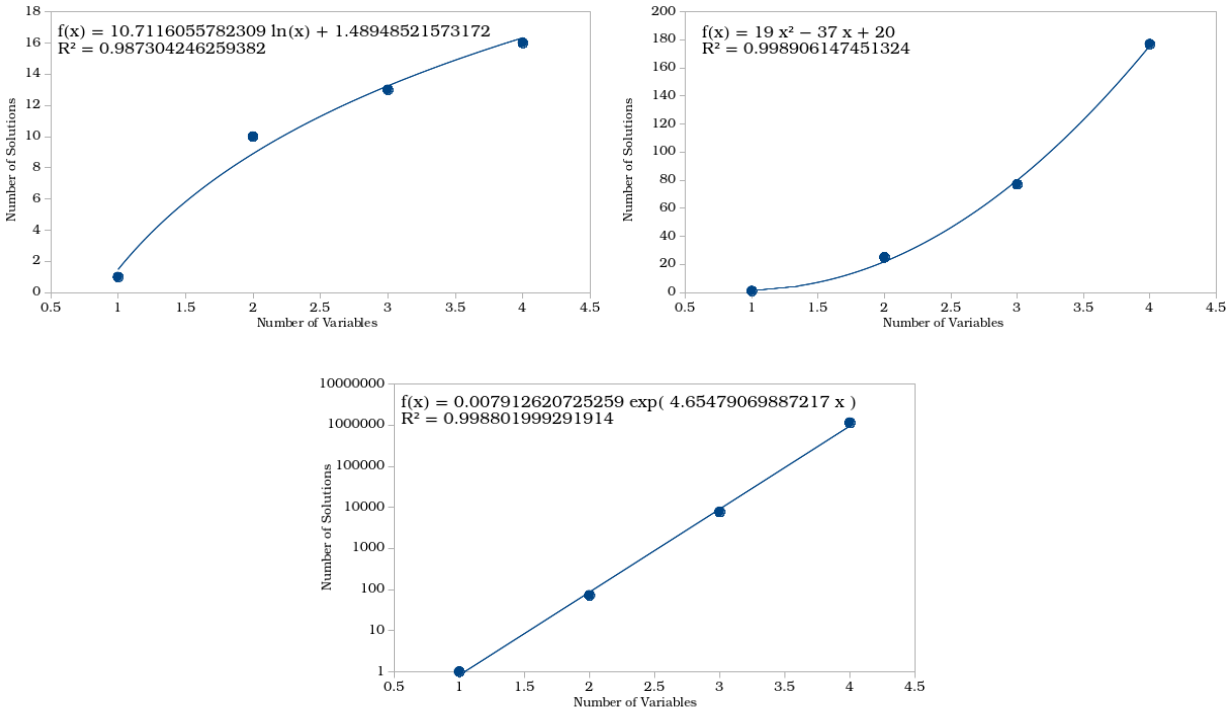


Figure 5: The number of solutions to an equation with a constant size of 3 (top left), a constant size of 4 (top right), a constant size of 5 (bottom), a varying number of variables, and an alphabet size of 2. Note the bottom graph has a logarithmically scaled Y-axis. Best fit line and coefficient of determination given in all three.

with the number of variables, with a constant size of 4, it grows quadratically, and with a constant size of 5, it grows exponentially. So with each increase in constant size, the function modelling the number of solutions, with respect to the number of variables, jumps an entire complexity class.

Additionally, the alphabet size has no effect on the number of solutions. The number of solutions is limited by the length of the constants, and is unaffected by what particular letters occupy them. For instance the word *aaaa* has just as many subwords as the word *abcde* and the algorithm only considers the number of subwords of each constant.

Overall, a universal equation for the number of solutions to a linear equation would have to take into account the number of variables, the size of each individual constant, and the points at which the search can find identical solutions. The data above demonstrates that the number varies quadratically with the size of the constants, regardless of the number of variables. However, depending on the size of the constants, the number of solutions grows at wildly different rates with respect to the number of variables.

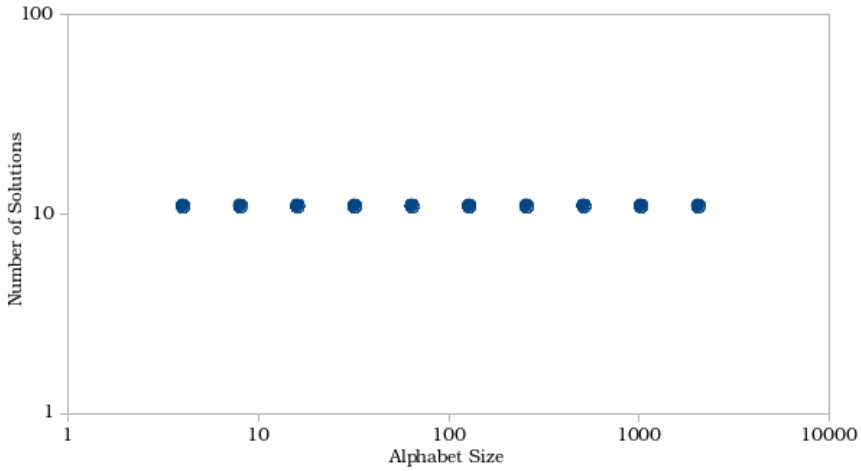


Figure 6: The number of solutions to an equation with a varying alphabet size, 2 variables, and constant size 2.

6 Quadratic Equations

This section covers quadratic equations, providing the mathematical background necessary to show that a constraint solver can be applied to a quadratic equation, describing the constraints needed to model the problem, and proving that the constraints are sufficient. Additionally, the constraint solver is tested on random samples of equations and its performance measured and compared against itself on different types of equations and against the algorithm for finding solutions to linear equation from section 5.

6.1 Mathematical Background

Quadratic equations are, understandably, more difficult to solve than linear equations. Unlike the linear equations, quadratics do not always necessarily have a solution, and whether or not they do is difficult to determine. Indeed, the problem of solvability has been proved to be NP-Complete. [7] So instead of creating a bespoke, deterministic algorithm for finding solutions to quadratic equations, we instead apply a common trick of mapping the problem to a different NP-Complete problem: constraint solving.

However, to use a constraint solver, there needs to be a limit on the size of the solutions to the equations. For linear equations, we defined a "short" solution and proved that every linear equation must have one. The same definition of short would be useful for quadratic equations, since the lengths of the constants could then be used as an upper bound on the size of the solutions. However, it is not immediately clear that if a quadratic equation is solvable, it must have a solution with bounded size. In *A Polynomial Bound on Solutions of Quadratic Equations in Free Groups*, Lysenok and Myasnikov prove that there is such a bound, though it is much looser than the bound on solutions to linear equations. [9]

Definition 6.1. Let $\alpha_1, \alpha_2, \ldots, \alpha_n \in F(A)$ be the constants of an equation $\omega \in F(A \cup Y)$.

We will denote the size of the equation by

$$c(\omega) = \sum_{i=1}^{n} |\alpha_i|.$$

Also, we will denote the number of variables in the equation by $v(\omega)$.

The bound proven by Lysenok and Myasnikov is dependent on the form of the equation. There are four possible bounds, depending on how "well behaved" the equation is, but we will only present the two loosest bounds here.

Definition 6.2. A quadratic equation ω is called *orientable* if every variable X occurring in ω occurs with two opposite exponents, once as X and once as X^{-1}, and *non-orientable* otherwise. [9]

Theorem 6.1. *Let $\omega = \epsilon$ be a quadratic equation that is solvable. Then there exists a solution S such that for any variable X of ω,*

$$|(x)S| \leq \begin{cases} 40v(\omega)c(\omega) & \text{if } \omega \text{ is orientable.} \\ 150v(\omega)^2c(\omega) & \text{if } \omega \text{ is non-orientable.} \end{cases}$$

Proof. See Lysenok and Myasnikov [9]. □

The tighter the bound is, the more efficient the constraint solver: so an orientable equation will be more efficient than a non-orientable equation, and an equation in standard form will be much more efficient (the two stricter bounds proven by Lysenok and Myasnikov are dependent on the equation being in a defined standard form). It is theoretically possible for every quadratic equations to be put into this standard form in polynomial time, with respect to the size of the equation, [7] but it is a completely separate problem in itself that has been omitted from this project for simplicity.

6.2 Constraints

By Theorem 6.1, a solvable quadratic equation is guaranteed to have at least one solution with a defined maximum size. This means that we can specifically search for solutions that satisfy this bound and still answer the solvability question with the presence or lack of a solution. So it is possible to apply a constraint solver to the problem.

Definition 6.3. A constraint satisfaction problem is a triple $\langle X, D, C \rangle$ where X is an n-tuple of variables $x_1, x_2, \ldots, x_n$, D is a corresponding n-tuple of domains $D_1, D_2, \ldots, D_n$ such that $x_i \in D_i$, and C is a t-tuple of constraints $C_1, C_2, \ldots, C_t$. A constraint C_j is a combination of valid values for a set of variables $S_j \subseteq X$. A solution to a constraint satisfaction problem is an assignment of values to the variables in $S_1, \ldots, S_t$ such that all the constraints $C_1, \ldots, C_t$ are satisfied. [11]

A constraint solver is a general purpose algorithm used to solve constraint satisfaction problems. Common examples of such problems are timetable scheduling and route planning. Most constraint solvers are based on a brute force backtracking search: a simple depth-first traversal of a search tree, where each node represents a value for each of the variables in the problem. Branches can be cut off early when it is obvious they don't satisfy the constraints of the problem, and there are numerous other optimization techniques that make constraint solvers efficient enough to be practical for many NP-Complete tasks. [11] Importantly, the number of variables, the number of constraints, and the domains of the variables of a problem must all be finite to be classified as a constraint satisfaction problem. Since solvable quadratic equation must have at least one finitely bounded solution, they can be represented by a finite number of variables, and therefore as a constraint satisfaction problem.

The constraint solver used in this project is Google OR-Tools [5], and the model and input parsing is implemented in Python. An equation is represented as a list of letters (represented as integers for simplicity), called E, and a mapping from indices of the list to other indices in the list, called M. The variables in E have a domain as large as the size of the alphabet, and the elements of M have a domain the size of the array E. The list of letters, E, represents the equation by storing precisely one letter at each index. Every letter in the alphabet is assigned an integer value, and a letter's inverse is represented by the negative of its integer mapping. The number 0 represents the identity. The mapping M of indices indicates which letters cancel with each other when the equation is reduced. For instance, a mapping $i \mapsto j$ indicates that at some step of the reduction of the word that E represents, the letter at index i cancels with the letter at index j.

The size of E is equal to size of the equation (i.e. the sum of the lengths of all the constants) plus the product of the number of variables with the upper bound on size of the variables. So if the size of the equation is n and there are m variables,

$$|E| = \begin{cases} n + 40m^2n & \text{if the equation is orientable.} \\ n + 150m^3n & \text{if the equation is non-orientable.} \end{cases}$$

These are the variables and domains of the problem, but now we must also formulate constraints on the elements in E and on the mappings in M that if satisfied, imply that a solution to the equation has been found.

Firstly, each of the constants of the equation must be fixed: i.e. for each constant letter a_i at position i in the equation, the constraint

1 $E[i] = a_i$

must hold. Also, the values at each of the indices of E that are part of the same variable must be equal. Or, if they are part of an inverse, must be in reverse order and inverted. Note that we can say without loss of generality that for a variable X in any equation, there is at least one instance of X^{+1}, since if there are none, we can simply apply an A-map that maps $X \mapsto X^{-1}$ and obtain an equivalent equation that does contain X^{+1}.

2 For each variable v with exponent $+1$ that occurs at indices $i_1, i_2, \ldots, i_n$, for all $j \in \{0, 1, 2, \ldots, N\}$ where N is the upper bound on the size of the variable,

$$E[i_1 + j] = E[i_2 + j] = \cdots = E[i_n + j].$$

In other words, the letters at each position of a variable with a positive exponent are the same in every location in the equation that the variable occurs.

3 For each variable v that occurs with exponent $+1$ at index i, for each occurrence of v with exponent -1 that occurs at indices $i_1, i_2, \ldots, i_n$, for all $j \in \{0, 1, 2, \ldots, N\}$ where N is the upper bound on the size of the variable,

$$E[i + j] = -E[i_1 + N - j] = -E[i_2 + N - j] = \cdots = -E[i_n + N - j].$$

In other words, the letters at each position of an inverted variable correspond to the inverted letters in reversed order of the non-inverted variable.

Note that when calculating the position of a letter in E, if N is the upper bound on the size of variable values, each variable takes up N spaces of E. The remainder of the constraints apply to the mapping M.

4 $M[M[i]] = i$: if the ith letter cancels with the jth letter, then the jth letter cancels with the ith letter.

5 $E[i] = -E[M[i]]$: the letter that the ith letter cancels with must be its inverse.

6 $E[i] = 0 \implies M[i] = i$: if a letter is the identity, then it cancels with itself.

7 For all $i \in \{1 \ldots |E|\}$, $i < M[i] \implies$ for all $j \in \{i + 1, \ldots, M[i] - 1\}$, $i < M[j]$ and $M[j] < M[i]$: all the letters between any two letters that cancel with each other (say at indices i and $M[i]$) cancel only with other letters that occur between indices i and $M[i]$.

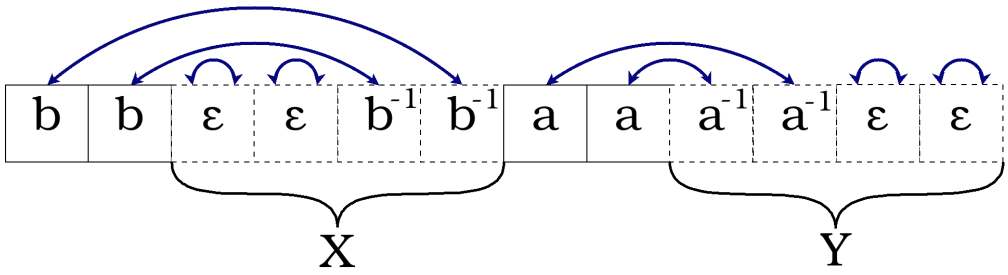


Figure 7: An example of an array E, with a valid solution to the constraint satisfaction problem that corresponds to the equation $bbXaaY$. The arrows indicate the mappings of M. The segments in dotted lines indicate the parts of E that represent a variable value.

If the solver finds an assignment of values for the elements in E that satisfies all of these constraints, then it has found a solution to the equation. But to prove this, we first need the following sub-lemmas.

Lemma 6.1. *If an array E and an accompanying mapping M, as described above, satisfy constraints 1 through 7, then E must contain an even number of non-identity elements.*

Proof. Suppose E contains $2n+1$ non-identity elements, for some natural number n. Since E and M satisfy the constraints, we know that every nonzero element a must be mapped to its inverse a^{-1}, and likewise a^{-1} to a. These mappings occur in pairs, so can at best account for $2n$ of the non-identity elements. This leaves a single non-identity element, say b, which by the definition of M must be mapped to another element in E. However, its inverse b^{-1} cannot occur, still unpaired, in E since b is the only remaining unpaired non-identity element. So the constraints are violated and we have a contradiction. □

Lemma 6.2. *Let E and M be a list and a mapping, as described above, that satisfies constraints 1 through 7. Let E contain at least two non-identity elements. Then there must exist a pair of non-identity elements that are assigned to each other in M, at indices i and $M[i]$, such that either i and $M[i]$ are adjacent or the only elements separating the elements at i and $M[i]$ are identity elements.*

Proof. Consider an array E with $2k$ non-identity elements and a mapping M that satisfies constraints 1 through 7. Since there are at least 2 non-identity elements in E, we can pick a pair of elements a and a^{-1}, occurring at indices i and $M[i]$, which are mapped to each other in M. Suppose i and $M[i]$ are not adjacent and are not separated by only identity elements. Then, since constraint 7 is satisfied, there is at least one pair of non-identity elements between indices i and $M[i]$ that are mapped to each other in M. Reassign i and $M[i]$ to the indices of this new pair of non-identity elements. Now if these elements are adjacent or separated only by identity elements, then we are done. If they are not, then we repeat the first step: since constraint 7 is satisfied, there must be another pair of non-identity elements which are mapped to each other in M between the newly reassigned indices i and $M[i]$. Reassign i and $M[i]$ again to these new indices. And repeat this process until either the elements at i and $M[i]$ are adjacent or are only separated by identity elements. Since there are a finite number of non-identity elements in E, this must eventually occur, and we have found a pair of non-identity elements in E which are mapped to each other in M and are either adjacent or separated by non-identity elements. □

Theorem 6.2. *An equation over a finite freely presented group has a solution with a maximum variable value size of N if and only if it can be represented as an E and M structure, as described above, that satisfies constraints 1 through 7.*

Proof. Suppose $\omega = \epsilon$ is a solvable equation. By assumption, we know that ω has a solution S such that for each variable X_i, $(X_i)S$ is bounded by some natural number N. Construct an array E such that $|E| = c(\omega) + Nv(\omega)$. Fill in each index of E with every letter of each constant consecutively. When a variable is reached, fill in E with the image $(X_i)S$ (or the inverse $(X_i)S^{-1}$). If this image has a size less than N, fill in the remaining $N - |(X_i)S|$ indices with the identity element. Clearly this E satisfies constraints 1, 2, and 3.

Now in M, pair each non-identity element in E with the element that it would cancel with in some reduction of the word $(\omega)S$. Notice that in the reduction of a word over a freely presented group, for the ith and jth letters to reduce at some stage of the reduction, the elements between i and j must reduce first. This corresponds to constraint 7. Further, the way the mappings in M have been assigned guarantees that letters are paired with their inverse and that the mapping is reflexive, satisfying constraints 4 and 5. Lastly, in M, map every identity element in E to itself. Then the last constraint, 6, is satisfied.

Now suppose that we have a filled-in array E, whose values are restricted by the constants and the variables of a quadratic equation ω, and a mapping M that satisfies the above constraints. We wish to prove ω is solvable by induction on the number of non-trivial letters in E.

If E contains only identity elements, then the only word represented by E is the empty word ϵ. Assume that all words represented by an E_{2k}, with $2k$ non-identity elements (since by Lemma 6.1, $|E|$ must be even), and a mapping M_{2k}, that satisfy the constraints, reduce to ϵ.

Suppose E_{2k+2}, an array of $2k+2$ non-identity elements, together with a mapping M_{2k+2}, satisfy the constraints. By Lemma 6.2, there exists at least one pair of elements in E_{2k+2} that are mapped to each other in M_{2k+2} and are either adjacent or separated only by identity elements. Pick such a pair, say at indices i and $M_{2k+2}[i]$.

Cancel the elements at i and $M_{2k+2}[i]$ by replacing them with the identity and setting $M_{2k+2}[M_{2k+2}[i]] = M_{2k+2}[i]$ and $M_{2k+2}[i] = i$, creating a new E'_{2k+2} and M'_{2k+2} with $2k$ non-identity elements. Note that the only difference between E'_{2k+2} and E_{2k+2} is that the elements located at $M_{2k+2}[i]$ and i have been replaced with the identity and mapped to themselves in M'_{2k+2}. So constraints 4, 5, and 6 are still satisfied.

Now notice that since the elements at i and $M'_{2k+2}[i]$ map to themselves, they do not map outside of the range $[i, M_{2k+2}[i]]$. Also, in the original mapping, M_{2k+2}, everything between and outside of indices i and $M_{2k+2}[i]$ satisfied constraint 7. So in M'_{2k+2}, constraint 7 is still satisfied, since every element of E'_{2k+2} with an index in the range $[i, M_{2k+2}[i]]$ still cancels with elements from within this same range.

Since E'_{2k+2} and M'_{2k+2} satisfy the constraints, by the inductive hypothesis, any word represented by them reduces to ϵ, and so any word represented by E_{2k+2} and M_{2k+2} also reduces to ϵ.

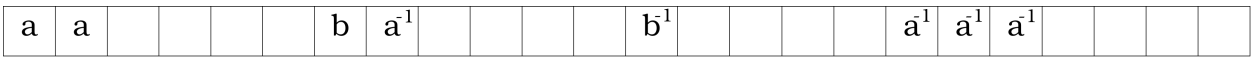
Since the word represented by an array E and a mapping M that satisfy the constraints must reduce to ϵ, there must exist values for the variables of any equation represented by E and M that when substituted into the equation, reduce the equation to ϵ. And so any equation represented by E and M must be solvable. □

Since finding a solution to an equation $\omega = \epsilon$ is equivalent to finding a corresponding E and M that satisfy the constraints, we can use the terminology of each interchangeably.

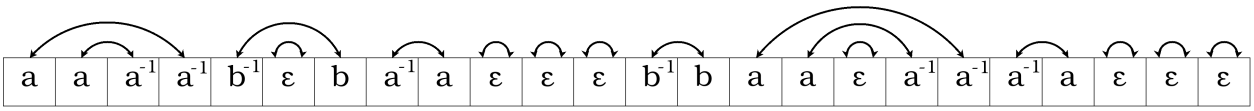
Example 6.1. Let $\omega = a^2Xba^{-1}Yb^{-1}X^{-1}a^{-3}Y$ and notice that $\omega = \epsilon$ has the solution

$$S = \begin{cases} X \mapsto a^{-2}b^{-1} \\ Y \mapsto a \end{cases}.$$

We will show that this solution can be turned into an array E and a mapping M that satisfies the constraints by constructing E and M. Note $c(\omega) = 8$ and $v(\omega) = 2$. For brevity, we will say that the upper bound for the size of the variables is 4, though using the non-orientable bound, it should be 4800. So the array E has length 24: 8 spaces for the constants of the equation, and 16 spaces for the variables. The indices of E that correspond to a constant in the equation are constrained to be precisely that constant, leaving the spaces for the variables empty, as demonstrated in the diagram below.



Next, fill in the variable slots with the values $(X)S$ and $(Y)S$.



We can see that the elements of E, when written as a word over the alphabet $\{a, b\}$, reduce to ϵ, and also, taking the arrows to represent the mappings in M, we can see that each of constraints are satisfied: each letter is mapped to its inverse, if a letter at index i is mapped to a letter at index j, then the letter at j is mapped to the letter at i, every identity letter maps to itself, and all letters between a mapped pair cancel with other letters between the indices of the pair.

The constraint solver does not actually distinguish between quadratic and non-quadratic equations. As long as a the number of variables needed to model the equation as a constraint satisfaction problem is bounded by a known finite number, then the solver will find a solution, given enough computation time. So the solver will also find all solutions to linear equations, and for any other class of equation that has the property that solvability implies the existence of a bounded solution. In the Python package, there are three different bounds for an equation ω implemented: $c(\omega)$, $40v(\omega)c(\omega)$, and $150v(\omega)^2c(\omega)$. The first is suitable for linear equations, the second for orientable quadratics, and the latter for non-orientable quadratics. Using a lower bound than the equation class necessitates is possible (i.e. using the linear bound $c(\omega)$ for a quadratic problem), but does not guarantee that if the equation is solvable, a solution will be found. In practice however, the larger two bounds take an unreasonable amount of time and memory to run on a standard computer, and the randomly generated solvable equations used in the following experiments are guaranteed to have solutions within the linear bound.

The constraints outlined above are enough to guarantee a solution is found if they are satisfied, and to guarantee that they will be satisfied if there is a solution. However, they may produce "undesirable" solutions, i.e. solutions which are not reduced, or multiple solutions which are identical once reduced.

To prevent this, the following constraints are also in place:

8 If $E[i]$ is part of a variable with exponent $+1$, is not the last letter of a variable, and $E[i] = 0$, then $E[i + 1] = 0$. This ensures that multiple assignments of E that are all actually the same, just padded with identity elements at different indices, are not considered. Instead, the only valid solutions are ones in which all of the identity elements are at the end of the non-inverted variable.

9 If $E[i]$ is part of a variable with exponent -1, is not the first letter of the variable, and $E[i] = 0$, then $E[i - 1] = 0$. This constraint corresponds to constraint 8, but for variables of negative exponent. Since all the identity elements in a variable of positive

exponent must appear at the end of the variable, when the variable is inverted (i.e. the letter order reversed and each letter inverted), the identity elements must appear at the beginning.

10 For each variable occurring at index i, and for all $j \in \{1, 2, \ldots, N\}$ where N is the bound on the size of the solution, if $E[i + j - 1]$ is not the identity element, then $E[i + j - 1] \neq -E[i + j]$. Since all identity elements must be at the beginning or end of the variable, this constraint ensures that no letters in a variable are immediate neighbours with their inverse, which as a consequence ensures that the variable values are reduced.

These constraints keep the solver, when searching for all solutions to an equation, from considering hundreds or thousands of results that have differing values in E and M but are actually equivalent as reduced words over an alphabet. However, they also place additional rules on what form an acceptable result can take, making it harder to find the first solution.

Theorem 6.3. *Suppose there is an array E and a mapping M that satisfies constraints 1 through 7 for some equation $\omega = \epsilon$. Then there is also an array E' and a mapping M' that satisfies constraints 1 through 10 with respect to the same equation.*

Proof. If the variable values in E have no identity elements in them, then constraints 8 and 9 are trivially satisfied. Suppose the variable values in E do contain identity elements. Then form E' by moving all the identity elements occurring in variables with a positive exponent to the end of the variables they are in and form M' by modifying M so that all the same elements that were mapped to each other in E are still mapped to each other in E' (since the elements' indices will be different in E'). Also, move all of the identity elements that occur in the inverted variable values to the beginning of the variable, and alter M' accordingly. Then constraints 8 and 9 are satisfied. Also, identity elements cancel with themselves so their location within a variable value does not change the value of the variable. So the solution represented by E' is equivalent to the solution represented by E, and therefore E' represents a solution to $\omega = \epsilon$. Since E' and M' represent a solution to the equation, by Theorem 6.2, they satisfy constraints 1 through 7.

Now suppose that the variable values of E contain sequences of letters of the form aa^{-1} or $a^{-1}a$. If a pair of such letters are mapped to each other in M, replace them both with the identity and alter M so that they map to themselves. Repeat this step for every such pair, creating a new array E' and a new mapping M'. E' and M' necessarily still satisfy constraints 1 through 7.

Suppose there are pairs of these letters that are not mapped to each other in M. Since E represents a word over a freely presented group, and since by Lemma 4.1 the reduction of a word is unique, regardless of what order the steps of reduction are taken in, we can choose to reduce each of the aa^{-1} and $a^{-1}a$ pairs with each other first and still obtain the same reduced word. So there is some valid mapping M^* in which each of these pairs are mapped to each other such that E and M^* still satisfy the constraints. Then we can repeatedly follow the process described above until there are no such subsequences of letters left in E to form E' and $M^{*\prime}$. By the above, E' and $M^{*\prime}$ will still satisfy constraints 1 through 7, and will additionally satisfy constraint 10. □

6.3 Complexity Analysis

In this section we will first profile the speed of the solver on the solvability problem for randomly generated equations. As with linear equations, the problem is simplified by making lengths of all the constants equal, and the tests are run varying either the constant size or the number of variables.

The first experiments are with linear equations, for comparison with the algorithm described in section 5.2. The constraint solver in general will perform worst when an equation does not have a solution, as it must exhaust the entire search space before it can conclude that the equation is unsolvable. However, all linear equations have solutions, so this shortcoming is not encountered in the results, visualized in figure 8.

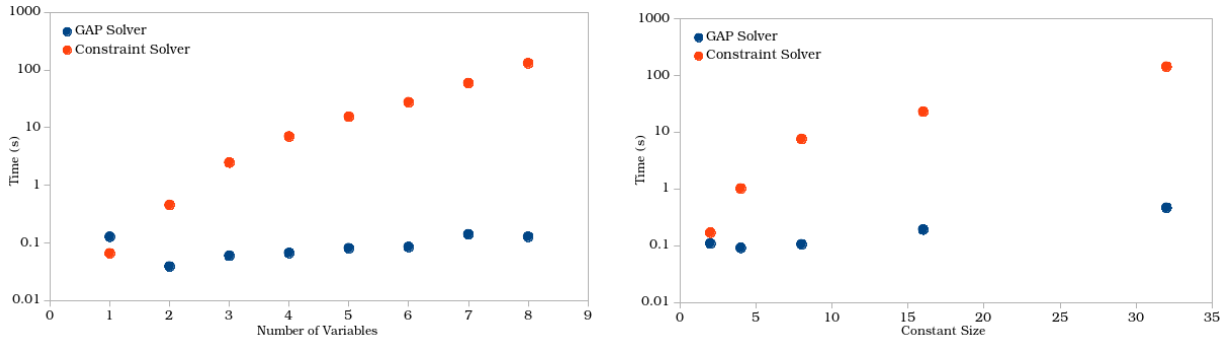


Figure 8: The amount of time taken, in seconds, for each of the constraint solver (orange) and the GAP linear equation solver (blue) to find a single solution to an equation of varying constant size and 2 variables (left) and varying number of variables and constants of length 2 (right). Alphabet size fixed for both at 3.

Regardless, the GAP solver far outstrips the constraint solver. Figure 8 shows that the constraint solver takes upwards of 10 seconds to find a solution to an equation with a constant size of at least 8, or to an equation with at least 4 variables, while the GAP solver can do both in less than a second. In this experiment, the GAP solver never took longer than a second to find a solution - in its worst case, the constraint solver took over 100 seconds. Also, notice that in this experiment, the constraint solver uses the tightest bound for variable size available (the linear bound) and is still considerably slower than the deterministic GAP algorithm. Solving a longer quadratic equation using the larger bounds from Theorem 6.1 would make the solver even slower.

For unsolvable quadratic equations (of which there are many), the solver is forced to explore the full search space, instead of ending as soon as a solution is found, resulting in the solver's slowest behaviour. Without any optimizations, it was immediately obvious that with the orientable and non-orientable bounds on the solutions, execution time was so unreasonable on randomly generated quadratic equations (which are almost always unsolvable), that no data could be gathered. Instead, the linear bound is used.

Tests were run using two types of randomly generated equations: equations that are guaranteed to have a solution, and equations that aren't. Random solvable equations are created by generating a pair of neighbouring constants for every variable in the equation. Whenever a variable is randomly selected to occur in the equation, it has the same two

neighbouring constants, each always occurring on the same side of the variable. When the variable is inverted, the neighbouring constants swap sides and are also inverted. This guarantees that the equation is solvable since there is at least one solution, formed by assigning each variable to the product of the inverses of the neighbouring constants. Notably, this solution has size equal to the lengths of the constants, so can be found using a linear bound. The constants, the order of the variables in the equation, and whether or not a variable is inverted are all randomly decided (though every variable is guaranteed to occur at least once with exponent +1). Equations that aren't guaranteed to have a solution are generated similarly, but without fixing the values of each variable's neighbouring constants.

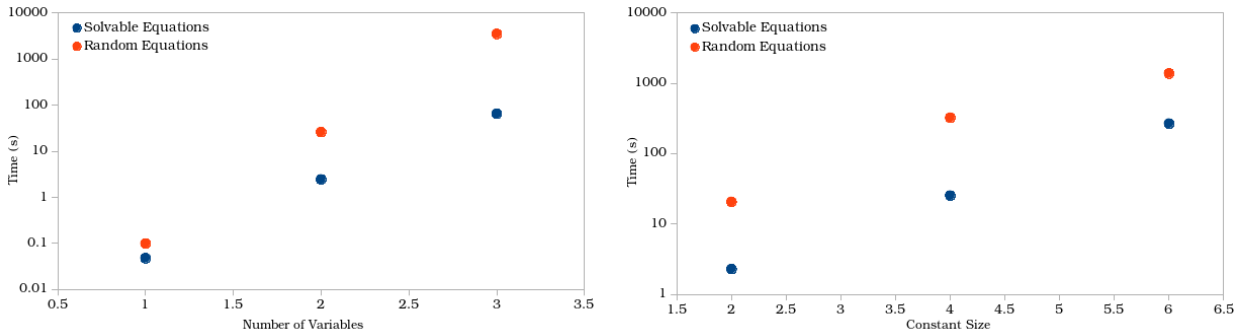


Figure 9: The amount of time taken, in seconds, for the constraint solver to find a solution to randomly generated solvable quadratic equations (blue) and to completely randomly generated quadratic equations (orange), for equations with varying number of variables (left) and varying constant size (right). Alphabet size is fixed in both at 3.

Note that since both problems require an exhaustive search through the full search space, determining that an unsolvable equation is unsolvable and finding all solutions to a solvable equation take the constraint solver approximately the same amount of time. So experiments were only run on the solvability problem since the time taken to evaluate random equations closely reflects the time taken to find all solutions to a random solvable problem.

Figure 9 compares the performance of the solver in determining solvability of completely randomly generated equations versus its performance in determining solvability of randomly generated solvable equations. As expected, the solver's performance on solvable equations was significantly superior to its performance on completely random equations (which were highly likely to be unsolvable). Figure 9 shows that the constraint solver is exponential with both the number of variables and the size of the constants. This is due to the bound on the size of the variable being directly proportional to the sizes of the constants: the more constants and the more variables there are, the more elements of the array E there are for the constraint solver to find a valid value for.

One possible solution to this problem is to attempt to solve the equation with smaller variable bounds which will be faster to run, but might not guarantee the existence of a solution. For instance, if an equation has a maximum variable value size of 10, start by running the solver with a bound of 5. Then if no solutions are found, try again with a bound of 7. And so on, until either a solution is found or the maximum bound is reached and the equation is determined to be unsolvable. For the purpose of referencing, this method

is called the incremental bounds method. It still guarantees that eventually a solution (if there is one) will be found, but also creates the possibility of finding a small solution much more quickly than it could have been found otherwise. The consequences, however, are that computation can be repeated and wasted: if the solution does require a large bound, or if the problem is unsolvable, then many searches on many different bounds (including the maximum) will be performed with no successful results, instead of just one search using the maximum. So for unsolvable equations, this method is strictly inferior.

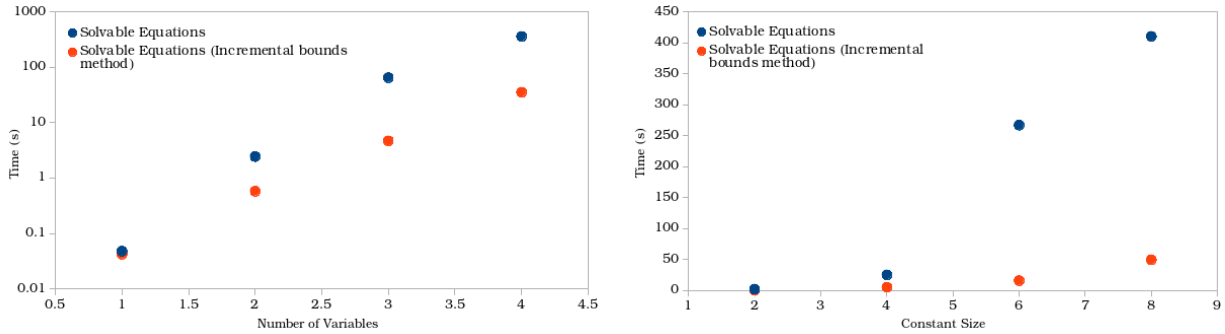


Figure 10: Amount of time taken, in seconds, to find a solution to randomly generated solvable equations with (blue) and without (orange) using incrementally increasing variables bounds, with constants of length 2 and varying number of variables (left) and with 2 variables and varying constant size (right). Alphabet size fixed at 3.

The benefits of the incremental bounds method for solvable equations, though, are demonstrated in figure 10. There is a sizeable difference in runtime between the solver that uses incremental bounds and the one that doesn't, for both varying number of variables and varying constant size. Additionally, the gap in runtime grows as the equation grows in size, indicating that the benefits of the method on large equations is potentially huge. However, use of this method will heavily slow down runtime for unsolvable equations. Similarly, this method does not help speed up the process of finding all solutions to an equation, as the solver still needs to search for solutions of the maximum possible size, regardless of whether it has previously found smaller solutions.

The incremental bounds method helps the solver determine solvability faster, provided the equation is solvable. There are also shortcuts to determine when an equation is unsolvable. It is sometimes possible to detect when a problem is unsolvable before ever even modelling it as a constraint satisfaction problem, allowing the solver to return a result early without ever performing a search.

Theorem 6.4. *Suppose $\omega = \epsilon$ is a solvable quadratic equation. Then the number of constants in ω is even.*

Proof. Suppose an equation $\omega = \epsilon$ is solvable with a solution S and has an odd number of constants. Let E be the array and M the mapping which correspond to S. By Lemma 6.1, the number of non-identity elements in E must be even. Since in E there are an odd number of non-identity elements in the constants, there must then be an odd number of non-identity elements in E contributed by the variables (under the solution S). However, ω is quadratic,

so each variable occurs precisely twice, so there must be an even number of non-identity elements in the variable values. This is a contradiction, so ω must have an even number of constants. □

Using this theorem, we can rule out a large class of quadratic equations before even using the constraint solver on them: if a quadratic equation has an odd number of constant letters, then it is not solvable.

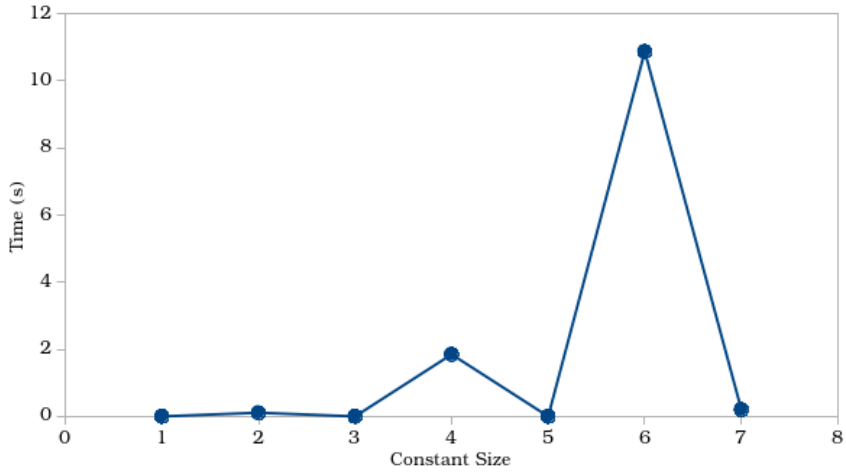


Figure 11: The time taken for the solver, using the optimization described in Theorem 6.4, to determine solvability of equations with 2 variables, a varying constant size between 1 and 7, and an alphabet size of 3, on completely randomly generated equations. The solver, due to the optimization, is extremely fast on all the equations with an odd number of constants.

Figure 11 demonstrates the pattern of runtime on randomly generated equations of different sizes. This data was collected in an experiment that used randomly generated quadratic equations with an odd number of variables and a varying constant size. Hence when the constant size is odd, there is an odd number of constants in the equation, and so runtime is negligible. When the constant length is even, however, the solver takes just as long as it normally would have without the optimization. So in the particular case of quadratic equations with an odd number of constants, this is a massive improvement on the unaltered constraint solver.

7 Usage and Documentation

This project consists of three parts: the written portion, a GAP package, and a Python package. The GAP package is located in the directory `GAP` and contains a fully functional GAP package which can be installed by simply being copied into the `pkg` directory of a GAP installation. Then all of the global functions of the package are accessible once the package is loaded in a GAP session, using the command

```
gap> LoadPackage("dshp");
```

This package contains the algorithm for finding solutions to linear equations, unit tests, and a script for running timing experiments. The source code is located in the directory `gap`, the unit tests in `tst/generic.tst`, and the experiment script in the file `benchmarking.g`. The algorithm can be run in GAP by importing the package, defining a free group, defining the constants of the equation in the order they appear, and passing them to the function `LinearEquation`. The below example represents the equation $aXbYab = \epsilon$.

```
gap> LoadPackage("dshp");;
gap> f := FreeGroup("a", "b");;
gap> consts := [f.1, f.2, f.1 * f.2];;
gap> LinearEquationSolutions(consts);;
```

An experiment can be run by reading the script file and then running the function inside it with the desired parameters. The parameters are the name of the attribute to vary, the minimum value of this attribute, the maximum value, the increment by which the attribute value is increased, the value of the first invariant, and the value of the second invariant. The three possible attributes to vary are `constSize` (the size of the constants), `alphabetSize` (the size of the basis of the free group), and `noVars` (the number of variables). They are referred to in this order, so if the variable is the alphabet size, then constant size is the first invariant and the number of variables is the second. For alphabet size and constant size, the values are incremented linearly, but the number of variables is incremented exponentially. The output is saved in a file named after the variable name, its minimum and maximum values, and the values of the invariants, all separated by underscores. In the below example, the algorithm is run with alphabet sizes 1, 2, 3, and 4, each with a constant size of 5 and 3 variables. The output would be saved in a file called `alphabetSize_1_4_5_3.csv`.

```
gap> LoadPackage("dshp");;
gap> Read("pkg/dshp/benchmarking.g");;
gap> BenchmarkLinearEquationSolution("alphabetSize", 1, 4, 1, 5, 3);;
```

To run the GAP tests, load the package in your GAP session and run the function `Test`.

```
gap> LoadPackage("dshp");;
gap> Test("pkg/dshp/tst/generic.tst");
```

The Python package is located in the directory called `constraints` and contains the constraint solver package, used for finding solutions to any type of equation. The source code is in the directory `solver`, tests in the file `tests.py`, and a script for running in experiments in `benchmarking.py`. All of the above should be run in a virtual environment which can be installed, activated, and set up by executing the following commands in the `constraints` directory.

```
$ python3 -m venv ./venv
$ . venv/bin/activate
(venv) $ pip install -r requirements.txt
```

The solver, the tests, and the experiments can all be run from within the activated virtual environment. The solver can take five arguments, all of which can be seen, with descriptions, by running the following command.

```
(venv) $ python -m solver -h
```

The equation to be solved is passed to the solver using the `--equation` flag followed by the equation as a string, where variables are represented as upper case letters from the Latin alphabet, constants are represented by lower case letters from the Latin alphabet, and an inverted constant or variable is denoted by a subsequent asterisk. The bound on the size of the variable can be set using the `--bound` flag followed by one of the following: linear, orientable, or non-orientable. If the bound is not set, it is automatically deduced from the form of the equation. Using the flag `--solvability` causes the solver to end the search as soon as the first solution is found. In the following example, the solver will attempt to find all solutions to the equation $aXbYab$ and the bound on variable size will be automatically deduced as the linear bound.

```
(venv) $ python -m solver --equation aXbYab
```

Tests can be run with the command `python tests.py` in the activated virtual environment. The experiment script can be run in the same way, but can take many arguments. Run

```
(venv) $ python benchmarking.py -h$
```

to see all of the execution options. Below is an example of the script being run with a set of typical arguments. In this example, the output file will be saved in the current directory, the attribute to vary is the number of variables (with the only other possible option being the constant size, `constSize`), and it varies in linear increments of 1 between 1 and 3. The experiment for each value will be repeated with a randomly generated equation 50 times, and the average time taken across all fifty repetitions written to the output file. The constant size, given by the invariant flag, is 3. The `--solvable` flag indicates that the randomly generated equations will all be solvable, and the lack of the `--solvability` flag means that the solver will attempt to find all possible solutions, not just one.

```
(venv) $ python benchmarking.py --variable noVars --equation quadratic
            --bound linear --savefile ./ --min 1 --max 3
            --intervaltype linear --interval 1 --solvable
            --invariant 3 --repetitions 50
```

Run the unit tests by running the file `tests.py` whilst the virtual environment is activated.

```
(venv) $ python tests.py
```

The virtual environment can be deactivated with the command `deactivate`.

The directory `data` contains all the data gathered during the course of this project that has been used in this paper. The directory `graphs` contains the corresponding graphs.

8 Conclusion

This paper established the mathematical background of equations over freely presented groups and explored methods of solving linear and quadratic equations. In it, I described and

evaluated an algorithm for deterministically finding a set of solutions to a linear equation. I also modelled solutions to equations as a constraint satisfaction problem, proved that the constraints are sufficient, and evaluated the efficiency of the constraint solver, both against the linear algorithm and against itself under various optimizations.

While the linear algorithm is faster than the constraint solver, and produces precisely the most desirable solutions to a linear equation, it is limited to only linear equations, which are always solvable and therefore, to a certain extent, uninteresting. The constraint solver, however, is applicable to any type of equation, not just linear, though it is only guaranteed to find a solution if there is an upper bound on its size. Quadratic equations do have such a bound, though it is unreasonably large for any practical use with a constraint solver. This issue can be offset using practical optimizations such as incremental bounds, which will find the smallest equations first, or theory-based optimizations that answer the solvability question by observing necessary conditions for certain types of equation to be solvable. For instance, in this paper I proved that a quadratic equation must have an even number of constants to be solvable, and also that a word must have an even number of non-identity elements to reduce to the identity. Proving such conditions and implementing them either as conditions to be checked before calling a constraint solver, or as further constraints on a potential solution help reduce the average runtime.

Given more time I would have explored these optimizations further, as well as addressed standard form quadratic equations, which have a much smaller bound on solution size. As is, however, this paper is already accompanied by a fully functioning GAP package for solving linear equations and a Python package, using Google OR-Tools, which is capable of finding a solution to any equation with a bounded solution.

9 Acknowledgements

Special thanks to Colva Roney-Dougal for assistance with mathematical writing and proofs, to Markus Pfeiffer with assistance in writing GAP code and formulating the algorithm for solving linear equations, and to Christopher Jefferson for formulating solutions to equations as a constraint satisfaction problem and for help with the proofs concerning the constraints.

10 References

[1] Bumagin, I., Kharlampovich, O., and Miasnikov, A. Isomorphism problem for finitely generated fully residually free groups. *arXiv Mathematics e-prints* (Feb 2005), math/0502496.

[2] Comerford, L. P. J., and Edmunds, C. C. Quadratic equations over free groups and free products. *Journal of Algebra 68* (1981), 276–297.

[3] Cook, M. The ping-pong lemma. https://math.la.asu.edu/~paupert/CookPingPongLemma.pdf, 2016. Arizona State University.

[4] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.10.1*, 2019.

[5] Google. Google or-tools. https://developers.google.com/optimization/, 2019.

[6] Johnson, D. L. *Presentations of Groups*, 2 ed. London Mathematical Society Student Texts. Cambridge University Press, 1997.

[7] Kharlampovich, O., Lysenok, I. G., Myasnikov, A. G., and Touikan, N. W. M. The solvability problem for quadratic equations over free groups is np-complete. *Theory of Computer Systems 47* (2010), 250–258.

[8] Lyndon, R. C. Equations in free groups. *Transactions of the American Mathematical Society 96*, 3 (1960), 445–457.

[9] Lysenok, I. G., and Myasnikov, A. G. Equations in free groups. *Proceedings of the Steklov Institute of Mathematics 274* (2011), 136–173.

[10] Robinson, D. *A Course in the Theory of Groups*. Springer-Verlag, 1981.

[11] Rossi, F., van Beek, P., and Walsh, T. *Handbook of Constraint Programming*. Elsevier Science Inc., 2006.

[12] Rotman, J. *An Introduction to the Theory of Groups*. Graduate Texts in Mathematics. Springer New York, 1999.

A Appendix

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☑ **Undergraduate Project**

Title of project

Solving Equations over Free Groups

Name of researcher(s)

DAPHNE BOGOSIAN

Name of supervisor (for student research)

COLVA RONEY-DOUGAL, MARKUS PFEIFFER

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☑ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

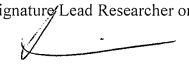
Print Name

DAPHNE BOGOSIAN

Date

25/09/2018

Signature Lead Researcher or Supervisor



Print Name

Markus Pfeiffer

Date

25/09/2018

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.