# A Lean tactic to solve the word problem in One Relator Groups

Christopher Hughes

April 11, 2021

CID 01360659
Supervised by Kevin Buzzard

# Contents

## 1 Introduction

This paper describes two methods for automating proofs of equalities in groups, given a number of equalities as hypotheses. Section 3 describes Magnus' method [MS73], which decides the word problem in one-relator groups. This method was first described by Wilhelm Magnus in 1932 (proper citation). Given a word in a free group, the "relator", it will decide whether any other word is in the normal subgroup containing this word. This is equivalent to deciding first order formulas in the language of groups of the form $t_1 = t_2 \implies t_3 = t_4$, where the $t_i$ are all terms in the language of groups. I am not aware of any other implementation of this method on a computer.

Section 5 describes an alternative method for solving the word problem, which we call the "Graph Search Method". This method is capable of using more than one equality as a hypothesis, with the caveat that it may not terminate if the equality it is asked to prove is false. It is a known result that the word problem for finitely presented groups is undecidable [Col86], however it is semidecidable meaning that there is an algorithm that will terminate with a proof whenever an equality is true, but will not necessarily terminate when an equality does not follow from the hypotheses.

Both methods are implemented as tactics in the Lean proof assistant, meaning that they will produce a formal proof of the result capable of being checked by the Lean proof assistant.

## 2 Introduction to Lean

Lean [dMKA+18] is a proof assistant; it is a language capable of expressing mathematical propositions and definitions, and also a language for writing formal proofs of these propositions. A formal proof in Lean is checked for correctness by Lean's kernel. A formal proof checked by Lean's kernel provides an extremely high level of confidence in a checked proof, at the expense of requiring a lot of time for a user to write a proof in a completely formal language.

As well as being a proof assistant, Lean is also a programming language. The definitions that are written in Lean are also executable programs, and Lean can be used to prove correctness of programs written in Lean as well as pure mathematical proofs. Lean can be used as a programming language to write Lean tactics, described in the next section.

There is a large library of formal mathematics in Lean, called mathlib [mC20]. As of January 2021 this contains 470000 lines of code.

### 2.1 Terms and Types

Lean is based on type theory; there are terms and types, and every term has a type. In Lean, the syntax `x : X` indicates that `x` has type `X`. Lean is based on type theory as opposed to set theory, so whilst on paper it might be more common to write $x \in X$, saying $x$ is an element of the set $X$, in Lean you would usually write `x : X` in Lean instead. So for any syntactically correct expression we write down in Lean, that term has a type, and we use the colon `:` to indicate the type of a term. `Type` itself is also a type, so for example the real numbers are a type, so `ℝ : Type`, and if `x` is a real number then `x : ℝ` or `x + 1 : ℝ`. Another very important type is the type of propositions `Prop`. `Prop` has type `Type`.

In Lean it is possible to make new types out of types. For example if `X : Type` and `Y : Type`, then `X × Y : Type`; `X × Y` is the cartesian product of the Types `X` and `Y`. Importantly you can also make the type of functions from `X` to `Y` using the arrow symbol, `X → Y : Type`. If `f : X → Y` and `x : Y`, then `f x : Y`, similar to how if $f : X \to Y$, then $f(x) \in Y$ in more conventional notation.

To define a binary function from a type `X` to itself in Lean, it is possible to give it the type `X × X → X`. However, this is not the most common way of defining binary functions, it is more common to use the canonically isomorphic type `X → (X → X)`. This is called the curried form of the function. This type is usually written without brackets as `X → X → X`, and this will be parsed by Lean as the type `X → (X → X)`, brackets are automatically inserted to the right. Functions of higher arity can be defined similarly.

We can also make the type of dependent functions between sets, where the type of the output depends on the input. These are known as Pi types, and correspond to indexed products in more familiar language. If `X : Type` and `Y : X → Type`, then we can write `Π i : X, Y i : Type`, and then if `i : X`, and `f : Π i: X, Y x` then `f i : Y i`. This corresponds to if $X$ is a set, and $(Y_i)_{i \in X}$ is a family of sets indexed by $X$, then $\Pi_{i \in X}, Y_i$ is the product of the $Y_i$, and if $f \in \Pi_{i \in X}, Y_i$, then you might write $f_i \in Y_i$.

Finally we will introduce the lambda notation used to define functions. This is similar to the $\mapsto$ symbol in conventional maths. You can write `λ x : ℝ, x + 1`, to define a function `ℝ → ℝ`, which corresponds to the function $x \mapsto x + 1$. To define a function in multiple variables, you can put two variables after the lambda, for example `λ x y : ℝ, x + y` define a binary function with type `ℝ → ℝ → ℝ`.

### 2.1.1 Propositions as Types

We now give some examples of propositions and theorems.

```
∀ p q : Prop, p ∧ q → q ∧ p
```

The above proposition says that the "and" connective, $\wedge$, is symmetric. Note that we use the same arrow as the function arrow, $\to$, for implication.

Below is a proof of the above proposition.

```
λ (p q : Prop) (h : p ∧ q), and.intro (and.right h) (and.left h)
```

Note that we use the lambda notation that we used to define functions. This is because Lean makes use of the Curry-Howard correspondence. The proof of this theorem can in fact be regarded as a function. The function takes two propositions `p` and `q`, and a proof of `p ∧ q`, and returns a proof of `q ∧ p`. The statement of the theorem is the type of the proof.

Now consider the following definition of the function `X × Y → Y × X` that swap the two elements. We want to define a function that works for any two types `X` and `Y`, not just defining it for a particular choice of types `X` and `Y`. Therefore, the function should take a pair of types as an input as well. The full type of this function is therefore

```
Π X Y : Type, X × Y → Y × X
```

Now we can define a function of this type.

```
λ (X Y : Type) (h : X × Y), prod.mk (prod.snd h) (prod.fst h)
```

Here, `prod.mk` is the natural function of type `Y → X → Y × X`. `prod.snd` is the natural map `X × Y → Y`, and `prod.fst` is the natural map `X × Y → X`.

This function definition looks very similar to the proof of the proposition `∀ p q : Prop , p ∧ q → q ∧ p`, and the type of the function looks very similar to the proposition itself. This is an example of the Curry Howard correspondence. `∀` for propositions corresponds to `Π` for types, logical and, ∧, corresponds to cartesian product, ×, and `prod.mk`, `prod.fst` and `prod.snd` correspond to `and.intro`, `and.left` and `and.right` respectively.

This way the same syntax that can be used to define terms of types such as `ℝ → ℝ` can also be used to write proofs of propositions, although proofs of propositions are usually far more complex than terms of other types.

## 2.2 Tactics

Now consider the following theorem called `thm` and its proof.

```
theorem thm (a b : ℤ) : a * (1 + b) - a = a * b :=
(mul_add a 1 b).symm ▷ (mul_one a).symm ▷ add_sub_cancel' a (a * b)
```

The proof of this theorem required explicitly invoking three elementary about commutative rings, the distibutivity law, the right multiplicative identity, and `add_sub_cancel'`, which says that for any $x$ and $y$, $x + y - x = y$. Proving more complicated equalities of ring expressions can often require a huge number of explicit applications of elementary lemmas about rings. This is where Lean's tactic framework becomes useful. A tactic is a program that helps to generate Lean proofs. There is a tactic for proving equalities of ring expresions called `ring`. Instead of the above proof, a Lean user could simply write

```
theorem thm (a b : ℤ) : a * (1 + b) - a = a * b :=
begin
  ring
```

```
end
```

The `begin` and `end` indicates that the user wants to use the tactic framework to generate a proof. Users can use more complicated combinations of tactics to prove a theorem. When using tactics to write a Lean proof, a tactic state is displayed to a user. This tactic state can be manipulated using tactics. An example of a tactic state is displayed below.

**Example 2.0.1.** The code below is a statement of the theorem that if $G$ is a group, and $a, b \in G$ are such that $ab = b^2a$, then for any natural number $n$, $a^nba^{-n} = b^{2^n}$. In between `begin` and `end` the user can write some tactics.

```
example {G : Type} [group G] (a b : G) (h : a * b = b ^ 2 * a) (n : ℕ
 ) :
  a ^ n * b * a ^ -↑n = b ^ 2 ^ n :=
begin

end
```

If the user types the above code then the following tactic state is displayed to the user.

```
1 goal
G : Type,
_inst_1 : group G,
a b : G,
h : a * b = b ^ 2 * a,
n : ℕ
⊢ a ^ n * b * a ^ -↑n = b ^ 2 ^ n
```

The turnstile ⊢ indicates the current goal, the proposition the user is currently trying to prove. The preceding lines indicate the hypotheses and variables. `G : Type` indicates that `G` is a type. `_inst_1` is a group structure on the Type `G`. A term of type `group G`, is a tuple containing a binary operation on `G`, an identity, and an inverse function, as well as proofs that these satisfy the group axioms. `a` and `b` are elements of `G`, `h` is the hypothesis that `a * b = b ^ 2 * a`, and `n` is a natural number.

If the user writes a tactic, then the tactic state will change. If the user wanted to prove this by induction on `n` they could write

```
induction n with k ih,
```

The tactic state after this line will now be

```
2 goals
case nat.zero
G : Type,
_inst_1 : group G,
a b : G,
h : a * b = b ^ 2 * a
⊢ a ^ 0 * b * a ^ -↑0 = b ^ 2 ^ 0
```

```
case nat.succ
G : Type,
_inst_1 : group G,
a b : G,
h : a * b = b ^ 2 * a,
k : ℕ,
ih : a ^ k * b * a ^ -↑k = b ^ 2 ^ k
⊢ a ^ k.succ * b * a ^ -↑(k.succ) = b ^ 2 ^ k.succ
```

The tactic split the goal into two cases, in the first case, labelled `case nat.zero`, `n` is zero, and in the second case, labelled `case nat.succ n` is the successor of `k`. In the second case, an induction hypothesis `ih` has been added.

The user could then use the `rw` tactic to solve the first goal, for example typing `rw pow_zero` changes `a ^ 0` to `1` and the new goal after this line will be

```
⊢ 1 * b * a ^ -↑0 = b ^ 2 ^ 0
```

`pow_zero` is the theorem that says if $a$ is an element of any monoid, then $a^0 = 1$.

To solve the first goal the `simp` tactic can be used. The `simp` tactic applies as many simplification rules as possible. These rules are lemmas in the library marked by the library authors as simplification lemmas that are proofs of an equality where the right hand side is judged to be simpler than the left hand side. For example there is a lemma simplifying `1 * a` to `a`, or `b ^ 1` to `b`.

To solve the second goal, the `group_rel` tactic described later in this document can be used, and it will finish off the proof.

## 3 Magnus' Method

This section starts by defining and stating a few important definitions and results about both HNN extensions and free groups.

The tactic for solving this method is split into three stages. Firstly, the Lean tactic state is converted into a word problem on an efficient representation of the free group. The tactic must compute the relator and the target word based on the tactic state. This process is very similar for both tactics described in this paper and is described in Section 6. Then the problem is solved using an efficient representation of the free group, and a proof certificate is computed. We call the function that performs this part *Solve*. The proof certificate is not a Lean proof, but does contain enough information about the solution to be able to be used to compute a formal Lean proof. This certificate is described in Section 3.3.

## 3.1 Free Group

The free group is implemented in Lean as the set of reduced words. An element of the free group over a type $S$ of letters is a list of pairs $S \times \mathbb{Z}$, the letter and the exponent. A list of the exponent part of every element of the list is non zero, and no two adjacent elements of the list have the same letter. The free group is the set of reduced lists.

Multiplication of elements of the free group is implemented by appending the lists whilst replacing any adjacent occurrences of $(s, m)$ and $(s, n)$ with $(s, m{+}n)$, and removing any occurrence of $(s, 0)$. Inversion is given by reversing the list and negating the exponent part of every pair. The identity is given by the empty list.

**Definition 3.1** (Length)**.** The length of a word $w$ in the free group is the sum of the absolute values of the exponent parts of each element of the corresponding reduced list.

**Definition 3.2** (Cyclically Reduced)**.** A word $w$ in the free group is cyclically reduced if it cannot be made shorter by conjugating.

We state the following theorem *Freiheitsatz*. This theorem is an important part of the correctness of Magnus' method. The proof is omitted.

**Theorem 3.3** (Freiheitsatz)**.** Suppose $F(S)$ is the free group over a set $S$ and $r$ is a a cyclically reduced word, and $T \subset S$ is a set of letters such that $r$ cannot be written using any letters in $T$. Then $T$ is a basis for a free subgroup of $F(S)$. [MS73]

## 3.2 HNN Extensions

Magnus' method makes use of isomorphisms between one-relator groups and HNN extensions. In this section we define an HNN extension of a group relative to an an isomorphism between two subgroups.

**Definition 3.4** (HNN Extension)**.** Given a group $G$, subgroups $A$ and $B$ of $G$, and an isomorphism $\phi : A \to B$, we can define the *HNN extension* relative to $\phi$ of $G$. Let $\langle t \rangle$ be an infinite cyclic group generated by $t$. The HNN extension is the coproduct of $G$ and $\langle t \rangle$ quotiented by the normal closure of the set $\{tat^{-1}\phi(a^{-1}) | a \in A\}$

The map from $G$ into the HNN extension is injective. TO DO citation.

**Definition 3.5** (HNN normal form)**.** Let $w = g_0 t^{k_1} g_1 t^{k_2} g_2 \cdots t^{k_n} g_n \in G * \langle t \rangle$. Then $w$ is in *HNN normal form* if for every $i$, $k_i \neq 0$, $k_i > 0$ implies $g_i \notin A$ and $k_i < 0$ implies $g_i \notin B$.

Note that the HNN normal form is not unique; two words $w, v \in G * \langle t \rangle$ that are equal after mapping into the HNN extension and both in normal form might not be equal as elements of $G * \langle t \rangle$. However, if $w \in G * \langle t \rangle$ maps to 1 in the HNN extension then the following lemma tells us that the unique HNN normal form for $w$ is 1.

**Theorem 3.6** (Britton's Lemma)**.** Let $w \in G * \langle t \rangle$. If $w$ is in HNN normal form and $w$ contains a $t$, then $w \neq 1$ [Mil68]

**Corollary 3.6.1.** If a word $w$ meets the conditions of Britton's Lemma, then $w$ cannot be written as a $t$ free word.

*Proof.* Suppose $w = g$ with $g \in G$; then $g^{-1}w$ also meets the conditions in Theorem 3.6, and therefore $gw^{-1} \neq 1$, contradicting $w = g$. □

Given a word $w \in G * \langle t \rangle$, the HNN normalization process replaces any occurrences of $ta$ with $\phi(a)t$ when $a \in A$, and any occurrence of $t^{-1}b$ with $\phi^{-1}(b)t^{-1}$ when $b \in B$. Applying this rewriting procedure will always produce a word $w'$ in HNN normal form and such that $w$ and $w'$ are equal after quotienting by the defining relations of the HNN extension, $\{tat^{-1}\phi(a^{-1}) | a \in A\}$.

The HNN normalization process describes an algorithm for deciding whether two words $w, v \in G * \langle t \rangle$ are equal after mapping into the HNN extension, by applying the normalization procedure to $wv^{-1}$. In order to compute this algorithm, it is also necessary to have an algorithm for checking equality of elements in $G$, for checking whether an element of $G$ is in either of the subgroups $A$ or $B$, and for computing $\phi$.

## 3.3 The Proof Certificate

An element of a group $G$ is equal to 1 in the quotient by the normal closure of a relator $r$ if and only if it can be written as a product of conjugates of $r$ and $r^{-1}$. More precisely, there is a group homomorphism $Eval : F(G) \to G$ from the free group over $G$ into $G$ that sends a basis element of $F(G)$, $g \in G$, to $grg^{-1} \in G$. The image of this map is exactly the kernel of the quotient map. Therefore an element $p$ of $F(G)$ such that $\text{Eval}(p) = w$ can be seen as a witness that $w$ is in the kernel of the quotient map.

**Definition 3.7.** (Eval) $\text{Eval}(r)$ is a map $F(G) \to G$, sending a basis element $g \in G$ to $grg^{-1}$.

The certificate is a pair of a normalised word $w'$ and $p \in F(G)$ such that $w = \text{Eval}(p)w'$.

**Definition 3.8** (P functor)**.** For any $g \in G$ define an automorphism $\text{MulFree}(g)$ of $F(G)$ by sending a basis element $h \in G$ to $gh$. This defines a left action of $G$ on $F(G)$. Define the group $P(G)$ to be

$$P(G) := F(G) \rtimes_{MulFree} G \tag{1}$$

This group has multiplication given by $(a, b)(a', b') = (a\text{MulFree}(b)(a'), bb')$

**Definition 3.8.1** (lhs and rhs)**.** Define two group homomorphisms from $P$ into $G$: let rhs be the obvious map sending $(a, b)$ to $b$ and let lhs be the map sending $(a, b)$ to $\text{Eval}(a)b$. Since $\text{Eval}(a)$ is in the kernel of the quotient map, for any $p \in P(G)$, $\text{lhs}(p)$ and $\text{rhs}(p)$ are equal in the quotient by $r$. Therefore an element $p$ of $P(G)$ can be regarded as a certificate of the congruence $\text{lhs}(p) \equiv \text{rhs}(p) \bmod r$.

One way of expressing this is that the group $G/r$ is the coequalizer of the two surjective maps lhs and rhs.

$$\begin{tikzcd} P(G) \arrow[r, shift left, "rhs"] \arrow[r, shift right, "lhs"'] & G \arrow[r] & G/r \end{tikzcd}$$

Because both lhs and rhs are group homomorphisms, if $p \in P(G)$ is a certificate of the congruence $a \equiv b \bmod r$ and $q$ is a certificate of the congruence $c \equiv d \bmod r$, then $pq$ is a certificate of the congruence $ac \equiv bd \bmod r$. Similarly, $p^{-1}$ is a certificate of the congruence $a^{-1} \equiv b^{-1} \bmod r$.

**Definition 3.8.2.** ($P$ is functorial). Given a homomorphism $f : G \to H$, functoriality of the free group gives a natural map $F(f) : F(G) \to F(H)$. Define the map $P(f) : P(G) \to P(H)$ to send $(p, b) \in P(G)$ to $(F(f)(p), f(b)) \in P(H)$. Given a certificate of the congruence $a \equiv b \bmod r$, this map returns a certificate of the congruence $f(a) \equiv f(b) \bmod f(r)$.

**Definition 3.8.3.** (Trans) Given $p, q \in P(G)$ such that $p$ is a certificate of the congruence $a = b \bmod r$ and $q$ is a certificate of the congruence $b = c \bmod r$, it is possible to define Trans$(p, q)$ such that Trans$(p, q)$ is a certificate of the congruence $a = c \bmod r$. If $p = (p_1, p_2)$ and $q = (q_1, q_2)$, then Trans$(p, q) = (p_1 q_1, q_2)$.

**Definition 3.8.4.** (Refl) Given $a \in G$, $(1, a)$ is a certificate of the congruence $a = a \bmod r$. Call this Refl$(a)$.

It is also possible to define *Symm* such that lhs(Symm$(p)$) = rhs$(p)$ and vice versa, but this is not used in the algorithm.

**Definition 3.8.5.** (ChangeRel) Given a certificate $p$ of the congruence $a \equiv b \bmod r$, it is possible to make a certificate of the congruence $a \equiv b \bmod grg^{-1}$ for any $g \in G$. For any $g \in G$, let $\phi(g) : F(G) \to F(G)$ be the map sending $h \in G$ to $hg$. Then ChangeRel$(g, (p_1, p_2))$ is defined to be $(\phi(g)(p_1), p_2)$ for $g \in G$ and $(p_1, p_2) \in P(G)$.

### 3.3.1 Performance

The representation of $F(F(S))$ can be improved. The automorphism $MulFree$ multiplies every letter in a word by another word. The consequence is that many of the words in $F(S)$ making up an element of $F(F(S))$ are very similar. Take the word $[w][v] \in F(F(S))$, where $w, v, u \in F(S)$, and consider the element MulFree$(u)[w][v] = [uw][uv]$. It is more efficient to only store the difference between adjacent letters, so the element $[w][v]$ would be represented as the sequence $w, w^{-1}v$, and the element $[uw][uv]$ would be represented as the sequence $uw, v^{-1}$. If $u$ is a long word, which it often is, then this representation will usually be shorter. The longer the word $x$ such that the algorithm is attempting to prove $\overline{w} = 1$, the longer a typical length of $u$ is, so in fact the standard representation tends to give certificates more of length or less quadratic in the length of $x$. This also has the advantage that to compute $MulFree$ only the first term in the sequence needs to be changed.

An improved representation of the group $F(F(S))$ for a set $F(S)$, is to again represent an element as a finite sequence of elements of $F(S) \times \mathbb{N}_{\geq 1} \times \{-1, 1\}$.

Define the set $X$ to be the set of finite sequences of elements of $F(S) \times \mathbb{N}_{\geq 1} \times \{-1, 1\}$, with the property that there are no adjacent pairs of the form $(g_i, n_i, s_i)(1, n_{i+1}, -s_i)$. A pair of this form will evaluate to $(g_i r^{s_i})^{n_i} r^{-s_i n_{i+1}} \dots g_i^{-n_i}$, so there is a natural cancellation that can be made, whilst preserving the same evaluation. This reduction corresponds to cancellation of inverses in the usual representation of the free group. Later, we will define a Reduce function on sequences.

Given a pair $(g, n, b) \in F(S) \times \mathbb{N}_{\geq 1} \times \{-1, 1\}$, define an element $f(g, n, b) \in P(F(S))$.

$$\begin{cases} f(g, n, 1) := ([g], g)^n \\ f(g, n, -1) := ((1, g^{-1})([g]^{-1}, 1))^{-n} \end{cases} \tag{2}$$

Therefore given a sequence $(g_1, n_1, s_1), (g_2, n_2, s_2) \dots (g_k, n_k, s_k)$, we can define the corresponding element of $F(F(S))$ to be

$$\left( \prod_{i=1}^{k} f(g_i, n_i, s_i) \right) \left( \prod_{i=1}^{k} (1, g_i)^{-n_i} \right) \tag{3}$$

The evaluation map into $F(S)$ can then be define as

$$\text{Eval}((g_1, n_1, s_1), (g_2, n_2, s_2) \dots (g_k, n_k, s_k)) := \left( \prod_{i=1}^{k} (g_i r_i^s)^{n_i} \right) \left( \prod_{i=1}^{k} g_i^{n_i} \right) \tag{4}$$

One way of viewing this representation is that it stores a way of representing as a sequence of applications of $h(g_i, s_i)$, to a word $w$, where each $n_i$ represents how many times the map $h(g_i, s_i)$ should be applied.

$$h(g_i, s_i)(w) = g_i r^{s_i} w g_i^{-1} \tag{5}$$

The map MulFree can be defined on this representation much more efficiently, since only the first element in the list need be changed.

$$\begin{cases} \text{MulFree}(w)(1) := 1 \\ \text{MulFree}(w)((g_1, 1, s_1), \dots, (g_k, n_k, s_k)) := (wg_1, 1, s_1), \dots, (g_k, n_k, s_k) \\ \text{MulFree}(w)((g_1, n_1, s_1), \dots, (g_k, n_k, s_k)) := \\ \qquad (wg_1, 1, s_1), (g_1, n_1 - 1, s_1), \dots, (g_k, n_k, s_k) & \text{if } n_1 > 1 \end{cases} \tag{6}$$

Multiplication can also now be defined on this representation.

We can now define a reduction map to eliminate pairs of the form $(g_i, n_i, s_i)(1, n_{i+1}, -s_i)$.

**Definition 3.9** (Reduce)**.** Reduce eliminates pairs of this form in a sequence with some rewriting rules. In this definition $S$ is some finite sequence of triples.

$$\begin{cases} (g_i, 1, s_i), (1, 1, -s_i), S \to \mathrm{MulFree}(g_i)(S) & \\ (g_i, n_i, s_i), (1, 1, -s_i), S \to (g_i, n_i - 1, s_i), \mathrm{MulFree}(g_i)(S) & \text{if } n_i > 1 \\ (g_i, 1, s_i), (1, n_{i+1}, -s_i) \to (g_i, 1, -s_i), (1, n_{i+1} - 1, -s_i) & \text{if } n_{i+1} > 1 \\ (g_i, n_i, s_i), (1, n_{i+1}, -s_i) \to (g_i, n_i - 1, s_i), (g_i, 1, -s_i), (1, n_{i+1}, -s_i) & \text{if } n_i, n_{i+1} > 1 \end{cases} \tag{7}$$

Multiplication can be defined on this representation. For a finite sequence

$$L := (g_1, n_1, s_1), \dots, (g_k, n_k, s_k) \tag{8}$$

define

$$p(L) := \prod_{i=1}^{k} g_i \tag{9}$$

For a pair of sequence $S$ and $T$ use the notation $S, T$ to append the sequences. Then the product of two sequences $S$ and $T$, $S \cdot T$ is defined to be

$$S \cdot T = \mathrm{Reduce}(S, \mathrm{MulFree}(p(S), T)) \tag{10}$$

It may be sensible to store $p(S)$ as part of the data of a sequence $S$ so it does not need to be recomputed every time sequences are multiplied.

As an illustration of the efficiency of this representation of the free group, consider certificates of equalities of the form $\bar{(}wr)^n\overline{w^{-n}} = 1$ for large positive values of $n$, have a much shorter representation. In the efficient representation this certificate will be

$$(w, |n|, \mathrm{sgn}(n)) \tag{11}$$

so the sequence will be of length either 2 or 1. In the less efficient representation, this certificate will be

$$\prod_{i=1}^{n} [w^i] \tag{12}$$

This certificate will be a sequence of length $n$ even after the word is reduced, the overall data used will be quadratic in $n$, since the length of the letters increases with the size of $n$ as well.

## 3.4 Adding and Removing Subscripts

Given a letter $t$ in the free group over a set $S$, we can define a map into a semidirect product.

**Definition 3.10** (ChangeSubscript)**.** Define a homomorphism ChangeSubscript from $\mathbb{Z}$ to the automorphism group of $F(S \times \mathbb{Z})$. If $(x, n) \in S \times \mathbb{Z}$ is a basis element of the free group, then ChangeSubscript$(m)(x, n) = (x, m + n)$.

**Definition 3.11** (AddSubscripts)**.** There is a homomorphism AddSubscripts$(t)$ from $F(S)$ into $F(S \times \mathbb{Z}) \rtimes_{\text{ChangeSubscript}} \mathbb{Z}$ sending a basis element $s \in S$ to $(s, 0) \in F(S \times \mathbb{Z})$ when $s \neq t$ and sending $t$ to $(1, 1_{\mathbb{Z}}) \in F(S \times \mathbb{Z}) \rtimes \mathbb{Z}$. Loosely, this map replaces occurrences of $t^n a t^{-n}$ with $a_t$.

The map AddSubscripts is only used during the algorithm on words $w$ when the sum of the exponents of $t$ in $w$ is zero, meaning the result will always be of the form $(w', 0_{\mathbb{Z}})$.

**Definition 3.12** (RemoveSubscripts)**.** RemoveSubscripts sends a basis element of $F(S \times \mathbb{Z})$, $(s, n) \in S \times \mathbb{Z}$, to $t^n s t^{-n}$.

RemoveSubscripts is a group homomorphism and if $r$ is a word such that AddSubscripts$(r)$ is of the form $(r', 0_{\mathbb{Z}})$, then RemoveSubscripts$(r') = r$.

## 3.5 Overview of The Method

Given an element $w \in F(S)$ of a free group, a relation $r$ in the free group, and a subgroup of the free group generated by a set of letters $T$, we write $\overline{w}$ for the corresponding element in $F(S)/r$ and $\overline{T}$ for the image of the subgroup generated by $T$ in $F(S)/r$.

The algorithm decides whether $\overline{w} \in F(s)/r$ is in $\overline{T}$, and if it is, returns an element $w'$ such that $\overline{w'} = \overline{w}$ and $w' \in T$.

We describe the implementation of a function *Solve* whose arguments are a word $w$ in the free group $F(S)$, a relator $r \in F(S)$, and a subset $T$ of $S$. If there is a word $w' \in F(S)$ such that $w' \in T$ and $\overline{w} \in F(S)/r$ is equal to $\overline{W'}$, then it returns an element $p$ of $P(F(S))$ such that lhs$(p) = w$ and rhs$(p) = w'$. It terminates without returning anything if there is no such word.

Without loss of generality we can assume $r$ is cyclically reduced and conjugate $r$ if this is not the case. We can use ChangeRel, to make the correct proof certificate after conjugating $r$.

### 3.5.1 Case 1: All letters in r are in T

For this case it is helpful to consider the group $F(S)$ as the coproduct of the subgroup generated by the letters in $T$ and the subgroup generated by the rest of the letters: $F(S) \cong F(T) * F(S \backslash T)$.

Since every letter in $r$ is also in $T$ then $F(S)/r \cong F(T)/r * F(S \backslash T)$. An element of $F(S)$ can therefore be written in the form $w_0 v_0 w_1 v_1 \dots w_n v_n$, where $w_i \in F(T)$ and $v_i \in F(S \backslash T)$. The problem can be reduced to deciding whether an element of $w_i \in F(T)$

is equal to an element of the quotient. To decide the word problem whenever $\overline{w_i} = 1$, perform this substitution and then check whether the resulting word in $F(T) * F(S\backslash T)$ is in $T$.

### 3.5.2 Case 2: There is a letter in r that is not in T

**Base Case** The base case is the case where the relation $r$ is of the form $a^n$ with $n \in \mathbb{Z}$, and $a$ a letter in $S$. It is straightforward to decide the word problem in this group, since $F(S)/a^n$ is isomorphic to the binary coproduct of $F(S\backslash\{a\})$ and $\mathbb{Z}/n\mathbb{Z}$.

**Case 2a: Letter with exponent sum zero** Apply the map AddSubscripts($t$) (Definition 3.11) to $r$. Since the exponent sum of $t$ is equal to zero, AddSubscripts($t$)($r$) is of the form $(r', 0_{\mathbb{Z}})$. The length (Definition 3.1) of the relation $r' \in F(S \times \mathbb{Z})$ is less than the length of $r$. If $t \notin T$ and the exponent sum of $t$ in $w$ is not zero, then $\overline{w} \notin \overline{T}$. If $t \in T$, then $w$ can be written in the form $w't^n$ where $t$ has exponent sum zero in $w'$, and $w'$ is a word in $T$ if and only if $\overline{w} \in \overline{T}$.

A naive approach would be to apply AddSubscripts($t$) to $w$, and solve the word problem in $F(S \times \mathbb{Z})$ with respect to $r'$. However, the image of the normal closure of $r'$ under AddSubscripts($t$) restricted to $F(S \times \mathbb{Z})$ is not the normal closure of $r'$; it is the normal closure of the set of all relations of the form ChangeSubscript($n$)($r'$) for every $n$.

Pick $x \in S$ such that $x \neq t$, $x$ is a letter in $r$ and such that $t \in T$ implies $x \notin T$. If this is not possible, then apply the procedure in Section 3.5.1. We can assume that the first letter of $r$ is $x$, since otherwise $r$ can be conjugated until the first letter is $x$. Let $a$ and $b$ be respectively the smallest and greatest subscript of $x$ in $r'$. Let $S'$ be the set

$$S' := \{(i_1, i_2) \in S\backslash\{t\} \times \mathbb{Z} \mid i_1 \neq x \vee a \leq i_2 \leq b\} \tag{13}$$

Define two subsets of $S'$ by

$$A := S'\backslash\{x_b\} \tag{14}$$

$$B := S'\backslash\{x_a\} \tag{15}$$

Then there is an isomorphism $\phi$ between $A$ and $B$ given by $\phi :=$ ChangeSubscript(1). We claim the group $F(S)/r$ is isomorphic to the HNN extension of $F(S')/r'$ relative to $\phi$.

The homomorphism $\alpha$ from $F(S)/r$ to the HNN extension sends a letter $s \in S\backslash\{t\}$ to $s_0$ and the letter $t$ to the stable letter $t$ of the HNN extension. Since $ts_it^{-1} = s_{i+1}$ in the HNN extension for $s_i \in S'$, $r$ is sent to $r'$ by this map so that $\alpha$ is well defined on the quotient.

Now let $\beta$ send $s_i \in S'$ to $t^ist^{-i}$ and the stable letter $t$ to $t$. Again, $r'$ is sent to $r$ by $\beta$, and $\beta(ts_it^{-1}) = t^{i+1}st^{-(i+1)} = \beta(\phi(s_i))$ so $\beta$ preserves the defining relations of the HNN extension and it is well defined. It can be checked $\beta$ is a two sided inverse to $\alpha$, and thus $\alpha$ is an isomorphism.

We then apply the HNN normalization procedure, which will be described in detail in Section 3.6. We chose $x$ and $t$ such that either $x \notin T$ or $t \notin T$.

In either case, if $\overline{w}$ can be written as a word in $T$, then an HNN normal form of $w$ will be of the form $gt^n$ with $g \in F(S')/r'$. In the case $x \notin T$, then this is because any word in $F(S')$ not containing $x_i$ must be in $A \cap B$, there can be no occurrence of $tg$ with $g \notin A$ or $t^{-1}g$. If $t \notin T$, then it must be possible to write $w$ without $t$, so in fact it can be normalized to $g \in F(S')/r'$. We can check whether any words in $F(S')/r'$ are in the subgroups generated by $A$ or $B$ using Magnus' method again for the shorter relation $r'$, and rewrite these words using the letters in $A$ or $B$ when possible.

Once in the form $gt^n$ with $g \in F(S')$, it is enough to check that $\overline{\text{RemoveSubscripts}(g)}$ can be written as a word in $T$. If $t \in T$ then this amounts to solving the word problem for $r'$ and the set $T' := \{s_i \in S' | s \in T, i \in \mathbb{Z}\}$. If $t \notin T$, this amounts to checking that $n = 0$ and solving the word problem for $r'$ and the set $T' := \{s_0 \in S' | s \in T\}$.

**Case 2.b: No letter with exponent sum zero**

If there is no letter $t$ in $r$ with exponent sum zero, then choose $x$ and $t$ such that $x \neq t$ and such that if $t \notin T$ then $x \notin T$. Let $\alpha$ be the exponent sum of $t$ in $r$ and let $\beta$ be the exponent sum of $x$.

Then define the map $\psi$ on $F(S)$ by

$$\psi(s) = \begin{cases} t^{\beta} & \text{if } s = t \\ xt^{-\alpha} & \text{if } s = x \\ s & \text{otherwise} \end{cases} \tag{16}$$

The map $\psi$ descends to a map $\overline{\psi} : F(S)/r \to F(s)/\psi(r)$. The map $\psi$ is equal to $\psi_1 \circ \psi_2$, where $\psi_2$ and $\psi_1$ are defined as follows:

$$\psi_1(s) = \begin{cases} xt^{-\alpha} & \text{if } s = x \\ s & \text{otherwise.} \end{cases} \tag{17}$$

$$\psi_2(s) = \begin{cases} t^{\beta} & \text{if } s = t \\ s & \text{otherwise} \end{cases} \tag{18}$$

We have that $\overline{\psi_1} : F(S)/r \to F(r)/\psi_1(r)$ is an isomorphism, with inverse given by sending $x$ to $xt^{\alpha}$. Meanwhile, $\overline{\psi_2} : F(S)/r$ to $F(r)/\psi_2(r)$ is also injective. This is proven constructively in Theorem 3.24. Hence $\overline{\psi} : F(S)/r$ to $F(r)/\psi(r)$ is injective.

The image of the subgroup generated by $T$ under $\psi$ might not be the subgroup generated by a set of letters, but it is always contained in $T$. By the Freiheitsatz, if $\overline{\psi(w)}$ can be written as a word $w'$ using letters in $T$ then this solution is unique. Therefore, to check if $\overline{\psi(w)}$ is in the subgroup generated by $\overline{\psi(T)}$, one can first write it as a word in

$w' \in T$ if possible, and then check if $w'$ is in $\psi(T)$. The exponent sum of $t$ in $\psi(r)$ is 0, so the problem of checking if $\psi(w)$ can be written as a word in $T$ can be solved using the method described in Section 3.5.2 (Sort out this label).

If $t \in T$ then $\psi(T)$ is generated by $T' := T \backslash \{t\} \cup t^{\beta}$. By the Freiheitsatz, if $\psi(w)$ can be written as a word $w'$ using letters in $T$ then this solution is unique. Therefore, to check if $\psi(w)$ is in the subgroup generated by $T'$, one can first write it as a word in $w'$ in $T$ if possible, and then check that for every occurrence of $t^k$ in $w'$, $k$ is a multiple of $\alpha$.

If $t \notin T$, then $\psi(T) = T$.

## 3.6 HNN normalization

We first present a simplified version of the HNN normalization that does not compute the proof certificates, and then explain how to compute the certficates at the same time as normalization.

To compute the HNN normalized term, first compute the following isomorphism from $F(S)$ into the binary coproduct $F(S') * \langle t \rangle$, where $\langle t \rangle$ is an infinite cyclic group generated by $t$.

**Definition 3.13.** Define a map on a basis element $i$ as follows

$$\begin{cases} i_0 \in S' & i \neq t \\ t & i = t \end{cases} \tag{19}$$

It is important that $a \leq 0 \leq b$, to ensure that the image of this map is contained in $F(S' \times \langle t \rangle)$.

Then apply the HNN normalization procedure. For this particular HNN extension $\phi$ is *ChangeSubscript* (Definition 3.10). We work in the $F(S') * \langle t \rangle$, and apply the following rewriting rules.

For each occurrence of $tw$ where there is an $a \in A$ such that $\overline{a} = \overline{w}$ replace $tw$ with $\phi(a)t$.

For each occurrence of $t^{-1}w$ where there is an $b \in B$ such that $\overline{b} = \overline{w}$ replace $tw$ with $\phi^{-1}(b)t^{-1}$.

We can use *Solve* to check whether there is such $a$ and $b$ with these properties.

### 3.6.1 Computing Proof Certificates

To compute proof certificates a slight modification of the procedure described in Section 3.6 is used.

First define a modification of Definition 3.13, from $F(S)$ into the binary coproduct $P(F(S \times \mathbb{Z})) * \langle t \rangle$.

**Definition 3.14.** Define a map on the basis as follows

$$\begin{cases} \text{Refl}(i, t^0) \in F(S' \times \langle t \rangle) & i \in S \text{ and } i \neq t \\ t & i = t \end{cases} \tag{20}$$

There is also a map $Z$ from $P(F(S \times \mathbb{Z})) * \langle t \rangle$ into $P(F(S))$. This map is not computed as part of the algorithm, but is useful to define anyway.

**Definition 3.15.** The map $Z$ sends $t' \in \langle t \rangle$ to $\text{Refl}(t) \in P(F(S))$. It sends $p \in P(F(S \times \mathbb{Z}))$ to $P(\text{RemoveSubscripts})(p) \in P(F(S))$

The aim is to define a normalization process into that turns a word $w \in F(S)$ into word $n \in P(F(S \times \mathbb{Z})) * \langle t \rangle$ such that after applying *rhs*, the same word is returned as in the normalization process described in Section 3.6. We also want $\text{lhs}(Z(n))$ to be equal to $w$, so we end up with a certificate that $w$ is equal to some normalized word.

**Definition 3.16.** (conjP) Let $(p, a) \in P(F(S \times \mathbb{Z}))$ and $k \in \mathbb{Z}$. Define *ConjP* to map into $P(F(S \times \mathbb{Z}))$

$$\text{ConjP}(k, (p, a)) = (\text{MulFree}((t, 0)^k, p), \text{ChangeSubscript}(k, a)) \tag{21}$$

*conjP* has the property that $\text{lhs}(Z(\text{conjP}(k, p))) = t^k \text{lhs}(Z(p)) t^{-k}$, and similarly for *rhs*. Note that *conjP* maps into $P(F(S \times \mathbb{Z}))$ and not $P(F(S'))$, although *rhs* of every word computed will be in $F(S')$.

The procedure described in Section 3.6 replaced each occurrence of $wt^{-1}$ with $t^{-1}\text{ChangeSubscript}(-1)(a)$, where $a \in A$ was a word equal to $w \in F(S')$ in the quotient $F(S')/r'$.

To compute the certificates apply the following rewriting procedure: for each occurrence of $tp$ where $\overline{\text{rhs}(p)} = \overline{a}$ for some $a \in A$, and $q$ is a certificate of this equality, replace $tp$ with $\text{ConjP}(1, \text{Trans}(p, q))t$

Similarly, for each occurrence of $t^{-1}p$ where $\overline{\text{rhs}(p)} = \overline{b}$ for some $b \in B$ and $q$ is a certificate of this equality, replace $t^{-1}p$ with $\text{ConjP}(-1, \text{Trans}(p, q))t^{-1}$

### 3.6.2 Performance

The order in which the rewriting rules are applied can have a big effect on the performance of the algorithm.

**Example 3.16.1.** Suppose $r' = x_1 {x_0}^{-2}$ and $w = t^n x_1 t x_0^{-1}$, where $n > 0$. Then $S'$ is the set $\{x_0, x_1\}$, $A$ is the subgroup generated by $S' \backslash x_1$ and $B$ the subgroup generated by

$S'\backslash x_0$. Suppose we first make the substitution $tx_0^{-1}$ to $x_1^{-1}t$; then $w$ becomes $t^n x_1 x_1^{-1} t = t^{n+1}$. This is in HNN normal form.

Now consider trying the HNN normalization process from the left. For any $m \in \mathbb{Z}$, $x_1^m = x_0^{2m}$, so the HNN normalization proces will rewrite $tx_1^m to x_1^{2m} t$. Therefore $t^n x_1$ will be rewritten to $x_1^{2^n} t^n$. Hence $w$ gets rewritten to $x_1^{2^n} t^{n+1} x_0^{-1}$, which then will eventually be rewritten to $t^{n+1}$. The maximum length of $w$ during the normalization process was greater than $2^n$.

Applying one rewrite rule first might mean that another rewrite is unnecessary, or a call to *Solve* is given an easier problem.

**Example 3.16.2.** Consider the word $tw_0t^{-1}w_1$ with $w_0, w_1 \in F(S')$. In this situation it is best to start by attempting to prove $\overline{w_0} \in \overline{A}$. Applying the left hand rewrite first will put the word into HNN normal form straight away; it will not be necessary to check $\overline{w_1} \in \overline{B}$.

Rewriting starting on the right first might give a word such as $tw_0\phi^{-1}(b)t^{-1}$, where $\overline{b} = \overline{w_1}$. But since $\phi^{-1}(b) \in A$, checking whether $\overline{w_0\phi^{-1}(b)} \in \overline{A}$ is no easier than checking $\overline{w_0} \in \overline{A}$ has not become any easier. So in this example it is better to start rewriting on the left, and furthermore, if $\overline{w_0} \notin \overline{A}$ then it will not be possible to eliminate the $t$'s, so the algorithm can fail straight away without attempting more rewrites.

### 3.6.3 Other Potential Improvements

There are other potential improvements that could be made but it is not clear what effect, positive or negative, they would have on performance. These are discussed in this section.

A potential improvement is changing the definition of the set $S'$ in Section 3.5.2, to make the set as small as possible. First define the set $X$ of letters that meet the criteria for $x$ from Section 3.5.2.

$$X := \{x \in S\backslash\{t\} | x \text{ is contained in } r \text{ and } \ (x \notin T \text{ or } t \notin T)\} \tag{22}$$

For any $x \in X$, $a(x)$ and $b(x)$ would be defined in a similar way to $a$ and $b$ in Section 3.5.2, as the maximum and minimum subscripts of $x$ in the word AddSubscript$(t)(r)$. Then

$$S' := \{(x, n) \in S\backslash\{t\} \times \mathbb{Z} \mid x \notin X \text{ or } a_x \leq n \leq b_x\} \tag{23}$$

The sets $A$ and $B$ are then defined in an analogous way to in Section 3.5.2.

$$A := S'\backslash\{(x, b(x)) \mid x \in X\} \tag{24}$$

$$B := S' \backslash \{(x, a(x)) \mid x \in X\} \tag{25}$$

The sets $A$ and $B$ are smaller than the alternative definition, where the constraints on $n$ are only applied to one letter $x$, rather than a set of letters $x$. The fact that the set is smaller means that the rewrites in the HNN normalization process are less likely to suceed. This may not seem like a good thing, but it is faster for some examples.

Consider the following Example, a slight modification of Example 3.16.5.

**Example 3.16.3.** Suppose $r = txt^{-1}(xy)^{-2}$, then
$r' := \text{AddSubscript}(t)(r) = x_1(x_0y_0)^{-2}$ and suppose $w = t^n x_1 y_1 t(x_0y_0)^{-1}$, where $n > 0$.

Suppose $S'$ is defined to be the set $\{x_0, x_1, y_0, y_1\}$. Then the only rewrite possible in the HNN normalization process is to substitute $t(x_0y_0)^{-1}$ with $(x_1y_1)^{-1}t$ after which the word will already be in HNN normal form.

If alternatively, $S'$ is defined to be just $\{x_0, x_1\}$, then it is also possible to perform a substitution on the left, rewriting $t^n x_1 y_1$ to $t^{n-1}(y_1x_1)^2 y_2$, after which it is still possible to perform many more rewrites on the left hand side, meaning the word will be normalized in many more steps.

Whilst changing the definition of $S'$ potentially takes away potential rewrites, in at least one example the option it took away was a bad rewrite to make anyway.

**Example 3.16.4.** Consider the word $tw_0t^{-1}w_1tw_2$. Here it is not possible to determine the best order without considering what particular words $w_0, w_1, w_2$ are. Starting in the middle gives the word $tw_0\phi(b)w_2$, and since $w_2$ might not be in $B$, the first problem may have become easier.

However starting on the left with the pair $tw_0$ might also be the best thing to do, since starting on the left would make the second problem easier, giving the word $\phi(bw_1tw_2)$.

**Example 3.16.5.** Consider the word $tw_0tw_1$. Here it is best to apply the right hand rewrite first. Applying the left hand rewrite first will not make the right hand one any easier; the $t$'s will not cancel, but applying the right hand one first could make the left hand problem easier. After applying the right hand rewrite, the word would become $tw_0\phi(a)t$, where $a \in A$ and $\overline{a} = \overline{w_1}$. It is possible that it is easier to check $\overline{w_0\phi(a)} \in \overline{A}$ than to check both $\overline{w_0} \in \overline{A}$ and $\overline{phi}(a) \in \overline{A}$.

Example 3.16.2 and Example 3.16.5 give an optimal normalization order for simple examples, with only two occurrences of a power of $t$. It is not possible to generalize this to more complicated examples, but some sensible heuristics are possible.

The implemented code normalises the word from left to right, always applying a substitution at the leftmost position where one is possible. This was found to have better performance than right to left normalisation, probably because the cancellation in Example 3.16.2 is more likely than the cancellation in the Example 3.16.1.

We describe below a potentially improved way of performing the rewrites in the an order with good performance.

Consider a word $W := w_0 t^{n_1} w_1 \dots t^{n_{k-1}} w_{k-1} t^{n_k} w_k$, and the following two sets

$$R^+ := \left\{ i \mid \forall j, \sum_0^i n_i \leq \sum_0^j n_j \right\} \tag{26}$$

$$R^- := \left\{ i \mid \forall j, \sum_0^i n_i \geq \sum_0^j n_j \right\} \tag{27}$$

Let $I$ be the smaller of the two intervals $[\min R^+, \max R^+]$ and $[\min R^-, \max R^-]$.

Within the interval $I$, let $Q$ be the set of pairs $t^{n_i} w_i$ such that $\operatorname{sgn}(n_i) \neq \operatorname{sgn}(n_{i+1})$.

Then $Q$ has the property that if there are no applicable rewrites in the set $Q$, then the word cannot be put into the form $wt^n$ with $w \in F(S')$. This is because if no rewrite can be performed at the first and last pairs $t^{n_i} w_i$ in the interval, then no substitutions performed outside this interval will make these rewrites possible.

The rewriting procedure always chooses a rewrite within the set $Q$, prioritising pairs $w_i t^{n_i}$ where $w_i$ has the least occurences of letters not in the set $T$. These are likely to be the fastest problems to solve. This heuristic mitigates the exponential behaviour described in Example 3.16.1.

Care must also be taken to ensure that equivalent problems are not attempted twice. For example if a rewrite is attempted at a pair $t^{n_i} w_i$, where $0 < n_i$ and fails because $\overline{w_i} \notin A$ is not in the relevant subgroup, and then later on after substitutions elsewhere, this pair becomes $t^{n_i} w_i \phi^{-1}(w_k)$, then since $\phi^{-1}(w_k) \in A$, then a rewrite is still not possible here so none should be attempted.

## 3.7 Shortening Proofs

Most of the tactic execution time is spent generating and checking the Lean proof, and not on generating the proof certificate described in Section 3.3. An algorithm was defined which implemented some heuristics to shorten the certificates produced. There are two heuristics used to do this.

There are three equalities that the heuristics make use of. Recall that a certificate is an element of $P(F(S))$, which is a semidirect product of $F(S)$ and $F(F(S))$. Given an element $p \in F(F(S))$, we aim to find an element $p' \in F(F(S))$ such that $\text{Eval}(p) = \text{Eval}(p')$ (Definition 3.7). Given a relator $r$, and words $w, v \in F(S)$, the heuristics make use of the following equalities.

$$\text{Eval}([w]) = \text{Eval}([wr^n]) \tag{28}$$

$$\text{Eval}([w][v]) = \text{Eval}([wrw^{-1}v][w]) \tag{29}$$

$$\mathrm{Eval}([w][v]) = \mathrm{Eval}([v][vr^{-1}v^{-1}w]) \tag{30}$$

A total order ($<$) is put on the set of words in $F(S)$, this order has the property the if $\mathrm{Length}(w) < \mathrm{Length}(v)$, then $r(w, v)$. It is not important what the relation is on words of the same length, as long as it is a total order, but the Lean implementation uses a lexicographic ordering.

**Definition 3.17** (Golf$_1$)**.** Golf$_1$ folds through a word $p \in F(F(S))$ and replaces each letter $w \in F(S)$ with the least word of the form $wr^n$. It also performs any cancellations that can be performed after performing these substitutions, to return a reduced word $p' \in F(F(S))$. In practice, there are very often cancellations that can be performed after this normalization of each letter.

**Definition 3.18** (Golf$_2$)**.** Golf$_2$ performs substitutions of the form in Equations 29 and 30. If $wrw^{-1}v$ is less than $v$, it will perform the substition in Equation 29, and if $vr^{-1}v^{-1}w$ is less than $w$ it performs this substitution. It performs these substitiutions until no more can be made. Again, it performs any cancellations that can be performed to return a reduced word.

**Definition 3.19** (Golf)**.** For an element $(p, w) \in P(F(S))$, $\mathrm{Golf}(p, w)$ is defined to be $((\mathrm{Golf}_2 \circ \mathrm{Golf}_1)(p), w)$.

These heuristics were surprisingly effective at shortening the proof certificates. As an extreme example, if $r = aba^{-11}b^4$, and $w = a^{10}b^{-4}a^{11}b^{-1}aba^{-11}b^5a^{-11}ba^{11}b^{-1}a^{-1}b^{-1}a^{-10}$, then the certificate produced by Solve that $\overline{w} = 1$ has length 72 before shortening and length 4 after shortening. This shortens the overall tactic execution from 54 seconds to around 3 seconds, with only 120ms spent executing Golf.

The biggest benefit of the $Golf$ function, is that it reduces the number of letters in a certificate $p \in F(F(S))$, not just the length of each letter $w \in F(S)$. It does this because shortening each letter, is also canonicalising the letters, making cancellation more likely. This is the motivation for putting a total order on the set of letters $w \in F(S)$. For example, if $w$ and $wr$ are the same length, but $w < wr$, then the word $[w][wr]^{-1}$, would not be reduced if Golf only compared lengths of letters, but would be is reduced to 1 by using a total order.

## 3.8 Heuristics

A few heuristics are implemented when there is a simpler method than Magnus' method. They are implemented in the following order

- Check whether the word $w$ is already written using letters in $T$. If $w \in T$, then trivially $\overline{w} \in \overline{T}$
- If there is a letter $x$ in the relator $r$ such that $x \notin T$, but $x$ is in $w$ then by the Freiheitsatz, $\overline{w} \notin \overline{T}$, so the algorithm can fail straight away.

- If $w$ is not in the subgroup generated by $r$ and $T$ after abelianizing the free group, the algorithm can fail straight away.
- If the relation $r$ has exactly one occurrence of a letter, say $x$, then the problem can be solved by rearranging the equation $r = 1$ to the form $x = v$, where $v$ is a word not containing $x$, and making this substitution everywhere in $w$.

### 3.8.1 Injectivity

The correctness of the algorithm relies on the fact that the map $\psi_2$ is an injective map. Since $\psi_2$ is injective, if $p \in P(F(S))$ is a witness of the congruence $\psi_2(a) \equiv \psi_2(b) \bmod \psi_2(r)$, then there must exist a certificate $q$ of the congruence $a = b \bmod r$. The question is how to compute this. The proof of this congruence given in [Put20] relies on the fact that the canonical maps into an amalgamated product of groups are injective. However the standard proof seems to rely on the law of the excluded middle, so it cannot be translated into an algorithm to compute $q$.

Suppose $p \in P(F(S))$ is a witness of the congruence $\psi_2(a) = \psi_2(b) \bmod \psi_2(r)$. It is not necessarily the case that $k$ is a multiple of $n$ in every occurrence of $t^k$ in $p$. For example $p := ([t][tr^{-1}t^{-1}][t]^{-1}, 1) \in P(F(S))$ is a witness of the congruence $r = 1$. Both $\text{lhs}(p)$ and $\text{rhs}(p)$ are in the image of $\psi_2$ for $n = 2$, when $r$ is in the image of $\psi_2$, but $p$ is not in the image of $P(\psi_2)$. However where there are occurrences of $t$, they are all cancelled after lhs is applied, in fact one could remove every occurrence of $t$ from $p$ and still have a certificate of the same congruence.

In practice, it is observed that whenever there is an occurence of $t^k$ it is always the case that $t$ is a multiple of $n$. If this were true all the time, then a slightly simpler algorithm could be possible.

**Definition 3.20.** Given a word $w \in F(S)$, define the set of partial exponent sums of a letter $t \in S$ to be the set of exponent sums of all the initial words of $w$. For example, the partial exponent sums of $t$ in $t^n at$ are the exponent sums of $t$ in $t^n$, $t^n a$ and $t^n at$.

**Definition 3.21.** $h$ is a map $F(S) \to F(S \cup \{t'\})$, where $t'$ is some letter not in $S$. $h$ replaces every occurrence of $t^k$ with $t'^a t^b$ in such a way that $a + nb = k$, and every partial exponent sum of $t'$ in $h(w)$ is either not a multiple of $n$, or it is zero.

**Definition 3.22.** $\theta$ is a group homomorphism $F(S \cup \{t'\})$. Let $s \in S$. Then

$$\theta(s) = \begin{cases} t & \text{if } s = t' \\ t^n & \text{if } s = t \\ s & \textit{otherwise} \end{cases} \tag{31}$$

$\theta$ and $h$ satisfy $\theta \circ h = \text{id}$. For any $w$ in $F(S)$, $\theta(w) = \psi_2(w)$.

**Definition 3.23.** (PowProof) *PowProof* is a map $F(S) \to F(S)$. $\text{PowProof}(w)$ is defined to be $h(w)$, but with every occurrence of $t'$ replaced with 1.

**Theorem 3.24.** For any $p \in F(F(S))$ if
$\text{Eval}(\psi_2(r))(p) = \psi_2(w)$, then $\text{Eval}(r)(F(\text{PowProof})(p)) = w$.

**Lemma 3.24.1.** Consider $\prod_{i=1}^{a} s_i^{k_i}$, as an element of the $F(S \cup \{t'\})$ with $s_i \in S \cup \{t'\}$ (Note that this is not necessarily a reduced word; $k_i$ may be zero and $s_i$ may be equal to $s_{i+1}$). Suppose every partial product $\prod_{i=1}^{b} s_i^{k_i}$, with $b \leq a$ has the property that if the exponent sum of $t'$ is a multiple of $n$, then it is zero. Suppose also that $\prod_{i=1}^{b} s_i^{k_i}$ has the property that for every occurrence of $t'^k$ in the reduced product, $k$ is a multiple of $n$. Then the reduced word $\prod_{i=1}^{a} s_i^{k_i}$ can be written without an occurrence of $t'$.

**Proof of Lemma 3.24.1.** $\prod_{i=0}^{a} s_i^{k_i}$ can be written as a reduced word $\prod_{i=1}^{c} u_i^{k'_i}$ such that $k'_i$ is never equal to zero and $u_i \neq u_{i+1}$ for any $i$. The set of partial products of this, $\prod_{i=1}^{c} u_i^{k'_i}$, is a subset of the set of partial products of $\prod_{i=1}^{a} s_i^{k_i}$, therefore the exponent sum of $t'$ in every partial product of $\prod_{i=1}^{a} s_i^{k_i}$, is either 0 or not a multiple of $n$. However, by assumption every occurrence $t'^k$ in $\prod_{i=0}^{a} s_i^{k_i}$, $k$, is a multiple of $n$, so the exponent sum of $t'$ in every partial product is 0. So $\prod_{i=1}^{a} s_i^{k_i}$ does not contain $t'$.

**Proof of Theorem 3.24**
If $\text{Eval}(\psi_2(r))(p)$ is in the image of $\psi_2$, then $\text{Eval}(r)(F(h)(p))$ has the property that for every occurrence of $t'^k$, $k$ is a multiple of $n$. If $p' := F(h)(p)$, then $\text{Eval}(r)(p')$ can be written as a product of the form in Lemma 3.24.1. If $r' = \prod_i u_i^{l_i}$, then to write $\text{Eval}(r)(p')$ in this form, send $\prod_i \left[\prod_{j=1}^{a} s_{ij}^{k_j}\right] \in P(F(S \cup \{t'\}))$, to

$$\prod_i \left( \left( \prod_{j=1}^{a} {s_{ij}}^{k_j} \right) \left( \prod_j u_j^{l_j} \right) \left( \prod_{j=1}^{a} {s_{i(a-j)}}^{-k_{a-j}} \right) \right) \tag{32}$$

If all the nested products in Equation 32 are appended into one long product, then the product has the form in Lemma 3.24.1. Therefore when the word is reduced it will not contain $t'$ by Lemma 3.24.1. This means that deleting all occurrences of $t'$ will in $p'$ will not change $\text{Eval}(r)(p')$, and therefore $\text{Eval}(r)(F(\text{PowProof})(p)) = \text{Eval}(r)(F(h)(p))$. Applying $\psi_2$ to both sides gives $\psi_2(\text{Eval}(r)(F(\text{PowProof})(p)) = \psi_2(\text{Eval}(r)(F(h)(p))) = \theta(\text{Eval}(r)(F(h)(p))) = \text{Eval}(\psi_2(r))(p)$.

## 4 Efficiency of the Algorithm

The worst case performance of this algorithm is worse than any finite tower of exponents [MUW11]. The more relevant question is what is the typical performance.

**Definition 4.1** (Area of a Relation)**.** For a finitely presented group $G := \langle S|R \rangle$, if $w \in F(S)$ is equal to 1 in the quotient $G$, then we say it is a *relation.* The *area of a relation* is the smallest $N$ such that $w$ can be written in the form $\prod_{i=1}^{N} g_i r_i^{\epsilon_i} g_i^{-1}$, where $\epsilon_i = \pm 1$ and $r_i \in R$ for all $i$.

**Definition 4.2** (Dehn Function)**.** For a finitely presented group $G := \langle S|R \rangle$, the Dehn function of the presentation $\text{Dehn}(n) \in \mathbb{N}$ is defined as the largest area of a relation of length at most $n$.

The Dehn function puts a lower bound on the complexity of the one-relator algorithm. The area of a relation is by definition the length of the shortest certificate that the algorithm might produce, so the complexity of the algorithm is bounded above by the Dehn function of a relator. The group $\langle a, b | bab^{-1}aba^{-1}b^{-1} = a^2 \rangle$ is such that Dehn$(n)$ is worse than any finite tower of exponents. This means that the complexity of the one relator algorithm is also worse than any finite tower of exponents.

Not all groups have such a fast growing Dehn function. For example, if the relator is of the form $r^k$ with $|k| \neq 1$ then Dehn$(n) \leq n$. Similarly, even in groups with a rapidly increasing Dehn function, there are words that do not have a large area as the worst case.

So, even though the worst case behaviour is very bad, there are still potentially many problems that the algorithm could solve in a practical amount of time. The aim of this implementation was to have good performance on relations with a small area. A typical Lean tactic state will usually be used on problems where the author knows the solution, but simply needs automation to write a formal proof of the solution. These relations will usually have a very small area. The aim of this implementation was that the algorithm should have good performance on relations with a small area, but makes no attempt to solve problems where the area of the relation is very large.

## 5 Graph Search Method

In this section we present an alternative method to solve word problems in groups. This method searches for a sequences of rewrites to prove an equality. This search will usually not terminate if there is no such sequence of rewrites. I conjecture that it will terminate whenever there is a such sequence of rewrites. It was inspired by an online solver written Kyle Miller, with some modifications since that solver would sometimes fail to prove true formulas.

This section describes a solver that attemps to prove a word in the free group is in the normal closure of a finite set of relators. It then describes a Lean tactic that makes use of this solver, but employs some methods to convert various Lean tactic states into a form that can be tackled by the solver.

Just as was the case with the tactic for Magnus' Method, the tactic has two parts, a solver that determines whether a word is in the normal closure of a set of relators, and uses a more efficient representation of elements of the free group, and keeps track of some sort of proof certificate, and a Lean tactic that translates a Lean tactic state into a problem about elements in the efficient representation of the free group, and writes a formal Lean proof of the result using the certificate.

## 5.1 Substitution heuristic

Sometimes it happens that it is very straightforward to eliminate a variable. For example, if one of the relators has exactly one occurence of one of the letters, then this letter and the relator can easily be eliminated.

For example, consider the following problem.

$$\begin{aligned} aba &= b^{-1}c \\ bac^2ba^2 &= 1 \\ &\vdash ab = ba \end{aligned} \tag{33}$$

The first equation can be rewritten as $c = baba$, which means that $c$ can be eliminated from the problem, and the new problem is

$$\begin{aligned} babababa^2 &= 1 \\ &\vdash ab = ba \end{aligned} \tag{34}$$

## 5.2 Outline

Given a set of relators $R$, the method first generates a set of rewriting rules from the set of relators. Given a relator, the algorithm generates all equalities that can be made using the generator such that there is one letter of the starting relator on the left hand side of the equality. For example, if a relator is $abab^2$, then the equalities generated are

$$\begin{gathered} a = b^{-2}a^{-1}b^{-1}, a^{-1} = bab^2 \\ b = a^{-1}b^{-2}a^{-1}, b^{-1} = ab^2a \\ a = b^{-1}a^{-1}b^{-2}, a^{-1} = b^2ab \\ b = a^{-1}b^{-1}a^{-1}b^{-1}, b^{-1} = babab = b^{-1}a^{-1}b^{-1}a^{-1}, \quad b^{-1} = abab \end{gathered} \tag{35}$$

Given a starting word, the algorithm then generated all words that can be generated from this starting word rewrites using the above rules and adds these words to a set of leaves. The process is then repeated at the word in the set of leaves with the least cost. The cost function assigns a natural number to each word in the free group. In general, shorter words have a lower cost, but different cost functions are discussed in section TODO. Before being added to the set of leaves, each word is cyclically reduced as well, a word is replaced by the shortest of its conjugates.

As an example, we could apply the first rewriting rule $a = b^{-2}a^{-1}b^{-1}$ to $aba^{-1}b^{-1}$ and obtain the word $b^{-2}a^{-1}b^{-1}ba^{-1}b^{-1} = b^{-2}a^{-2}b^{-1}$ which is then added to the set of leaves. The process is then repeated from the word in the set of leaves with the least cost, taking care not to repeat any words, until the word is rewritten to 1.

In summary, the process is as follows given a word $w$ that we are trying to prove is equal to 1.

1. Eliminate as many variables as possible using the method outlined in Section 5.1
2. Generate the set $G$ of all rewriting rules such that there is one letter on the left hand side that can be generated from the starting relators.
3. We store two sets of words, a set of seen words $S$, and a set of leaves $L$.
4. Add the word $w$ to the set of seen words leaves $S$
5. Generate all new words that can be made by applying a rewrite rule in $G$, and add these words to the set $L$ after cyclically reducing them.
6. Take the word $w$ in $L$ with the least cost. If it is equal to 1 then stop. Otherwise, check whether this word is in $S$. If it is not, then remove it from $L$ and go to step 3, otherwise remove it from $L$ and repeat this step.

A slightly different approach to the above approach would be to check if a word is in the set $S$ of seen words at step 5, and add it to both sets $S$ and $L$ if it was not in $S$, and neither set otherwise, and then there would be no need to check if words were in $S$ at step 6, and the set $L$ would be much smaller. However, this was found to be a lot slower, since so much time was spent checking whether words were in $S$.

## 5.3 Cost Function

The method mentions a cost function which was not defined yet. Two cost functions were tested, one was simply the length of a word, the other effectively ordered words by length first and then lexicographically, after the letters are put in some arbitrary order.

The second cost function was defined as follows. Suppose $n$ is the total number of letters in all words in $R$ and the target word $w$. Label the letters $a_1, \dots a_n$. Then the cost of a word is defined inductively on the length of the word as follows

$$\mathrm{Cost}(1) = 0 \tag{36}$$

$$\mathrm{Cost}(a_i^{\pm 1} w) = i + n\mathrm{Cost}(w) \tag{37}$$

This cost function was found to have better performance than simply using the length of a word as the cost function. As an example consider the problem of proving that if $ab = ba$, then $a^5b^5 = b^5a^5$. Using length as the cost function, the size of the search graph for this problem is 5906, but using the improved cost function, the search graph size contains 604 words. This method is obviously not the best method for problems in abelian groups like this, but the improved cost function should also speed up the search in problems where some, but not all of the variables commute.

## 5.4 Generating Proofs

In order to generate proof terms the algorithm must keep track of what path was taken to rewrite a word to the identity. The following information is stored at each node in the search graph, in the data type `path_step`

```
@[derive inhabited] structure path_step : Type :=
(rel_index : ℕ) -- Index of the relator used to make the rewrite
(rel_is_inv : bool) -- Boolean representing whether the relator or
  --its inverse was used to make the substitution
(old_word : free_group) -- word before the substitution was made
(new_word : free_group) -- word after the substitution was made
(new_word_cost : ℕ) -- cost of the new word
(word_letter_index : ℕ) -- Letter index in the old word,
  -- indicating where the substitution was made
(rel_letter_index : ℕ) -- Letter index in the relator indicating
  -- which letter in the relator was substituted.
```

This information is enough to retrack the path that was taken and write a proof that will always take the same standard form.

Given words $\texttt{word}_1$ `rel` $\texttt{word}_2$ `conj old_word new_word rel_conj` in the free group, it is true that if $\texttt{conj}^{-1}$ `*` $\texttt{word}_1$ `*` $\texttt{rel\_conj}^{-1}$ `* rel * rel_conj *` $\texttt{word}_2$ `* conj` is in the normal closure of a set of relators, $\texttt{word}_1$ `*` $\texttt{word}_2$ `= old_word`, and `rel` is one of the relators, then `old_word` is also in the normal closure of the set of relators.

Intuitively, we are splitting `old_word` into two parts, $\texttt{word}_1$ and $\texttt{word}_2$, inserting some conjugate of `rel` in between $\texttt{word}_1$ and $\texttt{word}_2$, and then conjugating the result as well.

# 6 The Tactic

The practical problems that might come up in Lean proofs sometimes are not already in exactly the form that can be solved by the algorithm described, and some work is needed to convert them to a form that can be solved.

As an example, consider the following proposition. If $a$ and $b$ are elements of a group, and $ab = b^2a$, then for any natural number $n$, $a^nba^{-n} = b^{2^n}$. The proof is by induction, and we consider the inductive step. The information visible to the user at the inductive step is as follows

For the inductive step of this proof, we have the hypothesis that $ab = b^2a$, the induction hypothesis says that $a^nba^{-n} = b^{2^n}$, and we need to prove $a^{n+1}ba^{-(n+1)} = b^{2^{n+1}}$.

This can be converted into a word problem. Our atoms will be $a$, $a^n$, $b$ and $b^{2^n}$ and we add assumptions for commuting powers, because $a$ commutes with $a^n$, and $b$ commutes with $b^{2^n}$.

We will introduce new variables $c := a^n$ and $d := b^{2^n}$. The problem then becomes the following

$$\begin{aligned} ac &= ca \\ bd &= db \\ ab &= b^2 a \\ cbc^{-1} &= d \\ \vdash acbc^{-1}a^{-1} &= d^2 \end{aligned} \tag{38}$$

We now descibe some of the process of converting the Lean tactic state into the problem above. The tactic has to effectively identify the atoms, `a` , `a ^ n` , `b` , and `b ^ 2 ^ n` .

```
G : Type,
_inst_1 : group G,
a b : G,
h : a * b = b ^ 2 * a,
n : ℕ,
ih : a ^ n * b * a ^ -↑n = b ^ 2 ^ n
⊢ a ^ n.succ * b * a ^ -↑(n.succ) = b ^ 2 ^ n.succ
```

Note that `n.succ` means $n+1$, or the successor of $n$. Another thing to note is the coercion arrow ↑ . This arrow represents the canonical map from the natural numbers to the integers - usually in maths, this is not explicitly written, but in Lean it must be used, and this provides a small challenge in identifying the atoms, care has to be taken to ensure that the tactic identifies that `a ^ -↑n` is indeed the inverse of `a ^ n` .

Formally speaking the `^` notation is being used for two distinct functions in this tactic state. One is a binary function whose domain is $G \times \mathbb{Z}$, the other has domain $G \times \mathbb{N}$. There is a lemma in Lean identifying that these two functions are compatible, in other words, that applying the natural number power function, is the same as applying the canonical map from $\mathbb{N}$ to $\mathbb{Z}$, and then applying the integer power. The tactic must invoke this lemma in order to prove that `a ^ -↑n` is the inverse of `a ^ n` .

The tactic must also identify that `b ^ 2 ^ n.succ` is equal to `(b ^ 2 ^ n) ^ 2` .

In order to transform this tactic state into the problem in Equation 38, the hypotheses and the goal must first be rewritten. Below is an incomplete list of some of the rules used to normalize the goal.

- Normalize all numerals to the form $1 + 1 + 1 + \dots$
- Normalize all powers to use the integer power, not the natural power
- Rewrite expression of the form $a^(m + n)$ to $a^m a^n$
- Apply the distributivity law within exponents, expand $a(b + c)$ to $ab + ac$.

Lean has an inbuilt tactic called `simp` which is used to perform the above substitutions.

Normalizing all numerals to the form $1+1+1+\ldots$, may seem very inefficient, but whilst it is certainly not the fastest approach, the solving algorithm has such poor performance on large exponents, that it is not made that much worse by this approach, and the tactic should probably not be used with large exponents anyway.

After this normalization procedure is applied, the new tactic state is as follows. Note that this stage will be invisible to the user of the tactic.

```
G : Type,
_inst_1 : group G,
a b : G,
h : a * b = b * b * a,
n : ℕ,
ih : a ^ ↑n * b * (a ^ ↑n)⁻¹ = b ^ ↑((1 + 1) ^ n)
⊢ a * a ^ ↑n * b * (a ^ ↑n * a)⁻¹ = b ^ ↑((1 + 1) ^ n) * b ^ ↑((1 + 1)
  ^ n)
```

Now it is easier to identify the atoms, since `a ^ ↑n` appears in exactly the same form wherever it appears, and similarly, `b ^ ↑((1 + 1) ^ n)` appears as the exact same expression each time it appears. It is now easy to automatically convert it to the problem in Equation 38.

## 7 Comparison of Methods

In this section we compare the two methods, Magnus' method and the graph search method. We also compare two different superposition provers, one implemented in Lean called `super`, and another much more optimised one called SPASS, which does not produce a proof verifiable by Lean.

A superposition is a more general automated theorem prover than either of the two methods described above; it is a general purpose first order logic prover. In order to use these provers the problems must be stated in a slighty different form to have reasonable performance. As an example, the problem of checking if $a$ is in the normal closure of the relators $aba^{-1}b^{-2}$ and $bab^{-1}a^{-2}$, would be written in the following form. We assume we have four unary function symbols in our first order language, $a$, $b$, $a'$ and $b'$, and we give the prover the following problem to prove. In this format, there is no need to explicitly apply the associativity law, which leads to a large performance improvement. The `super` tactic cannot solve even quite straightforward problems without stating them in the form below.

$$\begin{aligned}
\forall x, a(a'(x)) &= x \\
\forall x, a'(a(x)) &= x \\
\forall x, b(b'(x)) &= x \\
\forall x, b'(b(x)) &= x \\
\forall x, a(b(a'(b'(b'(x))))) &= x \\
\forall x, b(a(b'(a'(a'(x))))) &= x \\
\vdash \forall x, a(x) &= x
\end{aligned} \tag{39}$$

It is important to note that this is a comparison of one implementation of Magnus' method, and that other implementations that use the optimizations for the HNN normalization procedure described in Section 3.6.2 may perform much better on particular examples. The implementation of Magnus' method that we are comparing here uses a simple left to right HNN normalization procedure, where the leftmost possible rewrite is always attempted first.

Both tactics work in two stages, one stage solves the word problem, and a second stage to produce and check the formal Lean proof. For the best comparison of the two methods, it is best not to include the time taken to write and check a Lean proof, since this takes a lot of the time, and the differences in performance are not a reflection of the merits of one method over another, but instead reflect two different methods to produce a Lean proof. Both times are included in the table below however, the "solver time" is the time taken excluding the proof writing and checking time, and the "tactic time" is the total time taken to execute the tactic. For the superposition provers, the time taken for the `super` tactic to prove a problem is best compared with the tactic execution time of the two methods described in this paper, whilst the time taken by SPASS is best compared with the "solver" time for the two methods described in this paper.

For both methods we include the length of the solution found. For Magnus' method this means the length of the word in $F(F(S))$ which is computed as the certificate. For the graph search method, this means the number of substitutions required in the proof. We also include the tree size for the graph search method, which is the size of the set $L$ at the end of the computation.

| Label | Relators | Word | Magnus' method | | | Graph search method | | | | super | SPASS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Tactic | Solver | Length | Tactic | Solver | Length | Tree | Tactic | Solver |
| 1 | $abab$ | $babab^{-1}a^{-1}b^{-1}a^{-1}$ | 326ms | 1ms | 2 | 673ms | 1ms | 2 | 4 | 369ms | 20ms |
| | | \$(baba)ˆ4 | 337ms | 6ms | 4 | 782ms | 6ms | 4 | 4 | 683ms | 20ms |
| | | $(b^{-1}a^{-1}b^{-1}a^{-1})^4$ | 514ms | 1ms | 4 | 594ms | 5ms | 4 | 4 | 1.29s | 20ms |
| | | $(baba)^{10}$ | 669ms | 2ms | 10 | 1.73s | 31ms | 10 | 10 | >600s | 20ms |
| | | $(baba)^{100}$ | 4.26s | 10ms | 100 | 66.2s | 1.54s | 100 | 100 | | 30ms |
| | | $(baba)^{200}$ | 5.23s | 19ms | 200 | 436s | 6.61s | 200 | 200 | | 60ms |
| 2 | $aba^{-1}b^{-2}$ | $aba^{-1}bab^{-1}a^{-1}b^{-1}$ | 390ms | 2ms | 2 | 577ms | 64ms | 2 | 20 | 1.32s | 20ms |
| | | $a^2ba^{-2}ba^2b^{-1}a^{-2}b^{-1}$ | 324ms | 6ms | 6 | 529ms | 20ms | 6 | 91 | 5.17s | 20ms |
| | | $a^5ba^{-5}ba^5b^{-1}a^{-5}b^{-1}$ | 5.14s | 141ms | 62 | 5.60s | 3.89s | 70 | 20193 | >600s | 90ms |
| | | $a^9ba^{-9}ba^9b^{-1}a^{-9}b^{-1}$ | | 920ms | 1022 | | | | | | 160s |
| 3 | $(ac)b(ac)^{-1}b^{-2}$ | $acb(ac)^{-1}bacb^{-1}(ac)^{-1}b^{-1}$ | 180ms | 4ms | 2 | 277ms | 2ms | 2 | 20 | 3.3s | 20ms |
| | | $(ac)^2b(ac)^{-2}b(ac)^2b^{-1}(ac)^{-2}b^{-1}$ | 420ms | 7ms | 6 | 775ms | 13ms | 6 | 90 | 7.1s | 20ms |
| | | $(ac)^5b(ac)^{-5}b(ac)^5b^{-1}(ac)^{-5}b^{-1}$ | 4.97s | 57ms | 62 | 4.06s | 1.28s | 62 | 6338 | >600s | 40ms |
| 4 | $aba^{-1}b^{-2}$, $bca^{-1}$ | $aba^{-1}bab^{-1}a^{-1}b^{-1}$ | NA | | | 297ms | 2ms | 2 | 24 | 13.8s | 20ms |
| | | $a^2ba^{-2}ba^2b^{-1}a^{-2}b^{-1}$ | | | | 490ms | 19ms | 6 | 100 | 45.5s | 20ms |
| 5 | $aba^{-1}b^{-1}$ | $a^2b^2a^{-2}b^{-2}$ | 413ms | 13ms | 4 | 366ms | 6ms | 4 | 34 | 446ms | 20ms |
| | | $a^5b^5a^{-5}b^{-5}$ | 1.45s | 69ms | 25 | 1.14s | 100ms | 25 | 604 | 1.14s | 20ms |
| | | $a^{10}b^{10}a^{-10}b^{-10}$ | 4.21s | 110ms | 100 | 5.83s | 1.35s | 100 | 5314 | 106s | 30ms |
| 6 | $acb(ac)^{-1}b^{-1}$ | $(ac)^2b^2(ac)^{-2}b^{-2}$ | 215ms | 5ms | 4 | 339ms | 4ms | 4 | 34 | 3.34s | 20ms |
| | | $(ac)^5b^5(ac)^{-5}b^{-5}$ | 1.16s | 33ms | 25 | 1.17s | 96ms | 25 | 604 | 8.91s | 20ms |
| | | $(ac)^{10}b^{10}(ac)^{-10}b^{-10}$ | 4.32s | 92ms | 100 | 7.95s | 1.16s | 100 | 5314 | MEM | 20ms |
| 7 | $aba^{-3}b^4$ | $a^2baba^{-3}b^3a^{-2}b^{-4}a^3b^{-1}a^{-1}$ | 689ms | 359ms | 2 | 329ms | 12ms | 2 | 86 | 12.8s | 30ms |
| 8 | $aba^{-11}b^4$ | $a^{10}baba^{-11}b^3a^{-10}b^{-4}a^{11}b^{-1}a^{-1}$ | 1.42s | 768ms | 2 | 782ms | 56ms | 2 | 230 | >600s | 20ms |
| 9 | $aba^{-1}b^{-3}$ | $a^4baba^{-1}b^{-4}a^{-3}ba^{-1}b^{-3}$ | 11.1s | 67ms | 80 | 439ms | 6ms | 2 | 52 | 327ms | 20ms |
| 10 | $aba^{-1}b^{-2}$, $bab^{-1}a^{-2}$ | $a$ | NA | | | 337ms | 215ms | 5 | 1368 | 4.67s | 30ms |

| Label | Relators | Word | Magnus' method | | | Graph search method | | | | super | SPASS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Tactic | Solver | Length | Tactic | Solver | Length | Tree | Tactic | Solver |
| 11 | $aca^{-1}c^{-1}$, $bdb^{-1}d^{-1}$, $aba^{-1}b^{-2}$, $cbc^{-1}d^{-1}$ | $acbc^{-1}a^{-1}d^{-2}$ | | NA | | 492ms | 13ms | 3 | 120 | MEM | 30ms |
| 12 | $aca^{-1}c^{-1}$, $b^2d^2b^{-2}d^{-2}$, $ab^2a^{-1}b^{-4}$, $cb^2c^{-1}d^{-2}$ | $acb^2c^{-1}a^{-1}d^{-4}$ | | NA | | 1.50s | 61ms | 5 | 216 | 25.9s | 20ms |
| 13 | $aca^{-1}c^{-1}$, $c^{-1}dcdc^{-1}d^{-1}cd^{-1}$, $ac^{-1}dca^{-1}c^{-1}d^{-2}c$ | $ada^{-1}d^{-2}$ | | NA | | 570ms | 15ms | 3 | 120 | 18.7s | 20ms |
| 14 | $aca^{-1}c^{-1}$, $aba^{-1}b^{-2}$, $cbc^{-1}bcb^{-1}c^{-1}b^{-1}$ | $acbc^{-1}a^{-1}bacb^{-1}c^{-1}a^{-1}b^{-1}$ | | NA | | 502ms | 34ms | 8 | 354 | 35.9s | 30ms |

For problems where the single relator is of the form $r^n$, where $r$ is a word in the free group, and $n > 1$, then both methods perform very well. There is a theorem (proper citation) that says that these problems can be solved whilst only performing rewrites that make the word shorter. Neither method will perform a rewrite that makes the word longer on these problems, so both perform well. The table above shows the performance of each algorithm for the relator $abab$, labelled 1 in the table. Magnus' method appears to be linear time on these problems, whereas the graph search method is not. To prove $(baba)^n = 1$, will require $n$ steps and at each step the number of rewrites the algorithm will add to the graph will be proportional to the length of the graph, so one would expect at least quadratic time. The overall tree size is small because most of the words added to the graph will be duplicates of words already in the graph. The `super` tactic performed particularly badly here, which is surprising given how easy it is to simplify some of the words.

The next problem we compare, is given the relator $ab = b^2a$, prove that $a^nba^{-n}b = ba^nba^{-n}$ for various different values of $n$. The reason this is true is because in this particular group, $a^nba^{-n} = b^{2^n}$, and therefore $a^nba^{-n}$ must commute with $b$. Therefore, in order to prove the result, the graph search method must perform rewrites that make the word longer, whereas it aims to make words shorter, and will only perform rewrites that make a word longer after having exhausted all rewrite sequences that do not make the word longer. So one would expect the graph search method to perform poorly on these problems, and this is corroborated by the data, particularly the solve time, although the overall tactic execution time is similar. The superposition provers both performed badly on this problem as well, this was the only problem where either of my algorithms outperformed SPASS, since Magnus' method was much faster for large values of $n$.

I also experimented with the same relation but changing each occurence of $a$ with $ac$. This worsened the performance of `super`, perhaps because the extra letter $c$ necessitated the inclusion of two new function symbols, $c$ and $c'$, and two new equalities, $\forall x, c(c'(x)) = x$ and $\forall x, c'(c(x)) = x$ in the format that the problem is inputted to `super`.

I experimented here with using the same problem as above, with relator $ab = b^2a$, and attempting to prove $a^nba^{-n}b = ba^nba^{-n}$, but adding an entirely superfluous relator $bca^{-1}$. This significantly damaged the performance of `super`, but the performance of the graph search method was only slightly worse.

We also compare the relator $aba^{-1}b^{-1}$ and words of the form $a^nb^na^{-n}b^{-n}$ for different values of $n$. For this problem Magnus' method has the edge although of course neither method is a sensible choice for this problem. Again, Magnus' method particularly outperforms the search method on the solve time, although overall tactic execution times are similar.

The next three problems, labelled 7, 8 and 9, all have a very easy solution with only two subsitutions required. The problems labelled 8 and 9 are both problems where Magnus' method performs very badly. They both exhibit behaviour similar to the behaviour in Example 3.16.1. The graph search method is much faster on both of these examples, since they can both be done in just two substitutions, both of which make the word shorter. Note the difference in length between the solution found by Magnus' method in each of the final two problems. The first has length 80, which is much longer than the minimum required to solve the problem, which is 2. For the final problem it did find a solution of length 2, but this was only after the golfing heuristics described in Section 3.7. For this problem it initially found a certificate of length 36, before the golfing code shortened it. Surprisingly, `super` performed extremely badly on problems 7 and 8, despite them only requiring two substitutions, both of which made the word much shorter.

The final four problems are all related to the group with presentation $\langle a, b\ : ab = b^2a\rangle$. Problem 11 is the inductive step of the proof that $a^nb = b^{2^n}a$ in this group. `super` failed to prove this within ten minutes. The graph search method was able to eliminate the variable $b$, and this may have given it an advantage over `super`. Therefore problem 12 is the same problem, but with every occurence of $b$ replaced with $b^2$ and with every occurence of $d$ replace with $d^2$. It is of course still possible to eliminate $b$ or $d$, but the algorithm does not attempt this when every occurence is a square. The graph search method still performed well on this problem, and surprisingly this change meant that `super` was naw able to solve the problem, albeit very slowly. I also tried making it easier for `super`, by eliminating $b$ before giving the problem to `super` in the problem labelled 13. `super` was then able to solve it.

The final problem in the table is the inductive step of the proof that $a^nba^{-n}$ and $b$ commute in the group $\langle a, b\ : ab = b^2a\rangle$. The graph search method performs very well on this problem, but `super` is slow.

In conclusion, Magnus' method appears to be better than either the graph search method or `super` on most one-relator problems. The problems where it performs badly are carefully constructed to exploit the weaknesses, and these weaknesses could be fixed by implementing the optimizations in Section 3.6.2. Magnus' method even outperforms SPASS on one problem, so there is weak evidence to suggest this method is overall faster than a superposition prover, especially considering that SPASS has had far more time invested in optimizing it than my implementation of Magnus' Method. However it is difficult to draw any firm conclusions because of the drastic variation in performance depending on the problem given. Both the graph search method and Magnus' method outperform `super`.

# 8 To Do

- Mention crappy justification for termination of search.

## References

[Col86] Donald J. Collins, A simple presentation of a group with unsolvable word problem, Illinois J. Math. **30** (1986), no. 2, 230–234.

[dMKA+18] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer, The lean theorem prover (system description), Jun 2018.

[mC20] The mathlib Community, The lean mathematical library, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (2020).

[Mil68] Charles F. Miller, On Britton's theorem A, Proceedings of the American Mathematical Society **19** (1968), no. 5, 1151–1154 (eng).

[MS73] James McCool and Paul E. Schupp, On one relator groups and hnn extensions, Journal of the Australian Mathematical Society **16** (1973), no. 2, 249–256.

[MUW11] Alexei Miasnikov, Alexander Ushakov, and Dong Wook Won, The word problem in the baumslag group with a non-elementary dehn function is polynomial time decidable, 2011.

[Put20] Andrew Putman, One relator groups.