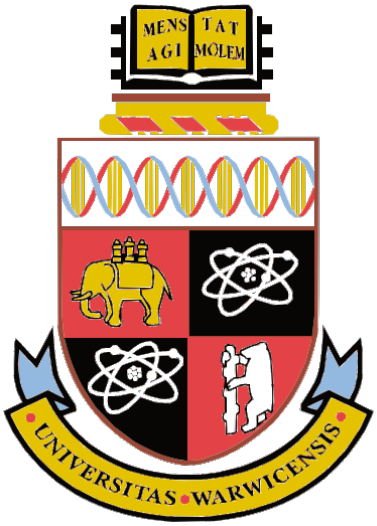


The Complexity of the Compressed Word Problem for Finitely Generated Free Groups

by

David Guo

MA4K9 Dissertation

Submitted to The University of Warwick

Mathematics Institute

April, 2020

THE UNIVERSITY OF
WARWICK

Student ID: 1618719

This student has been formally diagnosed with Specific Learning Differences.
Please make appropriate allowance when marking. Guidance is available at:

http://www2.warwick.ac.uk/services/tutors/disability/guidance

Disability Services

Contents

1 Introduction

In a finitely generated free group, given a word w with finite length l, how long would it take to check if w represents the identity element? One obvious approach is to start from the beginning of w, go through every pair of consecutive characters to identify all the cancellations and remove them. After that we start from the beginning of the reduced word and we repeat the same process until there are no more cancellations. It is clear that the number of pairs we need to check is bounded by a quadratic equation in l. It turns out there is a faster approach, using the idea that after removing the first cancellation, we examine the pair of characters around the place where we had our first removal and repeat the process. Indeed the improved method only needs to go through at most $l-1$ pairs of characters. What we have just described are quadratic and linear algorithms that solve the word problem in the finitely generated free group. But what if w happens to contain some repetitions of strings of characters. Is there a way to represent w such that the complexity of the algorithm could be improved? This has a strong resemblance to a computer science technique called *compressed word*, the simplest form of compression, where we look for repeated patterns in the data and store every repetition by a token to save disk space.

The study of the word problem can be dated back to 1910, it was one of the problems Max Dehn posted on his paper [2], when he considered the problem of determining whether two knots are the same. Then Lohrey brought in computer science techniques to study a slight variation of the word problem when the input is given by a compressed *straight-line program*. In his paper [7], he proved that the compressed word problem for the finitely generated free group can be solved in polynomial-time. After realising the similarity of compressed words and automorphisms, Schleimer [11] used Lohrey's theorem as a stepping stone to prove that the word problems for free-by-cyclic groups and automorphisms of free groups are soluble in polynomial-time .

In this thesis, we will explain the algorithms described in Schleimer's paper [11] in more detail, with most of the definitions, lemmas and theorems coming from there. The thesis is structured as follows: in Section 2, we first take a look at the two types of compressed words called straight-line programs and composition systems, then we introduce some polynomial-time algorithms. Section 3 establishes some results which are required to prove Lohrey's theorem [11]. Then in Section 4, we take a look at the polynomial-time algorithm for the word problem in free-by-cyclic groups which was one of extension problems solved by Schleimer. In Sections 5 and 6, we give a detailed description of Hagenah's [4] and Plandowski's [10] algorithms which Lohrey's theorem depends on. Finally, in Section 7, we explore how we can implement all of the algorithms we described in the previous sections using Matlab, which could be used to solve problems in practice.

2 Straight-Line Programs and Composition Systems

Straight-Line Programs

We will first introduce some terminology. Let $\mathcal{L} = \{a_1, a_2, \ldots, a_m\}$ be a set of characters, which we shall call our *alphabet*. A *word* is a finite sequence of characters in $\mathcal{L}$. We denote by $\mathcal{L}^*$ the set of all words in $\mathcal{L}$.

For a word $w \in \mathcal{L}^*$, denote by $|w|$ the *length* of w: the number of characters in w. Denote ϵ as the *empty* word: the word of length 0. For $u, v \in \mathcal{L}^*$, we say $u = v$ if and only if u and v are identical as strings. Finally, we denote $u \cdot v$ as the *concatenation* of u and v: the word formed by joining u and v together end-to-end.

Definition 2.1. A straight-line program $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A_r, \mathcal{P} \rangle$ contains the following:

(i) A finite alphabet $\mathcal{L} = \{a_1, a_2, \ldots, a_m\}$, the elements of which will be called *terminal*[1] characters.

(ii) A disjoint finite alphabet $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$, the elements of which will be called *non-terminal* characters.

(iii) A non-terminal $A_r \in \mathcal{A}$ which we shall call the *root* of the program.

(iv) A set of *production rules* $\mathcal{P} = \{A_i \to W_i\}_{i=1}^n$. The production rule of the form $A_i \to W_i$ allows us to replace a non-terminal A_i with its *production*[2]: a word W_i in $\mathcal{L}^* \cup \mathcal{A}^*$. Every non-terminal A_j appearing in W_i has index $j < i$.

Note that the indices for the non-terminals are defined so that the production of every non-terminal only depends on the previous non-terminals. When the indices are not important, we often write A as the root to simplify the notation.

Notation. Given a straight-line program $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$, denote by $w(A) = w_A$ the word that results from running the straight-line program. That is, replace A with its production W, then replace all non-terminals appearing in W with their productions, and continue doing so until the resulting word lies in $\mathcal{L}^*$. For a general non-terminal $A_i \in \mathcal{A}$, define $w(A_i) = w_{A_i}$ to be the word from running the straight-line program $\mathbb{A}$ with root A_i.

To further explain the term "root", we define the *production tree* of a character as follows: the tree for a terminal $a \in \mathcal{L}$ is a single vertex labelled a. The tree for a non-terminal A_i is a planer graph defined inductively. The root vertex is labelled A_i. Attach every character in the production W_i to the vertex A_i in left to right order. Then for every non-terminal A_j appeared in W_i, attach its production to it using the same process. Repeat the process

[1] Refine $\mathcal{L}$ if needed, we may assume that every terminal in $\mathcal{L}$ appeared in some productions in $\mathcal{P}$.

[2] In Schleimer's paper [11], he defined the production to be a word in $(\mathcal{L} \cup \mathcal{A})^*$, we can restrict it to be a word in $\mathcal{L}^* \cup \mathcal{A}^*$ without loss of generality: we can assign every terminal to a non-terminal at the beginning of the production rules. Then if a production has both terminals and non-terminals, we can replace all of the terminals by their corresponding non-terminals (e.g. if $A \to a, B \to A \cdot a$, we can change the production of B to $B \to A \cdot A$).

until all the leaves of the tree are characters in $\mathcal{L}$ (See Example 2.4 and Figure 1 for an example). Note that w_A is exactly the word appearing at the leaves of the production tree of A (from left to right).

Definition 2.2. For $B \in \mathcal{L} \cup \mathcal{A}$, the *height* of B, denoted by $\|B\|$, is the height of the production tree of B: the maximal distance from the root B to a leaf (each edge has unit length).

For example, terminals have height zero. Also, the height of every non-terminal on the left hand side of a production is strictly bigger than the height every non-terminal on the right hand side. Lastly, for every $A \in \mathcal{A}$, we have $\|A\| \leq |\mathcal{A}|$.

Definition 2.3. In a straight-line program, a production rule W_i is in *normal form* if it has either the form

(i) $W_i = A_j \cdot A_k$ with length 2 and A_j, $A_k \in \mathcal{A}$ or

(ii) $W_i = a_j$ with length 1 and $a_j \in \mathcal{L}$.

A straight-line program $\mathbb{A}$ is in *(Chomsky) normal form* if every production $W_i \in \mathcal{P}$ is in normal form. In the case when $\mathbb{A}$ has only one single non-terminal, A, with the production $A \to \epsilon$, we also say it is in normal form.

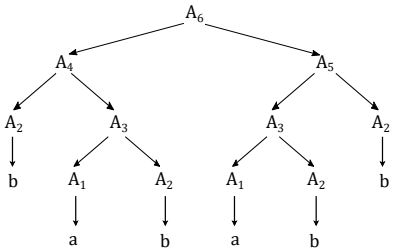


Figure 1: Production tree for A_6 in Example 2.4

Example 2.4. Consider the following straight-line program

$$\mathbb{A} = \left\langle \begin{array}{l} \{a, b\}, \{A_i\}_{i=1}^{2m}, A_{2m}, \{A_1 \to a, A_2 \to b\} \cup \\ \{A_{2k-1} \to A_{2k-3} \cdot A_2, A_{2k} \to A_{2k-2} \cdot A_{2k-1}\}_{k=2}^{m} \end{array} \right\rangle.$$

We have $w(A_6) = bababb$, $w(A_7) = abbb$ and $w(A_8) = bababbabbb$.

Note that $w(A_{2k+1})$ is a word of the form "a" concatenated with k occurrences of character "b", while $w(A_{2k})$ is concatenations of strings of this form (with a "b" in the beginning). Hence $|w(A_{2k+1})| = k + 1$, and $|w(A_{2k})| = \sum_{i=1}^{k} i = \frac{(1+k)k}{2}$. It follows that the length $|w(A_n)|$ grows quadratically[3] with respect to n.

Remark 2.5. Before we start to prove the existence of some polynomial-time algorithms, notice that the following two operations take unit time:

(i) Given a data position, accessing the data in that position.

[3] Schleimer [11] gave many interesting examples of straight-line programs with lengths of words grows exponentially.

(ii) Checking if a non-terminal is sent to a terminal, i.e. whether it has height 1 (this is a boolean value, as a production can either consist of a string of non-terminal or a string of terminals). Similarly, later in Section 3, the time it takes to check if a character is a formal inverse takes a unit time.

In the algorithms that we will describe in this thesis, the number of operations in (i) and (ii) we require is insignificant compared to the number of other operations. Since we will be looking for a polynomial-time algorithms, if we can show that other operations take polynomial-time steps, then the resulting algorithm is again polynomial time and hence we will have achieved our goal. So unless an algorithm requires an excessive number of operations in (i) and (ii), we shall not mention the time required for these two operations.

Likewise, in our algorithms, we need to allocate some storage space before we store the actual value, while the time it takes to allocate storage space is less than the time it takes to store the actual value. Hence, the time required for the storage allocation will not be mentioned either.

To simplify our description, if the size of a variable has polynomial growth with respect to n, we will say that the size of that variable is (bounded by) $f(n)$ for some polynomial f.

The straight-line programs in normal forms are relativity easy to deal with; it turns out that we can transform an input straight-line program to normal form in polynomial-time:

Lemma 2.6. *Every straight-line program $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ can be placed in normal form in polynomial-time with respect to $\sum_i |W_i|$, by adding at most $f(\sum_i |W_i|)$ non-terminals for some polynomial f.*

Proof. [4] We assume that introducing (or removing) a production of the form $B \to a$ takes 2 time steps: one for B and one for a. Similarly, creating (or removing) the production of the form $B \to C \cdot D$ takes 3 unit times. The algorithm is divided into two steps.

(Step 1) For every $A_i \in \mathcal{A} \setminus \{A\}$, we check if it is sent to ϵ in the order of their indices. If $A_i \to \epsilon$, then remove the non-terminal A_i and all the appearances of A_i in the later productions (replace any empty productions by ϵ if required). For every $A_i \in \mathcal{A} \setminus \{A\}$, this takes at most $\sum_i |W_i|$ time steps, hence the time steps required in total for **(Step 1)** is at most $(|\mathcal{A}| - 1) \sum_i |W_i| \leq (\sum_i |W_i|)^2$, which is a polynomial in $\sum_i |W_i|$. After this, we can assume there are no non-terminals in $\mathcal{A}$ (except possibly the root) are sent to ϵ.

(Step 2, Case 1) Suppose the root A is sent to ϵ. Then we simply remove all other non-terminals and their production; the number of time steps required is at most

$$\sum_i |W_i| + |\mathcal{A}| \leq 2 \sum_i |W_i| = \mathcal{O}(\sum_i |W_i|)$$

which is a linear function in $\sum_i |W_i|$, and we are done.

[4] This result was stated by Schleimer [11], but not proven.

(Step 2, Case 2) Next, we consider the case when the root A is not sent to ϵ. In particular, by **(Step 1)**, we know that there are no non-terminals that are sent to ϵ. The proof is done by introducing some new non-terminals immediately before every $B \in \mathcal{A}$, so we can put the production rule for B in normal form. We will prove, by induction on the height $\|B\|$, that every non-terminal B with production length p will cause $\mathcal{O}(p)$ new non-terminals to be created using $\mathcal{O}(\sum_i |W_i|)$ time steps.

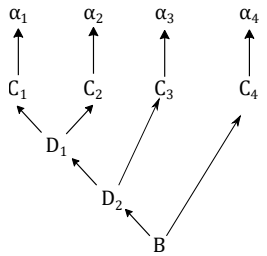


Figure 2: When the production consists of 4 terminals, we defined $4+2$ new non-terminals in order to put production of B in normal form.

Suppose $\|B\| = 1$, then $B \to \alpha_1 \cdot \alpha_2 \cdot \cdots \cdot \alpha_p$ for some $\alpha_i \in \mathcal{L}^*$. If $p = 1$, then the production is already in normal form. Suppose $p > 1$, define $C_1, C_2, \ldots C_p$ with productions $C_j \to \alpha_j$, for $1 \leq j \leq p$ (this takes $2p$ time steps). Next, we need to "combine" the non-terminals that we have created. Define D_1, D_2, ... D_{p-2}, with $D_1 \to C_1 \cdot C_2$ and $D_j \to D_{j-1} \cdot C_{j+1}$ for $2 \leq j \leq p-2$ (this takes at most $3p$ time steps). Lastly, let $B \to D_{p-2} \cdot C_p$ (we deleted p non-terminals in the production of B and replaced them by 2 non-terminals which takes $p+2$ time steps). We can deduce that we require $\mathcal{O}(p)$ time steps and introduced $\mathcal{O}(p)$ non-terminals overall (See Figure 2 for the production tree of B in the new program when the length of the production is equal to 4).

Suppose $\|B\| > 1$, then B has production $W = \beta_1 \cdot \beta_2 \cdots \beta_p$ with $\beta_i \in \mathcal{A}$. If $p = 1$, then replace W by the production of β_1 (deleting and replacing the production W takes at most 3 time steps). By induction hypothesis, β_1 has production in normal form, so we are done. If $p = 2$, W is already in normal form. If $p = 3$, we can use the same method as how we "combine" a set of non-terminals in $n = 1$ case. Hence, for $n > 1$ we also require $\mathcal{O}(p)$ time steps and introduced $\mathcal{O}(p)$ new non-terminals to put the production of B in normal form.

By induction, for each $B_i \in \mathcal{A}$ with production length $|W_i|$, we need $\mathcal{O}(|W_i|)$ steps to put W_i in normal form. So overall, **(Step 2, Case 2)** requires $\mathcal{O}(\sum_i |W_i|)$ time steps which is linear-time. Similarly, we introduced $\mathcal{O}(\sum_i |W_i|)$ new non-terminals, which is also a linear function in $\sum_i |W_i|$.

Finally, note that both **(Step 1)** and **(Step 2)** require polynomial time steps with respect to $\sum_i |W_i|$. Therefore, the whole algorithm is in polynomial time with respect to $\sum_i |W_i|$. □

Remark 2.7. Notice that we would potentially have many non-terminals producing the same words. A neater way to construct the new program would be: before constructing a new non-terminal, we check if it already exists in the set of non-terminals that we have created. This would reduce the total number of non-terminals in the new program.

Remark 2.8. Given an positive integer r and a property $\mathbf{P}$ of a word, suppose we want to prove the following:

> There exists an algorithm, call it $\mathfrak{A}$, that constructs a straight-line program $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ in normal form in polynomial-time with respect to r such that
>
> (i) the word w_A satisfies the property $\mathbf{P}$ and
>
> (ii) the number of non-terminals $|\mathcal{A}|$ is bounded by $f(r)$ for some polynomial f.

Then Lemma 2.6 tells us that it is sufficient to find an algorithm that constructs a straight-line program $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ in polynomial time with respect to r, with the length of every production bounded by $g(r)$ for some polynomial g such that:

(i) the word w_X satisfies the property $\mathbf{P}$ and

(ii) the number of non-terminals $|\mathcal{X}|$ is bounded by $f_0(r)$ for some polynomial f_0.

Then we argue as follows. Suppose there exists such an algorithm that constructs $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ as described above, call this algorithm $\mathfrak{A}'$. We will build algorithm $\mathfrak{A}$ by using $\mathfrak{A}'$: we first use $\mathfrak{A}'$ to construct such $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ that satisfies (i) and (ii) using $h(r)$ time steps for some polynomial h. Note that $g(r)f_0(r)$ is an upper bound for the sum of the lengths of productions. Then by Lemma 2.6, we can put $\mathbb{X}$ in normal from, call the normal form $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$, with $w_X = w_A$, using $h_1(g(r)f_0(r))$ time steps for some polynomial h_1. It is clear that w_A also satisfy property $\mathbf{P}$. Overall, this takes at most $h(r) + h_1(g(r)f_0(r))$ time steps which, as a sum of polynomials in r, is still a polynomial in r. By a similar argument, the number of non-terminals in $\mathbb{A}$ is also a polynomial function with respect to r and we completed the construction of algorithm $\mathfrak{A}$.

Remark 2.9. Note that we require at least 2 time steps to introduce a new non-terminal with its production. Hence, given an integer n, if an algorithm is in polynomial-time with respect to n, then in particular the number of non-terminals it created will be bounded by $f(n)$ for some polynomial f. Therefore in future results, if we need to prove that an algorithm creates at most $f(n)$ new non-terminals for some polynomial f, it is sufficient to prove that the algorithm is polynomial-time with respect to n.

Composition Systems

We will introduce more terminology in order to describe composition systems and compressed words.

Notation. For $w \in \mathcal{L}^*$, and $0 \le i \le j \le |w|$, define $w[i:j]$ to be the subword of w between $(i+1)^{th}$ character and j^{th} character. By convention, negative indices count from the end of w, i.e. $w[-j:-i] = w[|w|-j:|w|-i]$.

We will also use the following abbreviations.
Notation. For $0 \le i \le |w|$, we write $w[:i]$ for $w[0:i]$ which is the first i characters of w; we write $w[i:]$ for $w[i:|w|]$ which is the subword of w between $(i+1)^{th}$ character and the last character (informally, removing the first i characters of w). For $0 \le i \le |w|-1$, write $w[i]$ for $w[i:i+1]$ which is the $(i+1)^{th}$ character of w. Define $\mathrm{rev}(w)$ to be the *reverse* of w, i.e. $\mathrm{rev}(w)[i] = w[-i-1]$.

Thus $w[i:i] = \epsilon$ and $|w[i:j]| = j - i$. In particular, $w[-i] = w[|w|-i]$ for $1 \le i \le |w|$ and $\mathrm{rev}(w)[0] = w[-1]$ is the last character of w.
Example 2.10. Let $w = abcde$, then we have $w[1:-1] = w[1:|w|-1] = w[1:4] = bcd$ has length $4-1=3$, $w[:2] = ab, w[3:] = de$, $w[0] = a$ and $\mathrm{rev}(w) = edcba$.

Next, we will introduce a generalization of the straight-line program, it will allow truncations in the productions which consist of two non-terminals. Suppose $\mathcal{A}$ is a set of non-terminals of a program. For $A \in \mathcal{A}$ and $0 \le i \le j \le |w_A|$, we call $A[i,j]$ a *truncated non-terminal*. We denote by $\mathcal{T}_{\mathcal{A}}$ to be the set of all the truncated non-terminals in $\mathcal{A}$, i.e we have that $\mathcal{T}_{\mathcal{A}} = \{A[i,j] \big| A \in \mathcal{A} \text{ and } 0 \le i \le j \le |w_A|\}$.
Definition 2.11. A *composition system* $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A_r, \mathcal{P} \rangle$ contains the following:

(i) A finite alphabet of terminal characters $\mathcal{L} = \{a_1, a_2, \ldots, a_m\}$.

(ii) A disjoint finite alphabet of non-terminal characters $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$.

(iii) A root non-terminal $A_r \in \mathcal{A}$.

(iv) A set of *production rules* $\mathcal{P} = \{A_i \to W_i\}_{i=1}^n$. The production rule of the form $A_i \to W_i$ allows us to replace a non-terminal A_i with its *production* in the form of: a word W_i in $\mathcal{L}^* \cup \mathcal{T}_{\mathcal{A}}^*$. Every non-terminal A_j appearing in W_i has index $j < i$.

Truncated non-terminals only appear on the right hand side of a production rule, never on the left. Repeated truncation behaves as expected: $(B[i:j])[k:l] = B[i+k:i+l]$ and in particular $(B[i:j])[k] = B[i+k]$. Similar to the notations earlier, we shall define $w(B[i:j]) = w_B[i:j]$.
Example 2.12. The following composition system is obtained by adding some truncations to the straight line program in Example 2.4:

$$\mathbb{A} = \left\langle \begin{array}{l} \{a,b\}, \{A_i\}_{i=1}^{2m}, A_{2m}, \{A_1 \to a, A_2 \to b\} \cup \\ \left\{A_{2k-1} \to A_{2k-3} \cdot A_2, A_{2k} \to A_{2k-2} \cdot A_{2k-1}[:2]\right\}_{k=2}^{m} \end{array} \right\rangle.$$

After the truncation, the word $w(A_{2k})$ consists a character "b" and $k-1$ occurrences of the pair "ab". For example $w(A_8) = bababab$.

Remark 2.13. The notion of *normal form* in composition systems is defined analogously to the definition of normal form in straight-line programs but allowing truncated non-terminals in the productions. Hence, the composition system in Example 2.12 is in normal form.

It turns out that we can bound the length on the word produced by a non-terminal in terms of its height, a useful result that we will use later to prove that our further algorithms are indeed polynomial-time.

Lemma 2.14. *If $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ is a composition system in normal form, then we have $|w_B| \leq 2^{\|B\|}$, for all $B \in \mathcal{A}$.*

Proof. [5] The proof is done by induction on $n = \|B\|$, the height of B. If $n = 1$, then $B = a$ for some $a \in \mathcal{L}$. We have $|w_B| = 1 \leq 2 = 2^{\|B\|}$.

Suppose that $\|B\| = n > 1$ and for all $C \in \mathcal{A}$ with $\|C\| \leq n - 1$, we have $|w_C| \leq 2^{\|C\|}$. Since $\mathbb{A}$ is in normal form, B has the production of the form $B \to C[i : j] \cdot D[k : l]$ for some C, $D \in \mathcal{A}$, with $\|C\| \leq n - 1$, $\|D\| \leq n - 1$. Then we have

$$\begin{aligned} |w_B| &= |w_C[i : j]| + |w_D[k : l]| \\ &\leq |w_C| + |w_D| \\ &\leq 2^{\|C\|} + 2^{\|D\|} \\ &\leq 2^{n-1} + 2^{n-1} = 2^n = 2^{\|B\|}. \end{aligned}$$

Hence, by induction, for all $B \in \mathcal{A}$, $|w_B| \leq 2^{\|B\|}$. □

Before we start to introduce some algorithms that are in polynomial-time with respect to $|\mathcal{A}|$, we will first see that some maths operations involving the length of words take polynomial-time with respect to $|\mathcal{A}|$.

Lemma 2.15. *Let m and n be two non-negative integers. In a computer system, the time it takes to store m is $\mathcal{O}(\log(m))$, while the time it takes to compute the addition $m + n$ is $\mathcal{O}(\log(\max\{m, n\}))$.* □

The proof is elementary and lengthy, hence it is included in the Appendix (Section 9.11).

Remark 2.16. In fact, since computers use one binary digit to represent the sign of a number, by using a similar proof we can deduce that for any two arbitrary integers m and n, the time it takes to

(i) compute the sum $m + n$ is $\mathcal{O}(\log(\max\{m, n\}))$;

(ii) compare m and n is $\mathcal{O}(\log(\max\{m, n\}))$;

(iii) Store the value m is $\mathcal{O}(\log(m))$.

[5] This result was stated by Schleimer [11], but not proven.

Next, we apply the above observation to our systems.

Corollary 2.17. *Given a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ with two non-terminals $B, C \in \mathcal{A}$, the following operations take $\mathcal{O}(|\mathcal{A}|)$ time steps:*

(i) computing the sum of $|w_B| + |w_C|$ (similar for subtraction);

(ii) comparing $|w_B|$ and $|w_C|$;

(iii) store the value $|w_B|$.

Proof. Let $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ be a composition system with $B, C \in \mathcal{A}$. By Lemma 2.14, we have $|w_B| \leq 2^{\|B\|}$. Hence $\log_2 |w_B| \leq \|B\| \leq |\mathcal{A}|$, with the last inequality follows from the fact that the height of any non-terminal can not exceed the number of the non-terminals. Similarly, $\log_2(|w_C|) \leq |\mathcal{A}|$. Hence, by using Lemma 2.15, computing the sum of $|w_B| + |w_C|$ takes $\mathcal{O}(|\mathcal{A}|)$ time steps. The other results can be proven similarly. □

Remark 2.18. In the further proofs, if we are given two non-negative numbers, i and j, such that $i \leq |w_B|$ and $j \leq |w_C|$ for some $B, C \in \mathcal{A}$, then the addition, subtraction and comparing i and j takes $\mathcal{O}(|\mathcal{A}|)$ time steps. Similarly, multiplying and dividing i by a fixed constant takes $\mathcal{O}(|\mathcal{A}|)$ time steps. Furthermore, if i is stored, replacing i by j takes $\mathcal{O}(|\mathcal{A}|)$ time (since both deleting the old value and storing the new value take $\mathcal{O}(|\mathcal{A}|)$ time steps).

Example 2.19. In particular, constructing a new non-terminal A with the production $A \to B[i:j] \cdot C[k:l]$ takes $\mathcal{O}(|\mathcal{A}|)$ time steps, since the operations we did were storing 3 non-terminals (A, B and C) and 4 integers.

Remark 2.20. By adapting a similar approach as Lemma 2.6 where we put a straight-line program in normal form, we can transform a composition system into normal form in polynomial time with respect to the number of non-terminals (we need to store at most 4 integers when we construct a new non-terminal but this takes polynomial-time). From now on, we will assume that all composition systems considered in this thesis will be in normal form.

Remark 2.21. Let $\phi : \mathcal{L}^* \to \mathcal{L}^*$ be an operation that acts on the words in $\mathcal{L}^*$. In the contrary to Remark 2.9, suppose we want to prove the following:

> There exists an algorithm, call it $\mathfrak{A}$, which takes a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ as the input and constructs a new system $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ in polynomial-time with respect to $|\mathcal{A}|$, such that $w_X = \phi(w_A)$.

We have the following approach. First we make a copy of $\mathcal{L}$ in $\mathbb{X}$, it is clear[6] that storing them takes $\mathcal{O}(|\mathcal{A}|)$ time steps (this process is often omitted in the proof). Next for every $B \in \mathcal{A}$, we construct a list of non-terminals in $\mathbb{X}$, such that the last non-terminal in the list, called it Y, satisfies $w_Y = \phi(w_B)$. In particular, the root of $\mathbb{X}$, which we denote by X, will be the last non-terminal constructed in $\mathcal{X}$ and satisfy $w_X = \phi(w_A)$.

[6]Note that the number of terminals required in $\mathcal{X}$ is bounded by $|\mathcal{A}|$.

In order to compute the time complexity, we prove by induction that for every $B \in \mathcal{A}$, we require $f_B(|\mathcal{A}|)$ non-terminals to be constructed in $\mathbb{X}$, for some polynomial f_B. Hence the total number of non-terminals we constructed is $f(|\mathcal{A}|) = \sum_{B\in\mathcal{A}} f_B(|\mathcal{A}|)$ which, as a finite sum of polynomials in $|\mathcal{A}|$, is again a polynomial in $|\mathcal{A}|$. As shown in Example 2.19, every production takes $\mathcal{O}(|\mathcal{A}|)$ time steps, we can deduce that the total number of the time steps required to construct all the non-terminals in $\mathbb{X}$ is also a polynomial in $|\mathcal{A}|$, which proves the result.

The following lemma allows us to assume that the input composition system has no negative indices and abbreviated truncations.

Lemma 2.22. *Given a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P}\rangle$, there is a polynomial-time algorithm with respect to $|\mathcal{A}|$ which*

(i) stores $|w_B|$ for all $B \in \mathcal{A}$; and

(ii) amends the production rules in $\mathcal{P}$, so that every production is complete (i.e it does not contain abbreviated truncations or negative indices).

Proof. [7]First, we define an array of size $|\mathcal{A}|$ which will store $|w_B|$ for all $B \in \mathcal{A}$. We will prove, by induction on the height $n = \|B\|$, that for every $B \in \mathcal{A}$, the operations described in (i) and (ii) can be done in polynomial-time with respect to $|\mathcal{A}|$.

If $n = 1$, then $B \to a$ for some $a \in \mathcal{L}$ and so $|w_B| = 1$ (it takes a unit time step to check if B has height 1 and $\mathcal{O}(|\mathcal{A}|)$ time to store $|w_B|$).

Suppose that $\|B\| = n > 1$. If B has abbreviated productions we replace the missing value by the corresponding number. For example if we have $B \to C[i :] \cdot D[: l]$, replace it by $B \to C[i : |w_C|] \cdot D[0, l]$. (Since $\|C\| \leq n - 1$, $|w_C|$ is computed by induction hypothesis). This involves storing at most 4 integers, all of the them are less or equal to $|w_B|$, hence this process takes $\mathcal{O}(|\mathcal{A}|)$ time.

Next, if B has negative values in its production, replace them by the corresponding non-negative number. For example, we replace $C[i : -1]$ by $C[i : |w_C| - 1]$. Note that checking the sign of a number takes a unit time. At most this involves checking the sign of 4 numbers, computing 4 subtractions and storing 4 numbers, which again takes $\mathcal{O}(|\mathcal{A}|)$ time.

Finally, B will now have the production of the form $B \to C[i : j] \cdot D[k : l]$ with integers $0 \leq i \leq j \leq |w_C|$ and $0 \leq k \leq l \leq |w_D|$. Then we can compute $|w_B| = j - i + l - k$. Similar to above, computing and storing $|w_B|$ takes $\mathcal{O}(|\mathcal{A}|)$ time, which completes the proof of the induction statement.

Therefore, since there are $|\mathcal{A}|$ non-terminals, the total number of time steps required to perform operations in (i) and (ii) is $\mathcal{O}(|\mathcal{A}|^2)$, which is a polynomial with respect to $|\mathcal{A}|$. □

[7]This result was stated by Schleimer [11], but not proven.

Remark 2.23. Suppose we want to prove the following:

> There exists an algorithm, call it $\mathfrak{A}$, which takes a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ as the input, and outputs a property of the word produced by the root non-terminal, called it $\mathbf{P}(w_A)$, in polynomial-time with respect to $|\mathcal{A}|$.

Then Lemma 2.22 tells us that it is sufficient to show that such an algorithm exists for the composition systems with complete productions, and then we argue as follows:

Suppose such an algorithm exists for the composition system with complete productions, call this algorithm $\mathfrak{A}'$. We will construct algorithm $\mathfrak{A}$ as follows: given a general composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$, Lemma 2.22 tells us that we can amend it to a composition system $\mathbb{X} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{Q} \rangle$, with all productions being complete, using $f(|\mathcal{A}|)$ time steps for some polynomial f. Then apply $\mathfrak{A}'$ on $\mathbb{X}$, this takes $g(|\mathcal{A}|)$ time steps for some polynomial g by assumption, and it outputs $\mathbf{P}(w_A)$. This is exactly what we want $\mathfrak{A}$ to output. Overall, this takes at most $p(|\mathcal{A}|) + q(|\mathcal{A}|)$ time steps which, as a sum of polynomials in $|\mathcal{A}|$, is still a polynomial in $|\mathcal{A}|$. This completes the description of algorithm $\mathfrak{A}$.

There are some informations about the word w_A we can obtain without running the system $\mathbb{A}$ (which will compute the word explicitly).

Lemma 2.24. *Given a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ and an integer r such that $0 \leq r \leq |w_A| - 1$, there is a polynomial-time algorithm with respect to $|\mathcal{A}|$ which computes the character $w_A[r]$, the $(r+1)^{th}$ character of w_A.*

Proof. [8] We will consider a tuple in the form (B, i). Define $(B, i) \sim (C, j)$ if $w_B[i] = w_C[j]$. We begin with the tuple (A, r) (storing this takes $\mathcal{O}(|\mathcal{A}|)$ time).

If $\|A\| = 1$, then $A \to a$ for some $a \in \mathcal{L}$, so $r = 0$ with $w_A[0] = a$ and we are done (given a data position time, access the data at that position takes unit time).

If $\|A\| > 1$ and A has the production rule of the form $A = B[i : j] \cdot C[k, l]$ for some B, $C \in \mathcal{A}$. Let $l_B = j - i$. If $r \leq l_B - 1$, then change the tuple to $(B, r + i)$ then we have $(A, r) \sim (B, r + i)$; otherwise change it to $(C, r - l_B + k)$ and we still have $(A, r) \sim (C, r - l_B + k)$ (This process involves at most one comparison, three additions and subtractions, hence it takes $\mathcal{O}(|\mathcal{A}|)$ time). Now the non-terminal in the first entry will have height that is strictly less than A. Repeat this process until the first entry has height 1 (See Figure 3 for the example when $A = B[1 : 4] \cdot C[2, 5]$ and w_C has length 6).

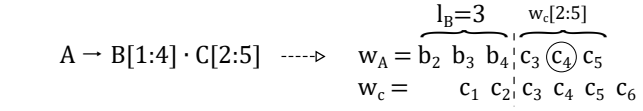


Figure 3: If $A = B[1 : 4] \cdot C[2, 5]$, then $(A, 4) \sim (C, 4 - 3 + 2) = (C, 3)$.

[8] This result was stated by Schleimer [11], but not proven. He used this to prove that the the word problem for automorphisms of free group is soluble in polynomial-time.

Since $\|A\| \leq |\mathcal{A}|$ and the height of the non-terminal in the first entry is reduced by at least 1 after every round, we need to repeat the process for at most $|\mathcal{A}|$ rounds. Each round we require $\mathcal{O}(|\mathcal{A}|)$ time steps, so in total it requires $\mathcal{O}(|\mathcal{A}|^2)$ time steps, which is polynomial-time with respect to $|\mathcal{A}|$. □

It is remarkable that Hagenah [4] discovered a polynomial-time algorithm which transforms a composition system to a straight-line program.

Theorem 5.1. *Given a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$, there is a polynomial-time algorithm with respect to $|\mathcal{A}|$, that finds a straight-line program $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ with $w_X = w_A$ and $|\mathcal{X}|$ is bounded by $f(|\mathcal{A}|)$ for some polynomial f.*

In order to make this thesis more self-contained, the algorithm and the proof will be presented in Section 5. We will also need the following result from Plandowski [10].

Theorem 6.1. *Given two straight-line programs $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ and $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ in normal forms, there is a polynomial-time algorithm with respect to $(|\mathcal{A}| + |\mathcal{X}|)$, that decides whether or not $w_X = w_A$.*

The proof of this will be presented in Section 6. Combining these two results, Schleimer [11] presented the following theorem by Gasieniec, Karpinski, Plandowski, and Rytter [3].

Theorem 2.25. *Given two composition systems $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ and $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$, there is a polynomial-time algorithm with respect to $(|\mathcal{A}| + |\mathcal{X}|)$, that computes the largest integer $k \geq 0$ such that $w_A[:k] = w_X[:k]$*

Proof. [9] First note that adding a production of the form $A \to B[i:j] \cdot C[k:l]$ in $\mathbb{A}$ or $\mathbb{X}$ takes $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$ time steps (we store 3 non-terminals and 4 integers). Similarly, amending a production rule in $\mathbb{A}$ or $\mathbb{X}$ takes $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$ time steps.

We first define the *prefix checker* algorithm which takes an integer i as the input and checks whether the first i characters of $\mathbb{A}$ and $\mathbb{X}$ are the same. Given an integer i with $0 \leq i \leq \min\{|w_A|, |w_X|\}$, we define[10]

$$\mathbb{A}^i = \langle \mathcal{L}, \mathcal{A} \cup \{A^i\}, A^i, \mathcal{P} \cup \{A^i \to A[0:i] \cdot A[0:0]\} \rangle$$
$$\mathbb{X}^i = \langle \mathcal{L}, \mathcal{X} \cup \{X^i\}, X^i, \mathcal{Q} \cup \{X^i \to X[0:i] \cdot X[0:0]\} \rangle.$$

Then, $\mathbb{A}^i$ is the composition system $\mathbb{A}$ with a new root non-terminal (similar for $\mathbb{X}^i$). As we discussed earlier, creating or amending the productions for the root non-terminals takes $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$ time steps. Clearly, the length of two roots are equal, i.e. $|w(A^i)| = |w(X^i)|$. By Theorem 5.1, there is polynomial-time algorithm[11] with respect to $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$, which

[9] This proof expands in more detail the proof outlined by Schliemer [11].

[10] Note that the superscripts here are used to show that the root non-terminals depend on i and should not be confused them with exponents. Also the truncations $A[0:0]$ and $X[0:0]$ are needed to make sure the productions are in normal form.

[11] The additional 2 non-terminals can change the run time by a constant, but we have $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}| + 2) = \mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$.

finds two straight-line programs $\widetilde{\mathbb{A}^i}$ and $\widetilde{\mathbb{X}^i}$ with roots $\widetilde{A^i}$ and $\widetilde{X^i}$ that satisfy $w(\widetilde{A^i}) = w(A^i)$ and $w(\widetilde{X^i}) = w(X^i)$ respectively. Also, $\widetilde{\mathbb{A}^i}$ and $\widetilde{\mathbb{X}^i}$ will have $h(|\mathcal{A}| + |\mathcal{X}|)$ non-terminals together for some polynomial function h. Hence, by Theorem 6.1, we can check whether $w(\widetilde{X^i}) = w(\widetilde{A^i})$ using $f(h(|\mathcal{A}| + |\mathcal{X}|))$ time steps for some polynomial f. This finishes the definition of prefix checkers. Since every step we described is in polynomial time with respect to $(|\mathcal{A}| + |\mathcal{X}|)$ and we have finitely many steps, running the prefix checker once takes polynomial-time with respect to $(|\mathcal{A}| + |\mathcal{X}|)$.

Now we will combine the prefix checker with the binary search algorithm. We first let $j_0 = \min\{|w_A|, |w_X|\}$. If the prefix checker accepts j_0 then take $k = j_0$ and we are done. Otherwise, let $i_0 = 0$ and start with the tuple $\{i_0, j_0\}$ (comparing two integers and storing two integers takes $\mathcal{O}(|\mathcal{A}| + |\mathcal{X}|)$ time). Proceed as follows, every round, we are given numbers $\{i_n, j_n\}$ with

(i) $0 \leq i_0 \leq i_n < j_n \leq j_0$ and

(ii) the truncations $w_A[: i_n] = w_X[: i_n]$ and $w_A[: j_n] \neq w_X[: j_n]$, i.e. the first i_n characters of w_A and w_X are the same, but the first j_n characters of w_A and w_X are different.

If $i_n + 1 = j_n$ then set $k = i_n$ and we are done. Otherwise, let i' be greatest integer less than $(i_n + j_n)/2$. We run the prefix checker with input i'. If i' is accepted then let $i_{n+1} = i'$ and let $j_{n+1} = j_n$. If i' is rejected then let $i_{n+1} = i_n$ and let $j_{n+1} = i'$.

From Remark 2.18 and the definition of prefix checker, for all n, the process changing $\{i_n, j_n\}$ to $\{i_{n+1}, j_{n+1}\}$ takes $g(|\mathcal{A}| + |\mathcal{X}|)$ time steps for some polynomial g (because it involves comparing two integers, adding two integers, diving the sum by 2, running prefix checker once and replacing two integers, which all takes polynomial time with respect to $(|\mathcal{A}| + |\mathcal{X}|)$).

Finally, because $j_{n+1} - i_{n+1} \leq (j_n - i_n)/2 + 1$, we modified the tuple at most $\log_2(j_0)$ times. By Lemma 2.14, $\log_2(j_0) \leq (|\mathcal{A}| + |\mathcal{X}|)$, hence the tuple is changed at most $(|\mathcal{A}| + |\mathcal{X}|)$ times. Changing it once requires $g(|\mathcal{A}| + |\mathcal{X}|)$ time steps, therefore the number of time steps required to change it $(|\mathcal{A}| + |\mathcal{X}|)$ times is $(|A| + |X|)g(|\mathcal{A}| + |\mathcal{X}|)$ which, as a product of polynomials in $(|\mathcal{A}| + |\mathcal{X}|)$, is still a polynomial in $(|\mathcal{A}| + |\mathcal{X}|)$. □

3 Lohrey's Algorithm

The aim of this section is to adapt our algorithms to solve the word problem in the finitely generated free group. We first extend the set of terminals $\mathcal{L}$ to $\mathcal{L}_m = \{a_i, \overline{a_i}\}_{i=1}^m$. Let $\bar{\cdot} : \mathcal{L}_m \to \mathcal{L}_m$ be the involution swapping a and $\overline{a}$. We will later use $\mathcal{L}_m$ to represent the set of generators of a group where $\overline{a}$ will represent the inverse of the generator a. Given a word $w(a_i) \in \mathcal{L}_m$ define $\overline{w}$ to be $\mathrm{rev}(w(a_i))$, i.e. $\overline{w}[r] = \overline{rev(w)[r]}$ (Informally, we reverse the order of the characters and then change every character to its inverse).

The following lemma tells us how involution swapping interacts with truncation and product operations.

Lemma 3.1. *Suppose w, u and v are words, then*

(i) $\overline{w[i:j]} = \overline{w}[-j,-i]$ *where* $0 \le i \le j \le |w|$;

(ii) $\overline{u \cdot v} = \overline{v} \cdot \overline{u}$.

Proof. The proof is straightforward by using repeated truncations, definitions of reverse and involution swapping. We will examine every single character in the word:

(i) Let $0 \le r \le j - r - 1$, then

$$\begin{aligned}
\overline{w[i:j]}[r] &= \overline{\mathrm{rev}(w[i:j])[r]} && \text{by definition of involution swapping} \\
&= \overline{(w[i:j])[-r-1]} && \text{by definition of reverse} \\
&= \overline{w[i:j][j-i-r-1]} && \text{since } |w[i:j]| = j-i \\
&= \overline{w[j-r-1]} && \text{by repeated truncations} \\
&= \overline{\mathrm{rev}(w)[r-j]} && \text{by definition of reverse} \\
&= \overline{w}[r-j] && \text{by definition of involution swapping} \\
&= \overline{w}[-j,-i][r] && \text{by repeated truncations.}
\end{aligned}$$

(ii) Let $0 \le r \le |u| + |v| - 1$. Suppose first that $0 \le r \le |v| - 1$, then

$$\begin{aligned}
\overline{u \cdot v}[r] &= \overline{\mathrm{rev}(u \cdot v)[r]} && \text{by definition of involution swapping} \\
&= \overline{(u \cdot v)[-r-1]} && \text{by definition of reverse} \\
&= \overline{(u \cdot v)[|u| + |v| - r - 1]} && \text{since } |u \cdot v| = |u| + |v| \\
&= \overline{v[|v| - r - 1]} && \text{since we assumed that } 0 \le r \le |v| - 1 \\
&= \overline{\mathrm{rev}(v)[r]} && \text{by definition of reverse} \\
&= \overline{v}[r] && \text{by definition of involution swapping} \\
&= \overline{v} \cdot \overline{u}[r] && \text{since we assumed that } 0 \le r \le |v| - 1.
\end{aligned}$$

The case when $|v| \le r \le |u| + |v| - 1$ is similar. □

Combining (i) and (ii) from above, we obtain the following:

Remark 3.2. Using the same notation as the above lemma, we have

$$\overline{u[i:j] \cdot v[k:l]} = \overline{v[k:l]} \cdot \overline{u[i:j]} = \overline{v}[-l:-k] \cdot \overline{u}[-j:-i].$$

Given a straight-line program or a composition system $\mathbb{A}$, the involution swapping given in the first paragraph can be extended to the non-terminals in $\mathcal{A}$. We define it inductively

on the height of $A \in \mathcal{A}$, if $A \to a$ then $\overline{A} \to \overline{a}$. Otherwise $A \to B[i:j] \cdot C[k:l]$ and we have $\overline{A} \to \overline{C}[-l:-k] \cdot \overline{B}[-j:-i]$. We extend the set of non-terminals $\mathcal{A} = \{A_i, \overline{A_i}\}_{i=1}^{m}$ and will introduce a structure which allows us to have involution wrappings in the productions.

Definition 3.3. A *compressed word*[12] is a straight-line program or composition system which allow involution swapping, i.e. characters from $\mathcal{A} \cup \mathcal{L}_m$, on the right hand side of productions. Its root produces a word in $\mathcal{L}_m$.

Note that for every $\alpha \in \mathcal{A} \cup \mathcal{L}_m$ we have $\|\alpha\| = \|\overline{\alpha}\|$.

Example 3.4. Consider the following straight-line program,

$$\mathbb{A} = \left\langle \begin{array}{l} \mathcal{L}_2 = \{a, b, \overline{a}, \overline{b}\}, \{A_i\}_{i=1}^{n}, A_n, \{A_i \to A_{i-1} \cdot A_4\}_{i=5}^{n} \cup \\ \{A_1 \to a, A_2 \to b, A_3 \to A_1 \cdot A_2, A_4 \to A_3 \cdot \overline{A_1}\} \end{array} \right\rangle.$$

We can see that for $n \geq 4$, $w(A_n) = (ab\overline{a})^{n-3}$. For example,

$$w(A_8) = ab\overline{a}ab\overline{a}ab\overline{a}ab\overline{a}ab\overline{a}.$$

Recall that a word is *freely reduced* if it is the empty word or it does not contain the subword of the form $\alpha\overline{\alpha}$, for any $\alpha \in \mathcal{L}_m$. For example, the free reduction of $w(A_8)$ is $ab^5\overline{a}$.

The following lemma allows us to compute the inverse of the root.

Lemma 3.5. *Given a compressed word $\mathbb{A} = \langle \mathcal{L}_m, \mathcal{A}, A_n, \mathcal{P} \rangle$, there exists a polynomial-time algorithm with respect to $|\mathcal{A}|$, that computes a new compressed word $\mathbb{X} = \langle \mathcal{L}_m, \mathcal{X}, X_n, \mathcal{Q} \rangle$ with $w(X_n) = \overline{w(A_n)}$.*

Proof. [13]We will prove the result for the composition system, the proof for straight-line program is similar. For every α in $\mathcal{A} \cup \mathcal{L}_m$, we will assume that storing the character $\overline{\alpha}$ takes 2 time steps, one for the bar $\bar{\cdot}$ and one for α.

Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ with A_n being the root of $\mathbb{A}$. We will use a similar idea from Remark 2.21. For each $A_r \in \mathcal{A}$, we construct a X_r in $\mathcal{X}$ such that $w(X_r) = \overline{w(A_r)}$. In particular, $w(X_n) = \overline{w(A_n)}$. We will prove by induction on $\|A_r\|$, that constructing the corresponding X_r takes $\mathcal{O}(|\mathcal{A}|)$ time steps for every $A_r \in \mathcal{A}$.

If $\|A_r\| = 1$, then $A_r \to \beta$ for some $\beta \in \mathcal{L}_m$, we define $X_r \to \overline{\beta}$ (This takes 3 time steps). It is clear that $w(X_r) = \overline{w(A_r)}$.

Suppose now that $\|A_r\| > 1$, then A_r has the production of the form $A_r \to \alpha_p[i:j] \cdot \alpha_q[k:l]$ for some $\alpha_p \in \{A_p, \overline{A_p}\}$ and $\alpha_q \in \{A_q, \overline{A_q}\}$, with $\|\alpha_p\| \leq \|A_r\| - 1$ and $\|\alpha_q\| \leq \|A_r\| - 1$. By induction hypothesis, there exists $\chi_p \in \{X_p, \overline{X_p}\}$, $\chi_q \in \{X_q, \overline{X_q}\}$ in $\mathcal{X}$ such that

[12]Sometimes we say *compressed straight-line program* or *compressed composition system* to emphasis that the program or system contains inverses in the productions.

[13]This result was stated by Schleimer [11], but not proven.

$w(\chi_p) = \overline{w(\alpha_p)}$ and $w(\chi_q) = \overline{w(\alpha_q)}$. Define $X_r \to \chi_q[-l : -k] \cdot \chi_p[-j : -i]$ (storing 3 non-terminals and 4 integers takes $\mathcal{O}(|\mathcal{A}|)$ time steps). By Remark 3.2, we have

$$\begin{aligned}\overline{w(A_r)} &= \overline{w(\alpha_p)[i:j] \cdot w(\alpha_q)[k:l]} \\ &= \overline{w(\alpha_q)}[-l:-k] \cdot \overline{w(\alpha_p)}[-j:-i] \\ &= w(\chi_q)[-l:-k] \cdot w(\chi_p)[-j:-i] = w(X_r).\end{aligned}$$

Therefore we proved, by induction, constructing every X_r takes $\mathcal{O}(|\mathcal{A}|)$ time steps. Since we need to construct $|\mathcal{A}|$ non-terminals, the total time steps required to construct $\mathbb{X}$ is $\mathcal{O}(|\mathcal{A}|^2)$, a polynomial function in $|\mathcal{A}|$. □

Remark 3.6. There is an alternative way to prove the above lemma for composition system $\mathbb{A}$. First we prove the lemma for straight-line programs. Then given a composition system $\mathbb{A}$, apply generalised Theorem 5.1 (i.e. allowing the involution swapping in the input productions) and then use a similar idea from Remark 5.3 to prove the result for composition systems.

The following theorem by Lohrey [7] is the key tool to prove the existence of polynomial-time algorithm that solves the compressed word problem in free group; the proof given by Schleimer [11] is very clear, hence we will not duplicate it here.

Theorem 3.7. *Given a straight-line program $\mathbb{A} = \langle \mathcal{L}_m, \mathcal{A}, A_n, \mathcal{P} \rangle$ in normal form, There is a polynomial-time algorithm with respect to $|\mathcal{A}|$ which finds a composition system $\mathbb{X} = \langle \mathcal{L}_m, \mathcal{X}, X_n, \mathcal{Q} \rangle$ where w_X is the free reduction of w_A.* □

Example 3.8. Apply Theorem 3.7 to Example 3.4, we would obtain the following:

$$\mathbb{X} = \left\langle \begin{array}{l} \mathcal{L}_2 = \{a, b, \overline{a}, \overline{b}\}, \{X_i\}_{i=1}^n, X_n, \left\{X_i \to X_{i-1}[:-1] \cdot X_4[1:]\right\}_{i=5}^n \cup \\ \{X_1 \to a, X_2 \to b, X_3 \to X_1 \cdot X_2, X_4 \to X_3 \cdot \overline{X_1}\} \end{array} \right\rangle.$$

Indeed we have $w(X_8) = ab^5\overline{a}$.

Here is the remarkable result by Lohrey [7].

Theorem 3.9. *(Lohrey) The word problem for compressed words in the free group is soluble in polynomial-time.*

Proof. [14] Here we view the terminals as the generators of the group and we want to check whether the root non-terminal produces the identity element. According to Remark 5.3, it is sufficient to prove for the case that the input is a straight-line program. Given a straight-line program $\mathbb{A} = \langle \mathcal{L}_m, \mathcal{A}, A_n, \mathcal{P} \rangle$ with $|\mathcal{A}| = n$, we can apply Theorem 3.7 to find a composition system $\mathbb{X} = \langle \mathcal{L}_m, \mathcal{X}, X_n, \mathcal{Q} \rangle$, where w_{X_n} is the free reduction of w_{A_n}, which takes $f(n)$ time steps for some polynomial f. Then, by Lemma 2.22, we can compute $|w_{X_n}|$ using $h(n)$ time steps for some polynomial h. Finally, it takes $g(n)$ time steps to

[14] This result was stated in Schleimer [11], but not proven.

check if $|w_{X_n}|$ equals to 0 for some polynomial g. Note that w_{A_n} is the identity element if and only if its free reduction w_{X_n} has length zero. In total, the number of time steps we require is $f(n) + h(n) + g(n)$ which, as a sum of polynomials in n, is still a polynomial in n. □

4 Word Problem in Free-by-Cyclic Group

The background of group theory that we will need for this section can be found in Lyndon and Schupps book [9]. Recall the definition of $\mathrm{Aut}(F_m)$: the group of all automorphisms of the free group F_m. We will be looking at the following group:

Definition 4.1. Fix $\Phi \in \mathrm{Aut}(F_m)$. Then the *free-by-cyclic* group G_Φ is presented by:

$$\langle a_i, t | t a_i \bar{t} = \Phi(a_i), i \in \{1, \ldots, m\} \rangle$$

Here, we use $\bar{t}$ to represent the group generator t^{-1}.

We will denote by $\mathbf{1}_G$ to be the identity element in G_Φ. We will let $\mathcal{L}_m = \{a_i, \overline{a_i}\}_{i=1}^m$ as before, then $\mathcal{L}_m \cup \{t, \bar{t}\}$ is the set of generators in G_Φ. The following result is needed for Schleimer's algorithm that solves the word problem in free-by-cyclic group:

Lemma 4.2. *Given a word $\bar{t}^k w t^l$ where $w \in \mathcal{L}_m^*$. Then $\bar{t}^k w t^l = \mathbf{1}_G$ if and only if $k = l$ and $w = \mathbf{1}_G$.*

Proof. Consider the homomorphism that adds up the exponents of t in the words from $(\mathcal{L}_m \cup \{t, \bar{t}\})^*$ defined by

$$\begin{aligned} \rho : G_\Phi &\to (\mathbb{Z}, +) \\ a_i &\mapsto 0 \quad \text{for } 0 \le i \le m, \\ t &\mapsto 1. \end{aligned}$$

It follows that $\rho(\mathbf{1}_G) = 0$.

($\Leftarrow$) We have $\rho(\bar{t}^k w t^l) = -k + l$. If $-k + l \neq 0$, then the word $\bar{t}^k w t^l \neq \mathbf{1}_G$.

($\Rightarrow$) On the other hand, if $k = l$, then

$$\mathbf{1}_G = \bar{t}^k \mathbf{1}_G t^l = \bar{t}^k w t^l \iff \mathbf{1}_G = w.$$ □

By using Lohrey's theorem, Schleimer [11] deduced the following:

Theorem 4.3. *The word problem for G_Φ is soluble in polynomial-time (with respect to the length of the word).*

Our implementation follows exactly the steps described in Schleimer's paper [11]. So we will only give a sketch of the proof here:

Proof sketch of Theorem 4.3. For simplicity of notation, we denote Φ^0 as the identity map. From the presentation of the G_Φ, We can deduce, by some computation, that for all $p \geq 0$,

$$t \cdot \Phi^p(a_i) = \Phi^{p+1}(a_i) \cdot t, \qquad \Phi^p(a_i) \cdot \overline{t} = \overline{t} \cdot \Phi^{p+1}(a_i),$$
$$t \cdot \Phi^p(a_i^{-1}) = \Phi^{p+1}(a_i^{-1}) \cdot t, \qquad \Phi^p(a_i^{-1}) \cdot \overline{t} = \overline{t} \cdot \Phi^{p+1}(a_i^{-1}).$$

We first construct a straight-line program with a set of non-terminals $\{A_{i,p} | 1 \leq i \leq m\}$ such that $w(A_{i,p}) = \Phi^p(a_i)$. Next given a word W in $(\mathcal{L}_m \cup \{t, \overline{t}\})^*$, by using the equations above, we can move all the occurrences of t to the right and $\overline{t}$ to the left. Then we can write W in the form of $\overline{t}^k W' t^l$, where W' is a word in $\{A_{i,p}, \overline{A}_{i,p}\}^*$. By Lemma 4.2, $W = \mathbf{1}_G$ if and only if $k = l$ and $w = \mathbf{1}_G$. The former is equivalent to checking if the total exponent of t in W is zero, while the latter can be solved as compressed words in free group using Lorhey's algorithm (Theorem 3.9). □

Denote r as the length of the longest image word when Φ applied to the generators in the F_m, i.e. $r = \max\big\{|\Phi(a_i)| \big| 1 \leq i \leq m\big\}$. The following result gives some justification for Theorem 4.3.

Lemma 4.4. *Suppose that the input word W in Theorem 4.3 has length l, then the following holds.*

(i) Let $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ denote the straight-line program defined in the proof. Then the number of the non-terminals needed is a linear function in l.

(ii) A straight-line program $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ in normal form can be construed in polynomial time with respect to l, such that $w_X = w_A$ and $|\mathcal{X}|$ is bounded by $f(|\mathcal{A}|)$ for some polynomial f.

(iii) We can check if w_X is the identity element in polynomial time with respect to l.

Proof. We will start with (i). Let $s \leq l$ be the number of occurrences of $\tau \in \{t, \overline{t}\}$ in W. In the proof, we moved them to one end of W one by one, we will label them $\tau_1, \ldots, \tau_s$ by the order we moved them.

Initially, we construct m non-terminals $\{A_{i,0} | 1 \leq i \leq m\}$, each of them is sent to a character in $\mathcal{L}_m$. Then, moving every τ_k to one end causes at most m new non-terminals to be created, the new non-terminals will be in the set $\{A_{i,\delta} | 1 \leq i \leq m, \delta \leq k\}$, each has length at most r. Finally, the last word $A \to W'(A_{i,p})$ is constructed. Therefore, at most $m + lm + 1$ non-terminals were created, which proves (i).

Next we proceed to the proof of (ii). Note that the possible length of a production in $\mathbb{A}$

were bounded by r and l' where $l' \leq l$ is the length of W' in the proof. Since r is fixed, the length of every production is bounded by $l + r$ which is a linear function in l. Then (ii) follows from Remark 2.8.

We can now prove (iii). We have that $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ is a straight-line program in normal form with $f(|\mathcal{A}|)$ non-terminals for some polynomial f. Apply Theorem 3.9 to check if $w_X = \mathbf{1}_G$ using $g(f(|\mathcal{A}|))$ time steps which, as a composition of polynomials in $|\mathcal{A}|$, is still a polynomial in $|\mathcal{A}|$. □

Remark 4.5. There is a small subtly which is worth noting in Schleimer's proof, the part about moving the occurrences of t and $\bar{t}$ to construct the root of the straight-line program. An alternative approach would be for every a_i in W, we count the occurrences of t on its left and $\bar{t}$ on its right then take their sum, which also gives us a way to construct an equivalent straight-line program. The alternative method is simpler to implement and will not affect the final time complexity being polynomial. However, Schleimer's method would require less non-terminals to be constructed because we might be able to cancel pairs of t and $\bar{t}$ when we move them.

5 Hagenah's Algorithm

In this section, we will examine the proof for Hagenah's Theorem [4], which allows us to transform a composition system into a straight-line program in polynomial-time.

Theorem 5.1. [15] *Given a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, \widetilde{A}, \mathcal{P} \rangle$, there is a polynomial-time algorithm with respect to $|\mathcal{A}|$ that, finds a straight-line program $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, \widetilde{X}, \mathcal{Q} \rangle$ with $w_{\widetilde{X}} = w_{\widetilde{A}}$ and $|\mathcal{X}|$ is bounded by $f(|\mathcal{A}|)$ for some polynomial f.*

Proof. [16] By Remark 2.23, we may assume that all productions in $\mathbb{A}$ are complete. The idea for the proof is the following: we will construct two types of non-terminals in $\mathcal{X}$. The first type consists of *plain* non-terminals: for every $A \in \mathcal{A}$, we construct a plain non-terminal X in $\mathcal{X}$, such that $w_X = w_A$. In particular, we will have $w_{\widetilde{X}} = w_{\widetilde{A}}$. The second type consists of *decorated* non-terminals, each associated to a plain non-terminal in $\mathcal{X}$. They are additional non-terminals which help us to write the production rules for plain non-terminals without truncations.

We will first introduce some notations related to decorated non-terminal: Fix any plain non-terminal Y in $\mathcal{X}$, and let $0 \leq i \leq j \leq |w_Y|$. Then we denote by $Y^{[i:j]}$ to be the *decorated* non-terminal which produces the same word as the truncation $Y[i : j]$ in $\mathcal{L}^*$ (from this we can deduce that $(Y^{[i:j]})^{[k:l]} = Y^{[i+k:i+l]}$). We have various cases:

[15] The symbol $\widetilde{\cdot}$ is used here for the convenience of the reader when referring to Schleimer's proof, as well as to separate the root from the general non-terminals.

[16] The idea and the tables in the proof were from Schleimer's paper [11]. We will see a diagram and extra explanation to some technical parts.

(a) If $0 < i < j < |w_Y|$ then $Y^{[i:j]}$ is a *subword* non-terminal.

(b) If $i = 0$ and $0 < j < |w_Y|$ then $Y^{[:j]}$ is a *prefix* non-terminal.

(c) If $0 < i < |w_Y|$ and $j = |w_Y|$ then $Y^{[i:]}$ is a *suffix* non-terminal.

(d) If $i = 0$ and $j = |w_Y|$, then $Y^{[:]} = Y$ is the plain non-terminal (a plain non-terminal is a special case of decorated non-terminal).

(e) For $0 \leq i \leq |w_Y|$, $Y^{[i:i]}$ is the empty word.

We will use the idea from Remark 2.21. We will show, by induction on the height $n = \|A\|$, that for every $A \in \mathcal{A}$, a set of at most $4n$ non-terminals are needed in $\mathcal{X}$ with the last one being the plain non-terminal X such that $w_X = w_A$. Then, we can complete the proof by finding the bound for the time steps required for constructing a single production.

If $n = 1$, then $A \to a$ for some $a \in \mathcal{L}$, we simply add one plain non-terminal Y to $\mathcal{X}$ with $Y \to a$. Note that we have $\|Y\| = 1 = \|A\|$.

Now suppose $A \in \mathcal{A}$ has height $n = \|A\| > 1$. Assume via induction for every other non-terminal $B \in \mathcal{A}$ with $\|B\| < n$, a plain non-terminal Y has been added to $\mathcal{X}$ so that $w_Y = w_B$. We will construct the productions for the set of decorated non-terminals and X from bottom up (for example, the production for X will be constructed first, but it will be at the bottom of the productions list, as its production depends on the other decorated non-terminals).

Suppose A has production $A \to B[i:j] \cdot C[k:l]$. Suppose that Y and Z are the plain non-terminals in X corresponding to the non-terminals B and C. Add a plain non-terminal X to $\mathcal{X}$ corresponding to A. Add the decorated non-terminals $Y^{[i:j]}$ and $Z^{[k:l]}$ to X. Add the production rule $X \to Y^{[i:j]} \cdot Z^{[k:l]}$ to $\mathcal{Q}$. It is clear that, after $Y^{[i:j]}$ and $Z^{[k:l]}$ are constructed, we will have $w_X = w_A$.

We now need to construct the production rules for the decorated non-terminal, $Y^{[i:j]}$ (the process for $Z^{[k:l]}$ is similar). If $Y^{[i:j]}$ is a plain non-terminal itself, we can just let $Y^{[i:j]} = Y$, and we are done. Similarly, If $Y^{[i:j]}$ is an empty word, we can let $Y^{[i:j]} = \epsilon$. Otherwise suppose that the plain non-terminal Y produces $U \cdot V$ in $\mathcal{Q}$. (Here it is possible that U and V are themselves decorated non-terminals.) There are several cases and subcases which depend on the type of $Y^{[i:j]}$. The following tables tell us the construction of the new non-terminals in different cases. Suppose first that $Y^{[i:j]}$ is a subword non-terminal.

Subcase	Add to $\mathcal{X}$	Add to $\mathcal{Q}$
$\|w_U\| \leq i$	$V^{[i-\|w_U\|:j-\|w_U\|]}$	$Y^{[i:j]} \to V^{[i-\|w_U\|:j-\|w_U\|]}$
$i < \|w_U\| < j$	$U^{[i:]}, V^{[:j-\|w_U\|]}$	$Y^{[i:j]} \to U^{[i:]} \cdot V^{[:j-\|w_U\|]}$
$j \leq \|w_U\|$	$U^{[i:j]}$	$Y^{[i:j]} \to U^{[i:j]}$

Table 1: Constructions of new non-terminals for the subword case

The table should be read as follows. The first column are the possible subcases, while the second and the third columns tell us the new decorated non-terminals we need to construct and their productions for different subcases.

Next, suppose now that $Y^{[:j]}$ is a prefix non-terminal.

Subcase	Add to $\mathcal{X}$	Add to $\mathcal{Q}$
$\lvert w_U\rvert < j$	$V^{[:j-\lvert w_U\rvert]}$	$Y^{[:j]} \to U \cdot V^{[:j-\lvert w_U\rvert]}$
$j \leq \lvert w_U\rvert$	$U^{[:j]}$	$Y^{[:j]} \to U^{[:j]}$

Table 2: Constructions of new non-terminals for the suffix case

Finally, suppose that $Y^{[i:]}$ is a suffix non-terminal.

Subcase	Add to $\mathcal{X}$	Add to $\mathcal{Q}$
$\lvert w_U\rvert \leq i$	$V^{[i-\lvert w_U\rvert:]}$	$Y^{[i:]} \to V^{[i-\lvert w_U\rvert:]}$
$i < \lvert w_U\rvert$	$U^{[i:]}$	$Y^{[i:]} \to U^{[i:]} \cdot V$

Table 3: Constructions of new non-terminals for the prefix case

We will carry on the same process on every single new decorated non-terminal added to $\mathcal{X}$ (the ones that appeared in the last columns in the above tables). The process will stop when all the new decorated non-terminals added to $\mathcal{X}$ turn out to be plain non-terminals (which are the non-terminals that already existed in $\mathcal{X}$) or empty word. We can see that $\|X\| = \|A\|$.

Next, our aim is to find an upper bound for the number of new non-terminals we constructed. Firstly, note that the creation for plain non-terminal X requires at most two decorated non-terminals to be created, both of them have lesser height than X. Next, from Table 1, we can deduce that the production for a subword non-terminal requires at most one subword non-terminal, or, at most one prefix and at most one suffix non-terminal. Again, the new non-terminals will have lesser height. Finally from Table 2 about the prefix case (Table 3 about suffix case), any prefix (suffix) non-terminal causes at most one prefix (suffix) non-terminal to be created. Similarly, the height decreases.

Therefore, we can deduce that the case that requires the most number of decorated non-terminals is when the plain non-terminal X splits into two subwords, then each of them splits into one prefix and one suffix non-terminal. After that the prefix causes a prefix to be created and a suffix causes a suffix to be created and same process repeats for the new prefixes and suffixes, and finally each produces a plain non-terminal(or empty word).

We can see from Figure 4 that for $A \in \mathcal{A}$ with height n, the creation of X require at most n "rows" of new non-terminals, and every "row" has at most 4 non-terminals. It follows that the creation of X adds at most $4n$ new non-terminals to $\mathcal{X}$ (including X).

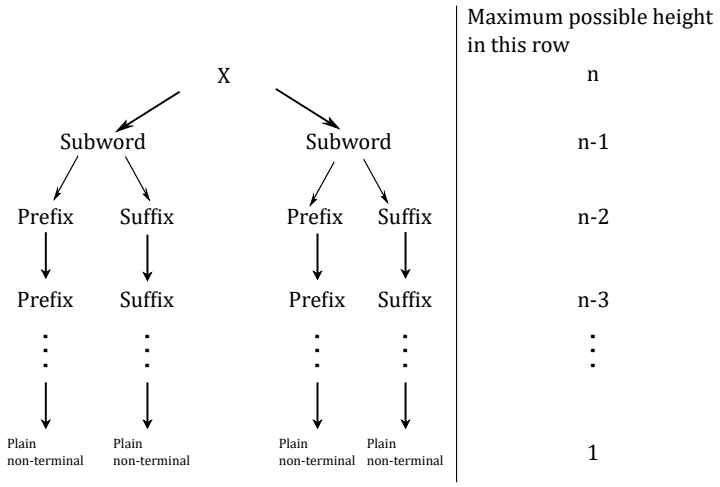


Figure 4: Case when the most decorated non-terminals are required.

This completes the proof for the induction. Let $N = |\mathcal{A}|$ which is greater or equal to the height of the root in $\mathbb{A}$. The case with the most new non-terminals needed is when all of the non-terminals in $\mathbb{A}$ have different heights; in that case we require the following number of non-terminals to be constructed in $\mathcal{X}$:

$$f(N) = \sum_{B \in \mathcal{A}} (4\|B\|) \leq 4 \sum_{i=1}^{N} i = 2N^2 + 2N.$$

Finally, from the tables above, we can deduce that the creation of a single non-terminal requires us to compute at most 2 additions and store 4 integers, which together take at most $g(|\mathcal{A}|)$ time steps for some polynomial g. Then, the total number of time steps required is at most $f(|\mathcal{A}|)g(|\mathcal{A}|)$ which, as a product of polynomials in $|\mathcal{A}|$, is still a polynomial in $|\mathcal{A}|$. □

Remark 5.2. In the above proof, the production of every non-terminal in $\mathcal{X}$ has length at most 2, so we can put $\mathbb{X}$ in normal form using $h(|\mathcal{A}|)$ time steps for some polynomial h by following the idea in Remark 2.8.

Remark 5.3. Suppose we want to prove the following:

> There exists an algorithm, call it $\mathfrak{A}$, which takes a composition system $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ as the input, and outputs a property of the root non-terminal, called it $\mathbf{P}(w_A)$, in polynomial-time with respect to $|\mathcal{A}|$.

Then the Hagenah's Theorem tells us that it is sufficient to show that such an algorithm exists for the straight-line program (which is a composition system with no truncations), then we argue as follows.

Suppose such an algorithm exists for straight-line programs, call it $\mathfrak{A}'$. Given a general composition system, Hagenah's Theorem tells us that we can construct a straight-line

program $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$ with $w_X = w_A$ using $g(|\mathcal{A}|)$ time steps with $|\mathcal{X}|$ bounded by $f(|\mathcal{A}|)$ for some polynomial g and f. Then apply $\mathfrak{A}'$ on $\mathbb{X}$, this take $h(f(|\mathcal{A}|))$ time steps for some polynomial h by assumption, and it outputs $\mathbf{P}(w_X) = \mathbf{P}(w_A)$. This is exactly what we want $\mathfrak{A}$ to output. Overall, the total number of time steps required are bounded by $g(|\mathcal{A}|) + h(f(|\mathcal{A}|))$ which, as a sum of polynomials in $|\mathcal{A}|$, is still a polynomial in $|\mathcal{A}|$. This completes the descriptions of algorithm $\mathfrak{A}$.

6 Plandowski's Algorithm

Finally, we take a look at the Plandowski's Algorithm [10] which allows us to compare whether two straight-line programs output the same word in polynomial-time.

Theorem 6.1. *(Plandowski [10]) Given two straight-line programs in normal forms:* $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ *and* $\mathbb{X} = \langle \mathcal{L}, \mathcal{X}, X, \mathcal{Q} \rangle$*, there is a polynomial-time algorithm with respect to* $(|\mathcal{A}| + |\mathcal{X}|)$*, that decides whether* $w_X = w_A$*.*

Proof. [17] First, we apply the algorithm in Lemma 2.22 on $\mathbb{A}$ and $\mathbb{X}$ which will store the lengths of all the non-terminals, as well as amend the productions so that every production is complete. Note that we assume, as we may, that $\mathbb{A}$ and $\mathbb{X}$ have the same terminal alphabet. Making a copy of $\mathcal{X}$ if necessary, assume that $\mathcal{A} \cap \mathcal{X} = \emptyset$ (i.e. they have distinct labelling). Finally assume that $|w_A| = |w_X|$ (otherwise clearly $|w_A| \neq |w_X|$) and $|A| = m \geq n = |X|$.

We will require the following definition: a triple (B, Y, i) is an *assertion* if $B \in \mathcal{A}$, $Y \in \mathcal{X}$ and $0 \leq i < |w_B|$. There are 2 types of assertion:

(i) If $|w_B| \leq i + |w_Y|$, then (B, Y, i) is an *overlap* assertion.

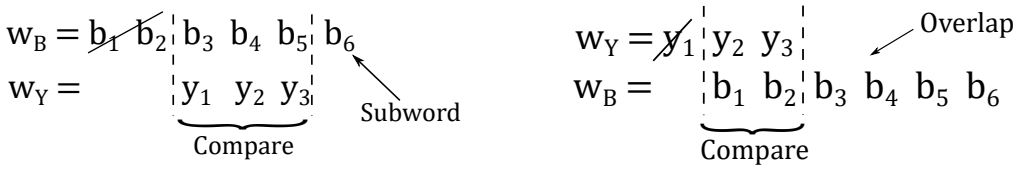
(ii) If $i + |w_Y| < |w_B|$, then (B, Y, i) is a *subword* assertion.

Assertions of the form (Y, B, i) are defined similarly. We will not have a pair of non-terminals from the same program appearing in a single assertion (this is why we need the distinct labelling in the assumption).

An overlap assertion (B, Y, i) is *satisfied* if $w_B[i :] = w_Y[: |w_B| - i]$. Similarly, a subword assertion is *satisfied*[18] if $w_B[i : i + |w_Y|] = w_Y$. We say a set of assertions, Γ, is *satisfied* if every assertion $\gamma \in \Gamma$ is.

[17] Again the idea and the tables in the proof come from Schleimer's paper [11] with a few typos corrected. We added some diagrams and extra explanation to the technical parts. We also proved some of the claims mentioned in Schleimer's proof.

[18] Informally, we can think of an assertion (B, Y, i) as follows: remove first i characters of w_B, call it w'_B. If the length of w'_B is longer than the length of w_Y, it is a subword assertion, otherwise it is an overlap assertion. Let $s = \min\{|w'_B|, |w_Y|\}$. We check its satisfiability by comparing the first s characters of w'_B and w_Y. See Figure 5 for the examples when $w_B = b_1b_2b_3b_4b_5b_6$ and $w_Y = y_1y_2y_3$.



(a) $(B, Y, 2)$ (b) $(Y, B, 1)$

Figure 5: Two assertions with $w_B = b_1b_2b_3b_4b_5b_6$, $w_Y = y_1y_2y_3$.

We will transform a set of assertions Γ_k into another such set, Γ_{k+1}. An assertion (B, Y, i) is called a *terminating assertion*, a special type of overlap assertion, if both B and Y have height 1 and $i = 0$. Our algorithm will check the satisfiability of (B, Y, i) if and only if it is an terminating assertion.

We begin with $\Gamma_0 = \{(A, X, 0)\}$. Throughout the algorithm, the following properties will be maintained:

(a) The set of assertions Γ_{k+1} is satisfied if and only if Γ_k is satisfied.

(b) At most $m + n - k$ non-terminals in $A \cup X$ are mentioned in Γ_k.

(c) For all k, $|\Gamma_k|$ is bounded by $(k + 1)4mn(m + n)$.

We will describe two processes involved to transform a set of assertions: splitting and compacting.

Splitting. Fix Γ, a set of assertions. Suppose out of all non-terminals from $\mathcal{A}$ and $\mathcal{X}$ appearing in Γ, the non-terminal $B \in \mathcal{A}$ has the maximal length. (If the non-terminal with the maximal length is in $\mathcal{X}$, the process will be similar.) Fix $\gamma \in \Gamma$. We will first define Split(γ, B). The aim is to replace γ with a set of equivalent assertions which do not have B as one of the non-terminals. If B does not appear in γ then Split$(\gamma, B) = \gamma$. Otherwise we have $\gamma = (B, Y, i)$ or $\gamma = (Y, B, i)$. Note that γ is either an overlap or subword assertion. Suppose that B has the production $B \to C \cdot D$. We will use the following tables to describe the output of the Split(γ, B) in all the possible cases:

First consider the case when $\gamma = (B, Y, i)$ is an overlap assertion then:

Subcase	Split(γ, B)	Type
$i < \|w_C\|$	(C, Y, i)	overlap
	$(Y, D, \|w_C\| - i)$	either
$\|w_C\| \leq i$	$(D, Y, i - \|w_C\|)$	overlap

Table 4: Splitting an overlap with the first entry having maximal length.

The table should be read as follows: When $i < |w_C|$ then Split(γ, B) contains two assertions: (C, Y, i) which is an overlap and $(Y, D, |w_C| - i)$ which could be either type. When $|w_C| \leq i$ then Split(γ, B) contains a single overlap assertion $(D, Y, i - |w_C|)$.

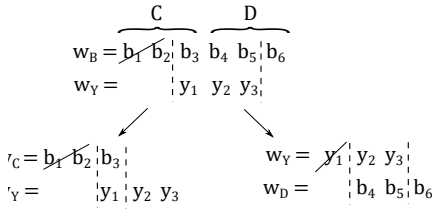


Figure 6: $\text{Split}((B, Y, 2), B) = \{(C, Y, 2), (Y, D, 1)\}$ with $2 < |w_C| < 2 + |w_Y|$.

Notice that the γ is satisfied if and only if the assertion(s) in $\text{Split}(\gamma, B)$ are satisfied. Suppose now that $\gamma = (B, Y, i)$ is a subword[19] assertion, we will preform similar process:

Subcase	$\text{Split}(\gamma, B)$	Type
$i + \|w_Y\| < \|w_C\|$	(C, Y, i)	subword
$i + \|w_Y\| = \|w_C\|$	(C, Y, i)	overlap
$i < \|w_C\| < i + \|w_Y\|$	(C, Y, i)	overlap
	$(Y, D, \|w_c\| - i)$	overlap
$i = \|w_C\|$	$(Y, D, 0)$	overlap
$\|w_C\| < i$	$(D, Y, i - \|w_c\|)$	subword

Table 5: Splitting an subword with the first entry having maximal length.

Next consider the case when B is in the second entry. First suppose $\gamma = (Y, B, i)$ is an overlap assertion, then we have the following:

Subcase	$\text{Split}(\gamma, B)$	Type
$\|w_Y\| \leq i + \|w_C\|$	(Y, C, i)	overlap
$i + \|w_C\| < \|w_Y\|$	(Y, C, i)	subword
	$(Y, D, i + \|w_C\|)$	overlap

Table 6: Splitting an overlap with the second entry having maximal length.

Finally (Y, B, i) cannot be a subword assertion because $|w_B| \geq |w_Y|$. This finishes the definition of $\text{Split}(\gamma, B)$.

Define

$$\text{Split}(\Gamma, B) = \cup_{\gamma \in \Gamma} \text{Split}(\gamma, B).$$

From this definition, we obtain the following result:

Claim 1. *A set of assertions* Γ *is satisfied if and only if* $\text{Split}(\Gamma, B)$ *is satisfied.* □

Define $o(\Gamma)$ to be the number of overlap assertions in Γ. Similarly let $s(\Gamma)$ be the number of subword assertions in Γ. So $|\Gamma| = o(\Gamma) + s(\Gamma)$. From the tables above, we see that one

[19] Carrying on with the previous example, and let $w_C = b_1 b_2 b_3$, $w_D = b_4 b_5 b_6$, then the subword $(B, Y, 2)$ splits into two assertions: $(C, Y, 2)$ and $(Y, D, 1)$, see Figure 6.

overlap assertion can split into at most 2 overlap assertions and 1 subword assertion (see Table 4 and 6), likewise one subword assertion can split into at most 2 overlap assertions and 1 subword assertion (see Table 5). We deduce:

Claim 2. *Suppose that* Γ *is a set of assertions. Then*

$$o(\text{Split}(\Gamma, P)) \leq 2o(\Gamma) + 2s(\Gamma),$$
$$s(\text{Split}(\Gamma, P)) \leq o(\Gamma) + s(\Gamma).$$

where $P \in \mathcal{A} \cup \mathcal{X}$ *is a non-terminal of maximal length in* Γ. □

Compact. Now we will define Compact(Γ). The aim of compacting is to reduce the number of overlap assertions we need after splitting.

Recall that, if u is a word, then $p \in \mathbb{N}$ is a *period* of u if

(i) $1 \leq p \leq |u| - 1$ and

(ii) $u[i] = u[i + p]$ for all $0 \leq i \leq |u| - 1 - p$.

Our goal is to replace three particular assertions with two equivalent ones. The following result acts as the first stepping stone towards this goal by allowing us to replace two overlap assertions with one overlap and one period (and vice versa).

Claim 3. *Suppose that* $\gamma = (B, Y, i)$ *and* $\gamma' = (B, Y, j)$ *are overlap assertions with* $i < j$. *Then* γ *and* γ' *are satisfied if and only if* γ *is satisfied and* $j - i$ *is a period of the word* $w_B[i :]$.

Proof of Claim 3. We will suppose all the assumptions in the claim and γ are satisfied. Since γ is satisfied, we have

$$(w_B[i :])[t] = (w_Y[: |w_B| - i])[t] \qquad \text{for all } 0 \leq t \leq |w_B| - i - 1.$$

($\Rightarrow$) Suppose γ' is satisfied. We want to show $j - i$ is a period of $w_B[i :]$. For every $0 \leq s \leq (|w_B| - i) - 1 - (j - i) = |w_B| - j - 1$, we have

$$\begin{aligned}
&(w_B[i :])[s + (j - i)] \\
=&(w_B[j :])[s] && \text{by repeated truncations} \\
=&(w_Y[: |w_B| - j])[s] && \text{since } \gamma' \text{ is satisfied} \\
=&(w_Y[: |w_B| - i])[s] && \text{since } w_Y[: |w_B| - j] \text{ is a prefix of } w_Y[: |w_B| - i] \\
=&(w_B[i :])[s] && \text{since } \gamma \text{ is satisfied.}
\end{aligned}$$

Therefore, $(j - i)$ is a period of the word $w_B[i :]$.

The proof for ($\Leftarrow$) direction is similar. □

Next, we will require the restricted version of Periodicity Lemma [8, p. 239], which allows us to replace two periods with one period.

Lemma 6.2. *If p and q are periods of u, where $p+q \leq |u|$, then $\gcd(p,q)$ is also a period of u.*

Combining the above two claims, the following result allows us to replace three particular overlap assertions by one overlap and one period.

Claim 4. *Suppose that $\gamma = (B,Y,i)$, $\gamma = (B,Y,j)$, and $\gamma = (B,Y,k)$ are overlap assertions with $i < j < k$ and $j - i + k - i \leq |w_B| - i$. Then γ, γ', and γ'' are satisfied if and only if γ is satisfied and $\gcd(j-i, k-i)$ is a period of $w_B[i:]$.*

Proof of Claim 4. We will suppose all the assumptions in the claim and γ are satisfied.

($\Rightarrow$) Suppose γ' and γ'' are satisfied. Then, by Claim 3, $j-i$ and $k-i$ are periods of $w_B[i:]$. Then by Lemma 6.2, $\gcd(j-i, k-i)$ is a period of $w_B[i:]$.

($\Leftarrow$)Suppose $\gcd(j-i, k-i)$ is a period of $w_B[i:]$. It is clear that if p is a period of a word u and $1 \leq mp \leq |u| - 1$ for some $m \in \mathbb{N}$, then mp is also a period of u. Since $j - i = m\gcd(j-i, k-i)$ for some $m \in \mathbb{N}$, $j-i$ is a period of $w_B[i:]$. Hence by Claim 3, γ' is satisfied. Similarly, γ'' is satisfied. □

Equivalently, follow from Claim 3, the overlap assertions γ, γ' and γ'' are satisfied if and only if γ and $\delta = (B, Y, i + \gcd(j-i, k-i))$ are satisfied[20]. This leads to the definition of SimpleCompact: given $\{\gamma, \gamma', \gamma''\}$ as in the hypothesis of Claim 4 define SimpleCompact$(\{\gamma, \gamma', \gamma''\}) = \{\gamma, \delta\}$, with δ as above.

Now, for any set of assertions Γ, we define Compact(Γ) as follows: order the sets $\mathcal{A}$ and $\mathcal{X}$ of non-terminals. If there are duplicate assertions, only keep one of them and remove the others. Also, check the satisfiability of every terminating assertion, if a terminating assertion is satisfied, remove it; otherwise, the algorithm output $w_X \neq w_A$ and stops. We will proceed as if all terminating assertions are satisfied. Let $\Gamma^0 = \Gamma$. For all j we may assume that the set Γ^j is ordered lexicographically. Then we scan through Γ^j to find the first consecutive triple of assertions that satisfies the requirements of Claim 4. Apply SimpleCompact to this triple to obtain Γ^{j+1} and notice that $|\Gamma^{j+1}| = |\Gamma^j| - 1$. We notice the following bound:

Claim 5. *Suppose that $|\Gamma| \geq 3$, then the number of triples[21] we need to check is at most $|\Gamma| - 2$.*

[20]It can be easily checked that $i + \gcd(j-i, k-i) \leq |w_B|$, so this is a well defined assertion.

[21]This claim was not in the original proof, because we could in theory check all the possible $\binom{|\Gamma|}{3} = \mathcal{O}(|\Gamma|^3)$ triples which does not affect the final result being polynomial-time. However we will used this claim in our implementation.

Idea of proof for Claim 5. The claim is clearly true when $|\Gamma| = 3$. Suppose we have

$$\Gamma = \{\gamma_1 = (B, Y, i), \gamma_2 = (B, Y, j), \gamma_3 = (B, Y, k), \gamma_4 = (B, Y, l), \ldots, \gamma_h\}$$

with at least 4 assertions ordered lexicographically. The idea is to realise the fact that the r^{th} triple that need to be checked contain an assertion of index $r + 2$. We will take a look at the first few triples that we need to check. The first one is $\{\gamma_1, \gamma_2, \gamma_3\}$, if they satisfy the condition, we replace γ_2 and γ_3 by δ describe above. Then the second triple that will be checked is $\{\gamma_1, \gamma_2, \gamma_4\}$.

Otherwise suppose $\{\gamma_1, \gamma_2, \gamma_3\}$ did not satisfy the condition. Then, by using the inequality in Claim 4, we have

$$j + k - i > |w_B|. \tag{1}$$

Then, it might seem that we have a few possible triples to check: $T_1 = \{\gamma_1, \gamma_2, \gamma_4\}$, $T_2 = \{\gamma_1, \gamma_3, \gamma_4\}$ and $T_3 = \{\gamma_2, \gamma_3, \gamma_4\}$. But, from equation (1) and the fact that $i < j < k < l$, it follows that $j + l - i > |w_B|$ and $k + l - i > |w_B|$. Hence T_1 and T_2 do not satisfy the condition. Therefore we can deduce that the only triple that needs to be checked is T_3. It follows that, in either case, the third check will involve the assertion with index 5 (if we have more than 4 assertions). We can check, by induction, that the r^{th} triple that needs to be checked contains an assertion of index $r + 2$. □

Finally, if no such triples exist then set $\text{Compact}(\Gamma) = \Gamma^j$. We also obtained the fact that,

Claim 6. Γ *is satisfied if and only if* $\text{Compact}(\Gamma)$ *is satisfied.*

Going back to our inductive definitions in the beginning, we will define Γ_{k+1} in terms of Γ_k. Suppose that $\mathrm{P} \in \mathcal{A} \cup \mathcal{X}$ is a non-terminal appearing in Γ_k with maximal length $|w_P|$. Then take

$$\Gamma_{k+1} = \text{Compact}(\text{Split}(\Gamma_k, P)).$$

Note that property (a) above is guaranteed by Claims 3 and 6 while property (b) is provided by the fact that splitting process will remove one non-terminal with maximal length, and after that it will not appear in any of later assertions. We need to bound $|\Gamma^k|$ to maintain property (c).

Fix $B \in \mathcal{A}$ and $Y \in \mathcal{X}$. Consider $\{(B, Y, i_j)\}_{j=1}^{N}$ the set of overlap assertions in Γ_k that mention B and Y in that order and indexed with $i_j < i_{j+1}$. As Γ_k is compact it follows from Claim 4 that

$$\begin{aligned} |w_B| - i_j &< i_{j+2} - i_j + i_{j+1} - i_j. \\ &< i_{j+2} - i_j + i_{j+2} - i_j && \text{since } i_{j+1} < i_{j+2} \\ &= 2(i_{j+2} - i_j). \end{aligned}$$

It follows that

$$\frac{1}{2}(|w_B| - i_j) < i_{j+2} - i_j. \tag{2}$$

Claim 7. *The number of such assertions is bound by* $N \leq 2\log_2(|w_B|) + 1$.

Proof. For $1 \leq j \leq N-2$, we have:

$$\begin{aligned}(w_B - i_{j+2}) + (i_{j+2} - i_j) &= w_B - i_j \\ &= \frac{|w_B| - i_j}{2} + \frac{|w_B| - i_j}{2} \\ &< (i_{j+2} - i_j) + \frac{|w_B| - i_j}{2} \qquad \text{by Equation(2).}\end{aligned}$$

It follows that,

$$w_B - i_{j+2} < \frac{|w_B| - i_j}{2}.$$

By induction,

$$w_B - i_{j+2k} < \frac{|w_B| - i_j}{2^k}.$$

Take j to be 1, we get

$$w_B - i_{1+2k} < \frac{|w_B| - i_1}{2^k} \leq \frac{|w_B|}{2^k}. \tag{3}$$

From the definition of assertion, $1 \leq |w_B| - i_N$, we have two cases:

Case 1: Suppose N is odd.
Take $k = \frac{N-1}{2}$, then $N = 1 + 2k$. Substitute k into Equation (3), we get

$$1 \leq |w_B| - i_N < \frac{|w_B|}{2^{\frac{N-1}{2}}}.$$

Case 2: Suppose N is even.
Take $k = \frac{N-2}{2}$, then $N - 1 = 1 + 2k$. Substitute k into Equation (3), we get

$$1 \leq |w_B| - i_N < |w_B| - i_{N-1} < \frac{|w_B|}{2^{\frac{N-2}{2}}}.$$

Hence, we have $2 \leq \frac{|w_B|}{2^{\frac{N-2}{2}}}$ and therefore $1 \leq \frac{|w_B|}{2^{\frac{N}{2}}} < \frac{|w_B|}{2^{\frac{N-1}{2}}}$.

In both cases, we have $2^{\frac{N-1}{2}} < |w_B|$, and the claim follows. □

Using the upper bound for length of w_B we saw in Lemma 2.14 and the Claim 7, we obtain the following:

$$N \leq 2\log_2(|w_B|) + 1 \leq 2\|B\| + 1. \tag{4}$$

Hence, we obtained an upper bound for the number of possible third entries i for an overlap assertion in the form (B, Y, i).

We will try to find an upper bound for the number of overlap assertions in Γ^k that does not depend on k. Suppose we have an overlap assertion of the form (B, Y, i) where $B \in \mathcal{A}$ and $Y \in \mathcal{X}$. Since $|\mathcal{A}| = m$ and $|\mathcal{X}| = n$, we have m choices for the first entry; n choices for the second; and by Equation (4), $2\|B\| + 1$ possible values for the last entry. Hence there are $mn(2m + 1)$ possible overlaps of such form. Similarly, there are $nm(2n + 1)$ possible overlaps of the form (Y, B, i). Since $m \geq n$, we have for all t,

$$o(\Gamma_t) \leq mn(2m + 1) + nm(2n + 1) \leq 4mn(m + n) \tag{5}$$

We also obtain a bound for the number of subword assertion in Γ_{k+1}:

$$\begin{aligned}
s(\Gamma_{k+1}) &= s(\text{Compact}(\text{Split}(\Gamma_k))) \\
&\leq s(\text{Split}(\Gamma_k)) \\
&\leq o(\Gamma_k) + s(\Gamma_k) && \text{by Claim (2)} \\
&\leq 4mn(m + n) + s(\Gamma_k) && \text{by Equation (5)} \\
&\leq (k + 1)(4mn(m + n)) && \text{by induction and } s(\Gamma_0) = 0.
\end{aligned}$$

This shows that the property (c) is also satisfied for Γ_{k+1}. Hence, by induction, all three properties are maintained for Γ_k throughout the algorithm.

Finally, let us denote $M = m+n$ and take a look at a summary of the main operations we performed in the algorithm to see that the whole process indeed takes a polynomial-time with respect to M.

First, we apply Lemma 2.22 to both $\mathbb{A}$ and $\mathbb{X}$. As we have seen in the proof of the lemma, this takes polynomial time with respect to M.

Next, given a set of assertion Γ_k with $|\Gamma_k|$ bounded by $g(M)$, a polynomial function in M, we have the following two operations:

(Split) We start off by finding the maximum length of all the non-terminals appeared in Γ_k, i.e. we need to find the maximum in the set of $2|\Gamma_k|$ integers, this take $f_k(|\Gamma_k|)$ time steps, for some polynomial f_k. Next, every assertion is split into at most two assertions, so we create at most $2|\Gamma_k|$ new assertions, creating a single assertion involves at most one addition or subtraction, storing 2 non-terminals and one integer which together takes polynomial time with respect to M (See the tables from earlier). It follows that constructing $2|\Gamma_k|$ new assertions also takes polynomial time with respect to M. Finally,

we can deduce that the whole splitting process takes polynomial time with respect to M.

(Compact) Note that we have at most $2|\Gamma_k|$ assertions created by splitting. The following operation take polynomial time: sorting them in lexicographic order, deleting repeated assertions and checking the satisfiability of all the terminating assertions. Next we require at most $2|\Gamma_k|$ searches and SimpleCompact operation. A single search requires comparing 2 integers and 4 additions and subtraction together, which takes polynomial time with respect to M. Applying SimpleCompact to a triple involves deleting two assertions, computing the greatest common divisor of two integer, 3 additions and subtractions to create a new assertion; this also takes polynomial time with respect to M. It follows that all the searches and SimpleCompact operations together take polynomial time with respect to M. Therefore the whole compacting process takes polynomial time with respect to M.

Finally, since every non-terminal is split for at most value of k, we need to transform the set of assertions at most M times (in particular, Split and Compact are performed for at most M times). Therefore, the whole algorithm is also a polynomial time with respect to $M = \|A\| + \|X\|$. □

7 Explaining the Matlab Code

The objects mentioned in the algorithms from the earlier theorems are easily represented as matrices (such as set of assertions in Theorem 6.1 and set of production rules). Matlab has excellent array handling capability; several mathematical operations that work on matrices are built-in to the Matlab (such as sorting rows of a matrix and remove repeated rows) which made it a natural choice. The algorithms corresponding to different theorems were written as different user-defined functions. Most of the objects will be referred to by their indices. As different objects might have the same indices, for example non-terminal A_1 and terminal a_1, this might seem to cause some ambiguities, we will overcome them through careful choices of their representations.

Representing straight-line programs

Suppose that $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ is a straight-line program with $|\mathcal{A}| = n$. The set of n production rules are stored as a matrix. We will first look at how straight-line programs in normal form [22] are stored. A production will have the form $A_i \to A_j \cdot A_k$ with length 2 and A_j, $A_k \in \mathcal{A}$ or $A_i = a_j$ with length 1 and $a_j \in \mathcal{L}$. Hence it is natural to define an n by 3 matrix with the i^{th} row storing the production of A_i. If we have $A_i \to A_j \cdot A_k$, then the i^{th} row is going to be $[i, j, k]$. Otherwise we have $A_i \to a_j$, for some non-terminal a_j, then

[22] In the case when $\mathbb{A}$ has only one single non-terminal A, with the production $A \to \epsilon$, we store it as $[1, 0, 0]$.

the i^{th} row is $[i, 0, j]$, where the 0 in the second entry indicates that the non-terminal is sent to a terminal (since the index of non-terminals will never be 0, when A has production rule that consists of non-terminals, 0 will not appear in the second entry, this resolves the ambiguity). Note that the number of non-terminals, $|\mathcal{A}|$, is equal to the number of rows.

Example 7.1. The straight-line program in Example 2.4 with $m = 3$ is represented as

$$P = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 0 & 2 \\ 3 & 1 & 2 \\ 4 & 2 & 3 \\ 5 & 3 & 2 \\ 6 & 4 & 5 \end{bmatrix}.$$

Suppose now that straight-line program $\mathbb{A}$ is not in normal form. Again, we use the second entry of each row to indicate whether the production is a string of non-terminals or terminals. Since the production rule might have different lengths, when we represent them as rows, we need to add dummy zeros to the rows with the shorter lengths, so that every row has the same size: if $A_i \to a_{j_1} \cdot a_{j_2} \cdots a_{j_s}$, then i^{th} row is $[i, 0, j_1, j_2, \ldots, j_n, 0, \ldots, 0]$; if $A_i \to A_{j_1} \cdot A_{j_2} \cdots A_{j_s}$, then the i^{th} row is $[i, j_1, j_2, \ldots, j_n, 0, \ldots, 0]$. Finally, if $A_i \to \epsilon$, the i^{th} row will be $[i, 0, 0, \ldots, 0]$.

Representing composition systems

Recall that a compositions system is straight-line program in normal form with truncation. So each production will have the form $A_r \to a_p$ or $A_r \to A_p[i, j] \cdot A_q[k, l]$ which has 2 indices and 4 integers in the production. Hence, if $\mathbb{A} = \langle \mathcal{L}, \mathcal{A}, A, \mathcal{P} \rangle$ is a composition system with $|\mathcal{A}| = n$, it will be natural to store it as n by 7 matrix. If we have $A_r \to a_p$, then the r^{th} row is $[r, 0, 0, 0, 0, 0, p]$ (Again, the second entry being 0 indicates that this non-terminal is sent to a terminal. The zeros between third and sixth entry are dummy zeros, we might sometimes use ∞ instead for convenience). Otherwise if we have $A_r \to A_p[i, j] \cdot A_q[k, l]$ then the r^{th} row will be $[r, p, i, j, q, k, l]$. Finally, if we have abbreviation in the production, we use 0 to indicate the truncation is from the beginning and ∞ to indicate the truncation is till the end[23], e.g. if $A_r \to A_p[i:] \cdot A_q[:l]$, then the r^{th} row will be $[r, p, i, \infty, q, 0, l]$.

[23] In Matlab, the maths symbol ∞ is represented as `inf`. The ∞ symbol was chosen because informally it indicates all the way to the end.

Example 7.2. The composition system in Example 2.12 with $m = 3$ is represented as

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 & 0 & 2 \\ 3 & 1 & 0 & \infty & 2 & 0 & \infty \\ 4 & 2 & 0 & \infty & 3 & 0 & 2 \\ 5 & 3 & 0 & \infty & 2 & 0 & \infty \\ 6 & 4 & 0 & \infty & 5 & 0 & 2 \end{bmatrix}.$$

Representing compressed words

Finally, a compressed word is when we might have inverses in the production. We represent the inverse by the negative sign. For example, if we have a compressed composition system, every row still has size 7. If we have $A_r \to \overline{a_p}$, then the r^{th} row is $[r, 0, 0, 0, 0, 0, -p]$; otherwise we have $A_r \to \overline{A_p}[i, j] \cdot \overline{A_q}[k, l]$ then the r^{th} row will be $[r, -p, i, j, -q, k, l]$.
Remark 7.3. Clearly, the first column of the matrices we described above is redundant as its entries are just the index of the rows, but it will give a better visualisation of production rules.

We will sometimes refer to a straight-line program or composition system by its production rules. In most of cases, the root is the last non-terminal.

7.1 Algorithm A for Lemma 2.6

We will refer to the production of a non-terminal by just the *non-terminal*. For example, if $A \to \alpha_1 \cdot \alpha_2 \cdot \cdots \cdot \alpha_l$, then we will say A has length l.

Algorithm A takes a straight-line program production, P_0 as input, with every production in P_0 has the form of a string of terminals or non-terminals. It outputs a straight-line program in normal form, P_1, which produce the same word as P_0.

The straight line programs that we will produce in the later algorithms will not have any production that contains a non-terminal that is sent to an empty word. Hence, we can skip **(Stage 1)** from the proof. For **(Stage 2 case 2)**, our approach was to construct some new non-terminals and add them to the input straight-line program $\mathbb{A}$ as well as change the productions in $\mathbb{A}$. While in the algorithm we will create a new straight-line program, P_1, that contains a "copy" of the original non-terminals (sets of non-terminals that produce the same words as the ones in P_0 with the corresponding indices stored in C) and the additional non-terminals.

In the beginning of our algorithm, we check if the root is sent to an empty word (Line 7). If it is, this corresponds to **(Stage 2 case 1)** in the proof, then P_1 will have only one

non-terminal which produces the empty word (Line 119). Next, we suppose that the root is not sent to the empty word, which corresponds to **(Stage 2 case 2)** in the proof.

First, we find an upper bound for the number of non-terminals required in P_1. In the case when a non-terminal A is sent to a string of l_p terminals, we require at most $2l_p$ new non-terminals to be constructed; this turns out to be the worst case as otherwise when A is sent to a string of l_p non-terminals, we require at most l_p non-terminals to be created.

Note that the length of the longest production is bounded by the number of columns in P_0 which we denoted by N_c (Line 8). Hence, we obtained a bound for the maximum number of non-terminals required for P_1 which is $2N_cN_r$ (Line 9), where N_r is the number of terminals in P_0. We also initialised an integer, N_n, which stores the number of non-terminals that have been defined in P_1 (Line 17). Lastly, given a non-terminal A_s in P_0, there will be a non-terminal X_j in P_1 such that A_s and X_j produce the same word; we will store this in the array C, by letting s^{th} entry of C, $C(s)$, equal to j.

Next, we have a for-loop (Lines 20 –117): for every A_s with production W_s, we construct a set of new non-terminals to P_1 with the last one corresponding to A_s. If W_s is a string of terminals (i.e.when the second entry of W_s is 0), we need to count the number of characters in W_s and stored this as l_p (Lines 30–40). Proceed as in the proof, in the first nested loop (Line 30), for every character $W_s(i)$ in W_s, we add a row $[N_n, 0, W_s(i)]$ to P_1. If $l_p = 1$, we are done. Otherwise, we construct other new $l_p - 1$ non-terminals inductively as in the proof (Lines 42 –61), so X can have production in normal form.

If W_s corresponds to a string of non-terminals, we construct new non-terminals inductively similar to the proof, while we also need to access the array C to find the corresponding non-terminal in P_1 that produce the same word as the non-terminal in P_0 (Lines 66–113).

In the algorithms which we will introduce in the later theorems, the production W_s might contain inverses. The function `inverse_sign` at the end of the algorithm takes an inverse non-terminal, say $-A_s$, from P_0, and outputs its the corresponding inverse non-terminal in P_1: it first finds $C(s)$, the index of non-terminal in P_1 which produces the same word as A_s, then it outputs $-C(s)$ which corresponds to $-A_s$ in P_1 (Lines 123–134).

The way we construct new non-terminal when we have inverses is very similar to what we did for the usual case, except for a small technical detail (we will use the same notation as in the proof here): suppose A in $\mathbb{A}$ has production consisting of exactly one inverse, i.e. $A \to \overline{B}$, and $B \to C \cdot D$ in $\mathbb{A}$. Then we need to reverse the order of non-terminals, i.e. A in the normal form will have production $A \to \overline{D} \cdot \overline{C}$ (Lines 106–108).

At the end of the loop we let $C(s) = N_n$, because N_n is the non-terminal in P_0 that corresponds to A_s in P_0 (Line 116).

Finally, the last line of the algorithm removes the trailing zero rows in P_1 (Line 136) and this will be the output of the algorithm.

7.2 Algorithm B for Lemma 2.22

Algorithm B takes a composition system production CS_0 as input and outputs a double: (i) an array, L, whose i^{th} entry stores the length of word produced by the i^{th} non-terminal in CS_0; (ii) and a composition system production CS_1 with all productions being complete, and its root produces the same word as CS_0.

First, we define an array L of size N_c (Line 7). We only stored the length of the non-terminals but not their inverse because the inverse function preserves the length of a word. Hence, we put `abs()` in front of indices of non-terminals before we pass it to the array L (i.e. if we want to retrieve the length of either A or $\overline{A}$, we will get the length of A).

Similarly to the previous algorithm, we have a for-loop (Lines 14–64): for every A_s, if A_s is sent to a terminal, then it has length 1 (Lines 16 –18). Otherwise we will have two truncations to amend; for the first, we start by replacing the negative indices (Lines 22-30), then amend the abbreviated truncation (Lines 33–36). Similarly, we amend the second truncation (Lines 39–50). For convenience of latter algorithms, if one of the truncations in the production gives a word of length zero, we make it our second truncation (Lines 54–58). Finally, we calculate the length of the current non-terminal A_s and store it as s^{th} element of L (Lines 61 –62).

At the end, after we amended the productions in CS_0, we set it as our output CS_1 (Line 67).

7.3 Algorithm C for Lemma 2.24

Algorithm C takes a double as the input: (i) A set of productions, CS in a composition system (ii) An index r, the program will output the $(r+1)^{th}$ character of root non-terminal. It outputs *char*, the $(r+1)^{th}$ character of root non-terminal.

To simplify the notation, initially we let $i = r + 1$ and nt be the root (Lines 6–7), we want to output the i^{th} character of nt.

We have a while-loop (Lines 13–27) which stops when A_{nt} is a non-terminal that is sent to a terminal: if the character is in the first truncation, we replace nt by the index of the first non-terminal in the production, and replace i as in the proof (Lines 15–19); otherwise the character is in the second truncation, we perform the similar process (Lines 23–25). After we exit the loop, we set *char* to be the character produced by A_{nt} (Line 29).

7.4 Algorithm D for Theorem 5.1

Algorithm D takes a composition system, CS as input, and it outputs a straight-line program SLP such that the last non-terminal (root) of SLP produces the same word as

the last non-terminal (root) of CS.

First we initialise a matrix SLP (Line 8) with the maximum number of rows required (in the proof we saw the bound is $2N_r^2 + 2N_r$). Also, the integer N_t records the number of non-terminals (Line 18). Lastly, we define an array that will store the indices of plain non-terminals in SLP (Line 20).

Similar to algorithm A, we have a for-loop (Lines 25–99): for every A_s, if A_s is sent to a terminal, we add a corresponding plain non-terminal (Lines 26–29), otherwise we need to construct the decorated non-terminals that required to define A_s. Since the non-terminals are stored by their indices, and we build the non-terminals from bottom up, we need to give an initial guess for the incrementation of the index of the last non-terminal, which is $M_s = 4s$ (Lines 32–35). As we saw in the proof, $4s$ is the number of the most non-terminals are required. We denote by Y, Z, i, j, k and l as variables in the proof (Lines 40–49). Then we construct the production rules for $Y^{[i:j]}$ (function `add_new_pro` in Lines 101–177). We will describe this process as 2 stages: before and after the subword split. We define U and V as the variables in the proof.

Initially, we have a while-loop (Lines 111–142) which constructs new productions for the case when a subword produces another subword. Once we exit this loop, the final subword cannot produce another subword which leaves with the following three possible cases:

(1) The final subword produces a suffix and prefix (Lines 146 – 163);

(2) The final subword produces a suffix (Lines 165–168) and

(3) The final subword produces a prefix (Lines 172–173).

We know that a prefix non-terminal will always produce a prefix (and maybe an additional plain non-terminal which has already being constructed), so we define a while-loop to repeat the process (function `prefix_case` between Lines 180 – 218) until the prefix produces a plain non-terminal. We also used the similar method for the suffix non-terminal (Lines 221 – 256 function `suffix_case`). We then construct the production rule for $Z^{[k:l]}$ using the same method (Lines 65–74).

At the end we correct all the indices for the new non-terminal by taking away R_s from them (Lines 80–83), then we compute the lengths of new non-terminals constructed (Lines 86–93). Finally, we store the index of the new plain non-terminal in array C (Line 98).

7.5 Algorithm E for Theorem 6.1

Algorithm E takes two straight-line programs P_A and P_X and outputs a boolean value, E, which tells whether the roots of P_A and P_X produce the same word.

An assertion is stored as an array of size 4, the first two entries store the indices of non-terminals, and the third is the integer i as in the proof. The last entry indicates whether the first entry is in P_A, it stores 1 for yes and 0 for no. For example, $(A_5, X_4, 2)$ will be stored as $[5, 4, 2, 1]$ while $(X_5, A_4, 2)$ will be stored as $[5, 4, 2, 0]$ where A_5 and A_4 are non-terminals in P_A.

First we define a similar function to Algorithm B which takes a straight-line program as input and calculate the length of every non-terminal (function `compute_length_slp`). Then we compute the length of every non-terminal in both P_A and P_X (Lines 9–10). We define a matrix Γ_k that will store all the assertions (Lines 19–21). Initially we set the boolean value E to be true (Line 29). We have a while-loop (Lines 31–412) which will keep transforming the set of assertion until it becomes an empty set. We define O_k to store overlap assertions and S_k to store subword assertions (Lines 75–78). First we find the non-terminal with maximal length (Lines 37–69) and store it as M_k. Next, we start Split process by going through every assertion γ_h, where we considered all the possible cases as in the tables in the proof (Lines 81 – 359), for every overlap assertion we also check if it is a terminating assertion (for example Line 119-123) and proceed as in the proof[24]. Then, we begin Compact process on O_k, we remove the repeated assertions and order the assertions lexicographically (Lines 366 – 368). Then we have a while-loop which goes through all the triples (Lines 373–403). If the triple satisfies the condition as in the proof, replace the triple accordingly (Lines 392–397). Finally combine the O_k and S_k as the new set of assertions Γ_k (Line 409) and repeat the transformation process.

7.6 Algorithm F for Theorem 2.25

Algorithm F takes two composition systems CS_A and CS_X and outputs the largest integers k such that the first k characters of CS_A and CS_X are the same.

As in the proof, we combined the binary search (Lines 15 – 36) and the $prefixchecker$ algorithm (function `prefix_checker`, Lines 38–59).

7.7 Algorithm G for Lemma 3.5

Algorithm G takes a composition system CS_0 and outputs a compressed composition system CS_1 such that its root is the inverse of the root of CS_0.

By following the proof, we changed the indices and the order of two truncations non-terminals in every production (Lines 9–24).

[24] In the proof, we check the satisfiability of every terminating assertion in Compact operation, while here we do it in Split operation. This will not affect the time complexity.

7.8 Algorithm H for Theorem 3.7

Algorithm H takes a compressed straight-line program P as input and outputs a compressed composition system CS, where the s^{th} non-terminal in CS produces the reduced word of the s^{th} non-terminal in P.

Recall that in the proof, we needed to apply generalised Theorem 5.1 and 6.1. There is a minor problem as our algorithms for these two theorems do not have the functionality to handle inverses in the production (which are represent by non-terminal with negative indices). Instead of changing theses algorithms, we introduce the function `amend_inverses`, which is a slight variation of the Algorithm G (function `inverse_pro`). It takes a compressed composition system CS_0 as an input, and outputs a composition system CS_1, which consists of every non-terminal in CS_0 with its inverse immediately after the non-terminal. Hence, the indices for all the non-terminals and their inverse are positive and we can construct all the production rules in CS_1 without the inverses. For example[25], if we have $A_1 \to a_1$, $A_2 \to A_1 \cdot \overline{A_1}$ in CS_0; then in CS_1 we have $X_1 \to a_1$, $X_2 \to \overline{a_1}$ and $X_3 \to X_1 \cdot X_2$ in CS_0 with X_1, $X_2$and X_3 corresponding to A_1, $\overline{A_1}$ and A_2 respectively.

Now, we have enough tools to construct CS. We have a for-loop (starts at Line 12 in function `free_reduction`), for every non-terminal A_s in P with production $A_i \cdot A_j$, we construct $\overline{X_i}$ (Lines 18–26), X_j (Lines 29–32) and then X_s (Lines 35–44) as in the proof.

7.9 Algorithm J for Theorem 3.9

Algorithm J takes a double as an input: (i) compressed straight-line program, P, and (ii) an index, s ,of the non-terminal in P, it outputs a boolean value I which tells whether A_s represents the identity.

We follow the proof of the theorem and apply Algorithm H to compute the reduction of A_s (Line 6) and use Algorithm B to calculate the length of the reduction (Line 7).

7.10 Algorithm K for Theorem 4.3

Let us first take a look at how we represent an automorphism of a finitely generated free group and a word in free by-cyclic group as matrices. For simplicity of notation, we will refer to an automorphism or word by its matrix representation. An automorphism in $\mathrm{Aut}(F_m)$, Φ ,is represented by a matrix with m rows, where the i^{th} row consists of the i^{th} generator of F_m and the image of Φ on that generator. Some rows might have dummy zeros at the end to make sure all the rows have the same length. For example, if we have

[25]For simplicity of notation, we used a straight-line program example here, we can use a same idea for the composition systems.

$F_3 = \langle a_1, a_2, a_3 \rangle$, and $\Phi(a_1) = a_1 a_2 a_3$, $\Phi(a_2) = a_1 a_2$ and $\Phi(a_3) = \overline{a_1}$, then Φ will be represented as

$$\Phi = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 2 & 1 & 2 & 0 \\ 3 & -1 & 0 & 0 \end{bmatrix}. \tag{6}$$

On the other hand, a word W with length l is represented by a 2 by l matrix, where the i^{th} column represents the i^{th} character of W. If the i^{th} character of W is the generator a_j, then the i^{th} column is going to be $(j, 0)^T$. Otherwise the i^{th} character is the generator t^p where $p = \pm 1$, then the i^{th} column is going to be $(0, p)^T$. We continue with the earlier example and suppose that $W = a_1 t a_2 t^{-1} a_3 t$ in F_3. Then W will be represented as

$$W = \begin{bmatrix} 1 & 2 & 0 & 3 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix}.$$

Algorithm K takes a double as the input: (i) An automorphism, Φ, of a finitely generated free group, and (ii) a word W in the free-by-cyclic group G_Φ. It outputs a boolean I which tells whether W represents the identity.

First we freely reduce[26] the word W (Lines 19–32). Then we move all occurrences of t to the right end (Lines 49–62), then all occurrences of $\bar{t}$ to the left end (Lines 70–82). We use the M_p, the maximum number of times t and $\bar{t}$ passed an element $a_i \in \mathcal{L}_m^*$, to deduce the number of non-terminals required (Line 97) for the straight-line program P. We then define the straight-line program P as in the proof (Lines 107–131), adding the root W' (Lines 146 –150), put it in normal form (Line 153) and finally solve it as the compressed word problem in free group using Algorithm J (Line 156).

There was a small trick[27] we used to construct P. Let us denote by $P_{[i,j]}$ the submatrix of P containing the entries between i^{th} row and j^{th} row. Let us construct the first few rows of P for Φ defined by Matrix (6) in the above. The first 3 rows are easy. Then if we increase the size of every element in the first column of Φ by 3, we get $P_{[4,6]}$. Next, if we increase the size of every non-zero element in $P_{[4,6]}$ by 3, we get $P_{[7,9]}$. Hence we have

$$P_{[4,6]} = \begin{bmatrix} 4 & 1 & 2 & 3 \\ 5 & 1 & 2 & 0 \\ 6 & -1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad P_{[7,9]} = \begin{bmatrix} 7 & 4 & 5 & 6 \\ 8 & 4 & 5 & 0 \\ 9 & -4 & 0 & 0 \end{bmatrix}.$$

After that we repeat the process until we construct all the required rows in the straight-line program P.

[26] In Schleimer's paper, he did not mention the algorithm for this step, so we used the linear algorithm which we saw in the Introduction Section.

[27] The original method, which we define non-terminals one by one, is also included in the implementation as comments.

We have now finished the descriptions of all the implementations! All the implementations can be found in the Appendix[28].

8 Conclusion

In the introduction, we asked whether the word problem in a given finitely generated free group has a faster algorithm than the linear one, given that the input word has some repeated patterns. We have discovered that we can indeed find a faster algorithm if the word is defined as a compressed word. In this case, we can see that the length of the words could grow exponentially, while the time required to check if the word produced by the root is the identity will be polynomial with respect to the length of the words.

As well as free-by-cyclic groups, Schleimer [11] also applied Lohrey's theorem to prove the existence of the polynomial-time algorithm for the word problem for the automorphisms of free groups, which would be our next algorithm to implement. As an alternative to Matlab, further work could be written in Prolog language, which contains syntax that uses similar notations as the truncations and the group presentation which we saw earlier.

Returning back to the compressed word problem; a natural question one would ask is whether the polynomial-time algorithms exist for other hyperbolic groups. For the background of geometric group theory, one can consult "MA4H4 Geometric Group Theory" lecture notes [1], where Bowditch proved that the word problem for hyperbolic groups is soluble. It has been known for a long time that the word problem in hyperbolic groups is soluble in linear time using Dehn's algorithm [6]. Remarkably, Schleimer, Lohrey and Holt have proven that the compressed word problem is soluble in polynomial time in hyperbolic groups [5]. One possible goal for the future would be looking for new techniques and identifying more groups that have polynomial-time algorithms for solving (compressed) word problems. Hopefully our algorithms can be applied to more advance computational group theory and maybe unlock the hidden properties of groups and lead to further breakthroughs.

[28]Two additional implementations were included: they both take a compressed word as input, and output the word produced by the root. They were used to check the correctness of other algorithms.

9 Appendix

9.1 Algorithm A for Lemma 2.6

Matlab name	Maths name	Description
`P0` (input)	P_0	A set of productions (not necessity in normal form)
`P1` (output)	P_1	A set of production in normal form with the root of P_1 produce the same word as the root of P_0
`no_of_coln`	N_c	The number of columns in P_0
`no_of_row`	N_r	The number of rows in P_0
`cor_nt`	C	An array whose i^{th} entry, $C(i)$, stores the index of non-terminal in P_1 that produces same word as i^{th} non-terminal of P_0
`len_of_pro`	l_p	The length of production of A_p
`P0 (S, 1)`	A_s	The first entry of the s^{th} row in P_0 (which represents the s^{th} non-terminal),
`P0 (S, 2:end)`	W_s	The s^{th} row in P_0 with first entry removed (which represents the production for the s^{th} non-terminal)

```
1  function P1 = slp_to_nf(P0)
2  % slp_to_nf takes an SLP production rule matrix "P0"
3  % it outputs an SLP production rule matrix in normal form "P1 ", such that
4  % the last non-terminal of "P1 " produces the same word as last word of ...
       "P0"
5
6  no_of_row=size(P0,1);% number of non-terminals in "P0"
7  if any(P0(no_of_row,2:end))  % if the root is not sent to the empty word
8      no_of_coln=size(P0,2);  % maximum length of production rule
9      max=no_of_coln*no_of_row*2;
10     % the upper bound for the number of non-terminals in P1
11
12     P1 = zeros(max,3);
13
14     cor_nt=zeros(no_of_row,1);
15     % this array will store the indices for plain non-terminals in "P1 "
16
17     no_of_nt = 0;
18     % this integer will record current numbers of terminals
19
20     for s = 1:no_of_row
21         % go through every non-terminal in P0
22
23         if P0(s,2)==0
24             % If the non-terminal is sent to a strings of terminal chars
25
26             len_of_pro=no_of_coln-2;
```

```
27              % This will record the number of terminal char in this ...
                    production
28              % (with an initial guess, will be changed if necessary)
29
30              for i = 3:no_of_coln
31                  % *assign a new non-terminal to each terminal
32                  % (start from the 3rd elements)
33
34                  if P0(s,i)==0   % *stop if we get to a 0 (in this row);
35                      len_of_pro=i-3;
36                      break
37                  end
38                  no_of_nt = no_of_nt+1;
39                  P1(no_of_nt,:) = [no_of_nt,0,P0(s,i) ];
40              end
41
42              if len_of_pro>1
43                  % if the number of terminal char in this production is ...
                        greater
44                  % than 1, we need to introduce more dummy variable
45
46                  no_of_nt = no_of_nt+1;
47
48                  P1(no_of_nt,:) = [no_of_nt,...
49                      no_of_nt-len_of_pro,...
50                      no_of_nt-len_of_pro+1];
51                  % group first non_terminal two together
52
53                  for j=1:len_of_pro-2
54                      % group the rest
55
56                      no_of_nt = no_of_nt+1;
57                      P1(no_of_nt,:) = [no_of_nt,...
58                          no_of_nt-1,...
59                          no_of_nt-len_of_pro+1 ];
60                  end
61              end
62
63          else
64              % if the non-terminal is sent to a string of non-terminal chars
65
66              if P0(s,3)≠0 % if the length of string is greater than 1
67
68                  no_of_nt = no_of_nt+1;
69                  P1(no_of_nt,:) = ...
70                      [no_of_nt,inverse_sign(s,2),inverse_sign(s,3)];
71                  % combine first two non-terminals
72
73                  for  i=4:no_of_coln
```

```
74                      if P0(s,i)==0
75                          % *stop if we get to a 0 (in this row);
76                          break
77                      end
78
79                      no_of_nt = no_of_nt+1;
80                      P1(no_of_nt,:) = ...
81                          [no_of_nt,no_of_nt-1,inverse_sign(s,i)];
82                  end
83
84              else
85                  % if the length of string is 1
86                  no_of_nt = no_of_nt+1;
87
88                  if P0(s,2)>0
89                      P1(no_of_nt,:) = [no_of_nt,...
90                          P1(cor_nt(P0(s,2)),2), ...
91                          P1(cor_nt(P0(s,2)),3)];
92                      % copy the same production rule as
93                  else
94                      % this is when the production is a single inverse
95                      % we need to make sure we have the correct order ...
                            base on
96                      % the type of production "P0(s,2)" has
97
98                      if P1(cor_nt(-P0(s,2)),2) ==0
99                          P1(no_of_nt,:) = [no_of_nt,...
100                             0, ...
101                             -P1(cor_nt(-P0(s,2)),3)];
102                         % if it is an terminal inverse, we store it the ...
                                same
103                         % order.
104                     else
105
106                         P1(no_of_nt,:) = [no_of_nt,...
107                             -P1(cor_nt(-P0(s,2)),3), ...
108                             -P1(cor_nt(-P0(s,2)),2)];
109                         % otherwise, if it is two non-terminals, we ...
                                reverse the
110                         % order
111                     end
112                 end
113             end
114         end
115
116         cor_nt(s)=no_of_nt;   % record the corresponding non-terminal ...
                for A_s
117     end
118 else
```

```
119     P1= [1,0,0];
120     % if the root is sent to the empty word,
121
122 end
123     function plain_nt_signed=inverse_sign(s,i)
124         % nt_signed take the index (s,i),
125         % it returns "plain_nt_signed" non-terminal in P1  which
126         % will be put in the "P1 "
127
128         % this is required in later stage about compressed word.
129         % As we might have negative number in a production.
130         % and we cannot pass a negative number to the "cor_nt" array
131
132         U=cor_nt(abs(P0(s,i)));
133         plain_nt_signed = sign(P0(s,i))*U;
134     end
135
136 P1 = P1(any(P1,2),:);      % remove all the zeros rows
137
138 end
```

9.2 Algorithm B for Lemma 2.22

Matlab name	Maths name	Description
`CS0` (input)	CS_0	A set of productions in a composition system (which might contain negative integers in truncations and abbreviated truncation)
`length_arr` (output)	L	An array whose i^{th} entry stores the length of word produced by the i^{th} non-terminal in CS_0
`CS1` (output)	CS_1	A composition system that obtained by amending the negative indices in truncations and abbreviated truncation CS_0
`CS0 (S, 1)`	A_s	The first entry of the s^{th} row in CS_0 (which represents the s^{th} non-terminal),

```
1  function [length_arr,CS1] = compute_length(CS0)
2  % compute_length takes an CS production rule matrix "CS_0"
3  % it outputs (1) length_arr: an array that stores the length of words ...
       produced by every non-terminal
4  %            (2) CS_1: a matrix that obtained by amending CS_0, it ...
       produces the same word as CS_0, with no truncation and negative ...
       indices in the productions.
5
6  no_of_row=size(CS0,1);
7  length_arr =  zeros([1,no_of_row]);
8  % this array will store the length of each word produced by the ...
       non-terminals
9  % we only store the length of every non-terminal, but not their inverses.
10 % because every non-terminal(represented by a positive number i) have ...
       the same length as its inverse (represented as -i).
11 % but sometimes we might have inverses in the production
12 % When access the length_arr array, we use function "abs()" so we do ...
       not pass a negative integer as an array index
13
14 for s = 1:no_of_row
15
16     if CS0(s,2)==0 && CS0(s,7)≠0
17         % if the non-terminal char is sent to a terminal, we store 1 as ...
               its length
18         length_arr(s)=1;
19     else
20
21         % amend the  in the first truncation
22         if CS0(s,3) < 0
23             % amend negative index in first entry, i.e. change B[-i;] ...
                   to B[|B|-i:]
24             CS0(s,3) =  length_arr(abs(CS0(s,2)))+ CS0(s,3);
25         end
```

```
26
27          if CS0(s,4) < 0
28              % amend negative index in second entry, i.e. change B[-i;] ...
                    to B[|B|-i:]
29              CS0(s,4) =  length_arr(abs(CS0(s,2)))+ CS0(s,4);
30          end
31
32
33          if CS0(s,4) == inf
34              % amend the default entry, i.e. change the form from B[i:] ...
                    to B[i:|B|]
35              CS0(s,4) =  length_arr(abs(CS0(s,2)));
36          end
37
38          % similarly, amend the  in the second truncation
39          if CS0(s,6) < 0
40              % change the form from C[-i;] to C[|C|-i:]
41              CS0(s,6) =  length_arr(abs(CS0(s,5)))+ CS0(s,6);
42          end
43
44          if CS0(s,7) < 0
45              CS0(s,7) =  length_arr(abs(CS0(s,5)))+ CS0(s,7);
46          end
47
48          if CS0(s,7) == inf
49              CS0(s,7) =  length_arr(abs(CS0(s,5)));
50          end
51          % When exactly one of the truncation produce an empty word, it ...
                is more convenient that to make it be the second truncation
52          % we are allowed to swap them as empty word commute with other word
53
54          if CS0(s,4)==CS0(s,3)&&CS0(s,7)≠CS0(s,6)
55              % if the first truncation produce  empty word and the ...
                    second one doesn't, swap them
56              CS0(s,: )= CS0(s,[1, 5, 6, 7, 2, 3,  4]);
57
58          end
59
60          % finally compute the length of the non-terminal
61          length_arr(s) = CS0(s,4)-CS0(s,3)...
62              +CS0(s,7) - CS0(s,6);
63      end
64  end
65
66  % set CS_1 to be the new production rule
67  CS1=CS0;
68  end
```

9.3 Algorithm C for Lemma 2.24

Matlab name	Maths name	Description
`CS` (input)	CS	A set of productions in a composition system
`r` (input)	r	The program will output the $(r+1)^{th}$ character of root non-terminal
`char` (output)	$char$	The $(r+1)^{th}$ character of root non-terminal
`CS (S, 1)`	A_s	The first entry of the s^{th} row in CS (which represents the s^{th} non-terminal)
`i`	i	Initially $i = r+1$, we want to output the i^{th} character of the root(used to simplify the notation)
`nt`	nt	We will keep amending (nt, i), and the output character will always be the i^{th} character of A_{nt}

```
function char = get_i_th_char (CS,r)
% input: (1) a composition system CS
%        (2) position r
% get_i_th_char output char which is the (r+1)-th character of the root ...
    non-terminal

i=r+1; % simplified the notation, now we are lookinf for the i_th char ...
    of root-terminal
nt=size (CS,1); % intitial non-terminal
% we start with the tuple (nt,i),and them change them as in the proof.

[¬,CS] =compute_length (CS);
% ammend the negative indices and abbreviated truncation to the ...
    standard one

while CS (nt,2)≠0    % we stop when the non-terminal in the first entry ...
    is sent to a terminal

    if i ≤ CS (nt,4)-CS (nt,3)  % if the character we are looking for ...
        is in the first non-terminal

        % we update the (i,nt) tuple accordingly
        i = i+CS (nt,3); % (Due to trancation,) we  add the length we ...
            removed in the truncation
        nt=CS (nt,2);

    else % else, the character we are looking for is in the second ...
        non-terminal

        i = i- (CS (nt,4)-CS (nt,3)); % remove the legth contributed by ...
            the first truncation
        i=i+CS (nt,6); % (Due to trancation), we add the length we ...
            removed in the trucation
```

```
25          nt=CS (nt,5);
26      end
27  end
28  % the loop will terminate when nt is a non-terminal that sends to a ...
        terminal and at the end we output that terminal
29  char=CS (nt,7);
30
31  end
```

9.4 Algorithm D for Theorem 5.1

Matlab name	Maths name	Description
`CS` (input)	CS	A set of productions in a composition system
`SLP` (output)	SLP	A straight-line program such that the last non-terminal of SLP produces the same word as the last non-terminal of CS
`len_slp`	l_s	An array whose i^{th} entry stores the length of word produced by the i^{th} non-terminal in SLP
`len_cs`	l_c	An array whose i^{th} entry stores the length of word produced by the i^{th} non-terminal in CS
`no_of_nt`	nt	The current numbers of the non-terminals in SLP
`cor_nt`	C	The i^{th} entry of $C(i)$ stores the index of non-terminal in SLP that produces same word as i^{th} terminal of CS
`current_index`	I_c	This integer stores the index of current non-terminal
`add_new_pro`		This function takes a decorated non-terminal that is in the production of a plain non-terminal and adds required productions to SLP to construct the decorated non-terminal
`suffix_case`		This function takes a suffix decorated non-terminal and adds required productions to SLP to construct the decorated suffix non-terminal
`prefix_case`		This function takes a prefix decorated non-terminal and adds required productions to SLP to construct the decorated prefix non-terminal
`plain_nt`	p_s	The plain non-terminal which corresponds to the s^{th} non-terminal in SLP
`nt_max_pro`	M_s	The maximum number of production rules required to construct A_s which equal to $4s$ from the proof of the theorem
`reduced_index_by`	R_s	Since we do not know how many decorated non-terminals we need initially, we give an initial guess for I_c, but we need to reduce the initial guess by R_s to correct it

```
function SLP = cs_to_slp (CS)
% slp_to_nf takes an composition production rule matrix "CS"
% it outputs an straight-line production rule matrix such that
% the last non-terminal of "SLP" produces the same word as last word of ...
    "CS"
% in the comment  Y_i_j denotes the decorated non-terminal
no_of_row=size (CS,1);

max=2*no_of_row^2+2*no_of_row;
% this is an upper bound of the number of the non-terminals in CS

[len_cs,CS] = compute_length (CS);
% We retrieve "len" array which contain the length of every ...
    non-terminals, and amend the productions in CS

SLP = zeros (max,3);  % this will store all the production rules

len_slp=zeros (max,1); % this will store the length of the word in SLP

no_of_nt = 0;  % this records current numbers of non-terminals

cor_nt=zeros (no_of_row,1);  % there will be a plain non-terminal in ...
    normal form that corresponds to a non-terminal in the original slp

current_index=0; % this store the index of current non-terminals, which ...
    is different to current numbers of non-terminals
% Because built the production rule from bottom up, we only give a ...
    initial guess of the first non-terminals

for s = 1:no_of_row
    if CS (s,2)==0      % if the non-terminal has height 1, i.e it get ...
        sent to a terminal
        no_of_nt = no_of_nt+1;
        SLP (no_of_nt,:) = [no_of_nt,0,CS (s,7)];   % add only one ...
            plain non-terminal
        len_slp (no_of_nt)=1;
    else
        % when the height is more than 1
        nt_max_pro=4*s; % max number of production rules required for ...
            this non-terminal

        % we start building the bottom non-terminal, so give an initial ...
            guess of the index
        current_index= no_of_nt+nt_max_pro;
        plain_nt=current_index; % we keep a record of this as we don't ...
            know where Y_i_j and Z_k_l are going to be

        % suppose the production rule is A->B[i:j}. C[k:l}
        % we rename the variables to match the their name in the proof ...
```

```
            so the code will be easier to understand
40          B=CS (s,2);
41          i=CS (s,3);
42          j=CS (s,4);
43
44          C=CS (s,5);
45          k=CS (s,6);
46          l=CS (s,7);
47
48          Y = cor_nt (B);
49          Z = cor_nt (C);
50
51          SLP (plain_nt,1)=plain_nt;
52
53          if ¬ (i==j && k==l)  % if this non-terminal does not produce an ...
                empty word
54
55              if i==0 && j==len_cs (B) % if B is just a plain ...
                    non-terminal (i.e no truncations),we simply let  Y_i_j ...
                    to be B
56
57                  SLP (plain_nt,2)=Y;
58              else % else we need to construct Y_i_j
59                  current_index=current_index-1;
60                  SLP (plain_nt,2)=current_index;
61                  add_new_pro (current_index,Y,i,j);   % we will ...
                        construct new non-terminals by using this function
62              end
63
64              % similar for  Z_K_L
65              if k==0 && l==len_cs (C)
66                  SLP ( plain_nt,3)=cor_nt (C);
67
68              elseif k==l
69                  SLP ( plain_nt,3)=0;
70              else
71                  current_index=current_index-1;
72                  SLP (plain_nt,3)=current_index;
73                  add_new_pro (current_index,Z,k,l);
74              end
75          end
76
77          SLP (no_of_nt+1:current_index-1,:)=[]; % Since the nt_max_pro ...
                was an upper bound, we need to remove the extra 0 rows that ...
                weren't used
78
79          % Next, we also need to amend the non-terminal to correct indices
80          no_of_nt_added_for_this_nt = plain_nt-current_index+1;
81          reduced_index_by = plain_nt-no_of_nt_added_for_this_nt-no_of_nt;
```

```
82          t = SLP > no_of_nt;
83          SLP (t) = SLP (t) -reduced_index_by;
84
85          % we calculate the length for every non -terminals in SLP
86          for s1 = no_of_nt+1:no_of_nt+no_of_nt_added_for_this_nt
87              if SLP (s1,3)≠0
88                  len_slp (s1)=len_slp (SLP (s1,3));
89              end
90              if SLP (s1,2)≠0
91                  len_slp (s1)=len_slp (s1)+len_slp (SLP (s1,2));
92              end
93          end
94
95          no_of_nt=no_of_nt+no_of_nt_added_for_this_nt; % then we record ...
                the number of non-terminals we have defined so far
96      end
97
98      cor_nt (s)=no_of_nt; % and store the plain non-terminal in cor_nt ...
            as we need to refer them later
99  end
100
101     function add_new_pro (current_row,Y,i,j)
102         % the function "add_new_pro" takes the current-index for the ...
                non-terminal: current-row, a decorated non-terminal Y_i_j
103         % we use this when when Y_i_j, Z_k_l are in the production of a ...
                plain non-terminal (first split)
104         % it will add the required production rules to SLP to construct ...
                Y_i_j
105
106         % the production rule in slp for the Y is slp is Y->U.V
107         U= SLP (Y,2);
108         V= SLP (Y,3);
109
110         % if we are in subword case, and we only split it either part ...
                of U or part of V (which is another subword)
111         while (0<i && j <len_slp (Y))&& (¬ ( (i<len_slp (U) &&len_slp ...
                (U)<j)))
112             % This case will carry on until one of following happens (***)
113             % 1. subword include part of U AND part of V
114             % or 2. we require either prefix of Y
115             % or 3. we require either suffix of Y
116
117             if len_slp (U) ≤i % subword include part of V only
118
119                 % then construct Y_i_j
120                 current_index = current_index-1;
121                 SLP (current_row,:) =  [current_row,current_index,0];
122                 current_row=current_index;
123
```

```
124                 % we then define the new subword decorate non-terminal
125                 % this is why we need the length array
126                 Y=V;
127                 i=i-len_slp (U);
128                 j=j-len_slp (U);
129
130             elseif j≤len_slp (U) % similar for the subword include part ...
                    of U only
131
132                 current_index = current_index-1;
133                 SLP (current_row,:) =  [current_row,current_index,0];
134                 current_row=current_index;
135                 Y=U;
136             end
137
138             % the production rule in slp for  Y is  is Y->U.V
139
140             U= SLP (Y,2);
141             V= SLP (Y,3);
142         end
143
144         % once the while loop terminates, we know we have one of three ...
                conditions in (***)
145
146         if 0<i  && j <len_slp (Y)         % if we still have Y_i_j ...
                being subword case we define have the split case
147
148             SLP (current_row,1) =  current_row;
149
150             % we construct the prefix U_i_:
151             current_index = current_index-1;
152             U_i_colon = current_index;
153             SLP (current_row,2) =  U_i_colon;
154
155             suffix_case (U_i_colon,U,i);
156
157             % we construct the suffix V_:_j
158             current_index = current_index-1;
159
160             V_colon_j_minus_len_of_w_u = current_index;
161             SLP ( current_row,3) =  V_colon_j_minus_len_of_w_u;
162             j=j-len_slp (U);
163             prefix_case (V_colon_j_minus_len_of_w_u,V,j)
164
165         elseif 0<i       % else, if Y_i_j is suffix case
166
167             Y_i_colon= current_row;
168             suffix_case (Y_i_colon,Y,i)
169
```

```
170         else  % otherwise Y_i_j is prefix case
171
172             Y_colon_j = current_row;
173             prefix_case (Y_colon_j,Y,j)
174
175         end
176
177     end
178
179
180     function prefix_case (Y_colon_j,Y,j)
181         % the function prefix_case takes a prefix decorated nonterminal ...
                (Y,j) and its current index of non-terminal which is Y_colon_j,
182         % it will add required production to the SLP to construct Y_colon_j
183
184         U= SLP (Y,2);% the production rule in slp for the Y is slp is ...
                Y->U.V
185         V= SLP (Y,3);
186
187         while len_slp (U)≠j  % stop when we need exactly whole U
188
189             if  len_slp (U)<j    % when we require whole of U and part ...
                    of V
190
191                 current_index=current_index-1;
192                 V_colon_j_minus_len_of_w_u = current_index;
193
194                 SLP (Y_colon_j,:) = [Y_colon_j,U, ...
                        V_colon_j_minus_len_of_w_u]; % adding Y[:j] -> U. ...
                        V[:j-|w|] to the production rule
195
196                 % perform the prefix case on V_:_j-|w_u|
197                 Y_colon_j=V_colon_j_minus_len_of_w_u;
198                 Y= V;
199                 j=j-len_slp (U);
200
201             else      % otherwise we require part of U
202                 current_index=current_index-1;
203                 U_colon_j = current_index;
204
205                 SLP (Y_colon_j,:) = [Y_colon_j,U_colon_j,0];   % adding ...
                        Y[:j] -> U[:j] to the production rule
206
207                 Y_colon_j=U_colon_j;  % perform the suffix case on U_:_j
208                 Y=U;
209             end
210
211             U= SLP (Y,2);% the production rule in slp for the Y is slp ...
                    is Y->U.V
```

```
212             V= SLP (Y,3);
213
214         end
215
216         SLP (Y_colon_j,:) = [Y_colon_j,U,0];  % adding Y[i:] -> U.0 to ...
                the production rule
217
218     end
219
220
221     function suffix_case (Y_i_colon,Y,i)
222         % the function suffix_case takes a suffix decorated nonterminal ...
                Y_i and current index of non-terminal which is  Y_i_colon,
223         % it add the required production to the SLP to construct Y_i
224
225         U= SLP (Y,2);% the production rule in slp for the Y is slp is ...
                Y->U.V
226         V= SLP (Y,3);
227
228         while len_slp (U)≠i  % stop when we need exactly whole V (i.e. ...
                we removed the whole U)
229
230             if  i <len_slp (U)   % if we require part of U and whole V
231
232                 current_index=current_index-1;
233
234                 U_i_colon = current_index;
235
236                 SLP (Y_i_colon,:) = [Y_i_colon,U_i_colon,V];  % adding ...
                        Y_i_: -> U_i_:. V to the production rule
237
238                 Y_i_colon=U_i_colon;  % perform the suffix case on U_i_:
239                 Y=U;
240
241             else      % otherwise we require part of V only
242
243                 current_index=current_index-1;
244                 V_i_minus_len_of_w_u_colon = current_index;
245                 SLP (Y_i_colon,:) = ...
                        [Y_i_colon,V_i_minus_len_of_w_u_colon,0];
246
247                 % adding Y_i_: -> V_i-|w|_: to the production rule
248
249                 Y_i_colon=V_i_minus_len_of_w_u_colon;  % perform the ...
                        suffix case on V_i-|w|_:
250                 Y=V;
251                 i = i-len_slp (U);
252             end
253
```

```
254             U= SLP (Y,2);% the production rule in slp for the Y is slp ...
                    is Y->U.V
255             V= SLP (Y,3);
256         end
257
258         SLP (Y_i_colon,:) = [Y_i_colon,V,0];
259     end
260 SLP =SLP (any (SLP,2),:);% remove all the zeros rows
261
262 end
```

9.5 Algorithm E for Theorem 6.1

Matlab name	Maths name	Description
`PA` (input)	P_A	First set of productions in a straight-line program
`PX` (input)	P_X	Second set of productions in a straight-line program
`equal_or_not` (output)	E	A boolean value which tells whether the roots of P_A and P_X produce the same word
`compute_length_slp` (output)		A function takes an input straight-line program, and outputs an array which stores the length of every non-terminal.
`len_PA`	l_A	An array whose i^{th} entry stores the length of the word produced by the i^{th} non-terminal in PA
`len_PX`	l_X	An array whose i^{th} entry stores the length of the word produced by the i^{th} non-terminal in PX
`Gamma_k`	Γ_k	An array that stores all the assertions
`max_length`	M_k	The maximal length of a non-terminal in the set of assertions
`subword_k`	S_k	The set of subword assertions in Γ_k
`overlap_k`	O_k	The set of overlap assertions in Γ_k
`PO (h)`	γ_h	The h^{th} assertion

```
function len = compute_length_slp(slp)
% the function compute_length_slp takes an input SLP,
% and output an array len store the length of every non-terminal

no_of_rows =size(slp,1);
len =  zeros([1,no_of_rows]);   % initialise an array to record the ...
    length of each word
for i = 1:no_of_rows
    if slp(i,2)==0
        if slp(i,3)≠0
             len(i)=1;  % assign 1 to non-terminal that sends to ...
                 terminal charaters
        end

    else
        len(i)=len(abs(slp(i,2)))+len(abs(slp(i,3)));
    end
end
```

```
17 end
```

```
1  function equal_or_not = check_two_slps_equal (PA,PX)
2  % The function check_two_slps_equal takes two slp systems: PA and PX
3  % and output a equal_or_not, which tells whether the root of PA and PX ...
       produce the same word
4
5  m =size (PA,1);
6  n=size (PX,1);
7
8  % store the length of every non-terminal
9  len_PA=compute_length_slp (PA);
10 len_PX=compute_length_slp (PX);
11
12
13 % we might need to include the following code if we don't assume them ...
       have the same length
14 % if len_PA (m)≠len_PX (n)
15 %     equal_or_not=false;
16 % else
17
18
19 max_k=m+n+1; % the max number of time we need to transform the set of ...
       assertions
20
21 Gamma_k=zeros (max_k*4*m*n* (m+n),4); % we store a single by a row of 4 ...
       entries,
22 % the first two for the indices of non-terminals, the third is the ...
       integer i as in the proof, and the last one tells whether the index ...
       in first entry is in PA (1 for yes, -1 for no)
23
24 ass0=[m,n,0,1]; % first assertion
25 Gamma_k (1, :)=ass0;
26 number_of_ass_k =1; % record the number of assertions
27
28 k=0;
29 equal_or_not=true;
30
31 while Gamma_k (1,4) ≠0  % stop when there is no more assertions
32
33     max_length=0;
34     nt_with_max_len=0; % store the non-terminal with maximal length
35     in_PA_or_PX=0;
36
37     for h=1:number_of_ass_k  % go through every assertion to find ...
           maximal length
38         if Gamma_k (h,4)==1   % if it's in (AX) order, i.e, the first ...
               entry is in PA, and second entry is in PX
```

```
39              if len_PA (Gamma_k (h,1)) > max_length  % if the ...
                    non-terminal in first entry has a longer length than ...
                    maximal length
40                  max_length=len_PA (Gamma_k (h,1)); % store the maximal ...
                        length
41                  nt_with_max_len=Gamma_k (h,1);  % and store this terminal
42                  in_PA_or_PX=1;
43              end
44
45              if len_PX (Gamma_k (h,2)) > max_length  % if the ...
                    non-terminal in second entry has a longer length than ...
                    maximal length
46                  max_length=len_PX (Gamma_k (h,2));
47                  nt_with_max_len=Gamma_k (h,2);
48                  in_PA_or_PX=-1;
49              end
50
51          end
52
53          if Gamma_k (h,4)==-1 % if it's in XA order
54
55              if len_PA (Gamma_k (h,2)) > max_length
56                  max_length=len_PA (Gamma_k (h,2));
57                  nt_with_max_len=Gamma_k (h,2);
58                  in_PA_or_PX=1;
59              end
60
61              if len_PX (Gamma_k (h,1)) > max_length
62
63                  max_length=len_PX (Gamma_k (h,1));
64                  nt_with_max_len=Gamma_k (h,1);
65                  in_PA_or_PX=-1;
66              end
67          end
68
69      end
70
71
72      % splitting process
73      k=k+1;
74      max_ass_in_Gamma_k=k*4*m*n* (m+n); % maximal number of assertions ...
            require for Gamma^k as in the proof
75      subword_k=zeros (max_ass_in_Gamma_k,4);
76      number_of_ass=0;
77
78      overlap_k=zeros (4*m*n* (m+n),4); % we need to store the overlap ...
            assertion separately as we need to compact them later
79      num_overlapk=0;
80
```

```
81      for h=1:number_of_ass_k  % go through every assertion
82
83          if in_PA_or_PX==1 % when the maximal length non-terminal is in A
84
85              if (Gamma_k (h,1)≠nt_with_max_len && Gamma_k (h,4)==1)||...
86                      (Gamma_k (h,2)≠nt_with_max_len && Gamma_k ...
                            (h,4)==-1) % if the max non-terminal does not ...
                            appear in this assertion
87
88
89                  i=Gamma_k (h,3);
90
91                  % we check if it is an overlap case
92                  if Gamma_k (h,4)==1 &&...
93                          (0≤i&&len_PA (Gamma_k (h,1))≤...
94                          i+len_PX (Gamma_k (h,2)))... % AX order ...
                                &overlap case
95                          ||Gamma_k (h,4)==-1 &&...
96                          (0≤i&&len_PX (Gamma_k (h,1))≤...
97                          i+len_PA (Gamma_k (h,2))) % XA order &overlap case
98
99                      % if it is an overlap case, store it as an overlap ...
                            assertion
100                     num_overlapk=num_overlapk+1;
101                     overlap_k (num_overlapk,:)=Gamma_k (h,:);
102
103                 else % otherwise (subword case), we store it in the set ...
                        of general assertion
104                     number_of_ass=number_of_ass+1;
105                     subword_k (number_of_ass,:)=Gamma_k (h,:);
106                 end
107
108             else % case2: the max non-terminal does appear in this ...
                    assertion
109
110                 if Gamma_k (h,4)==1 % if  we have AX order
111                     B=Gamma_k (h,1);
112                     Y=Gamma_k (h,2);
113                     C= PA (B,2);
114                     D= PA (B,3);
115                     i=Gamma_k (h,3);
116
117                     if 0≤i&&len_PA (B)≤i+len_PX (Y)  % we have an overlap
118
119                         if len_PX (Y)==1 && len_PA (B)==1
120                             if PA (B,3)≠PX (Y,3)  % if both ...
                                    non-terminal have length 1 and are different
121                                 equal_or_not=false; % we can just end ...
                                        the program here and output false
```

```
122                             return
123                         end
124                     else
125
126                         if i<len_PA (C) % otherwise we split the ...
                                assertion as in the proof (overlap case ...
                                table)
127                             num_overlapk=num_overlapk+1;
128                             overlap_k (num_overlapk,:)=...
129                                 [C,Y,i,Gamma_k (h,4)];
130
131                             if len_PX (Y)≤ (len_PA (C)-i)+len_PA (D)
132                                 num_overlapk=num_overlapk+1;
133                                 overlap_k (num_overlapk,:)=...
134                                     [Y,D,len_PA (C)-i,-Gamma_k (h,4)];
135                             else
136                                 number_of_ass=number_of_ass+1;
137                                 subword_k (number_of_ass,:)=...
138                                     [Y,D,len_PA (C)-i,-Gamma_k (h,4)];
139                             end
140
141                         else
142                             num_overlapk=num_overlapk+1;
143                             overlap_k (num_overlapk,:)=...
144                                 [D,Y,i-len_PA (C),Gamma_k (h,4)];
145                         end
146                     end
147                 end
148
149
150                 if 0≤i&&i+len_PX (Y)<len_PA (B)  % If we have a ...
                        Subword case
151
152                     % we we split the assertion as in the proof ...
                            (subword case table)
153                     if i+len_PX (Y)<len_PA (C)
154                         number_of_ass=number_of_ass+1;
155                         subword_k (number_of_ass,:)=...
156                             [C,Y,i,Gamma_k (h,4)];
157
158                     elseif i+len_PX (Y)==len_PA (C)
159
160                         num_overlapk=num_overlapk+1;
161                         overlap_k (num_overlapk,:)=[C,Y,i,Gamma_k ...
                                (h,4)];
162
163                     elseif i<len_PA (C)&&len_PA (C)<len_PX (Y)
164
165                         num_overlapk=num_overlapk+1;
```

```
166                             overlap_k (num_overlapk,:)=[C,Y,i,Gamma_k ...
                                    (h,4)];
167                             num_overlapk=num_overlapk+1;
168                             overlap_k (num_overlapk,:)=...
169                                 [Y,D,len_PA (C)-i,-Gamma_k (h,4)];
170
171                         elseif i==len_PA (C)
172
173                             num_overlapk=num_overlapk+1;
174                             overlap_k (num_overlapk,:)=...
175                                 [Y,D,0,-Gamma_k (h,4)];
176                         else
177                             number_of_ass=number_of_ass+1;
178                             subword_k (number_of_ass,:)=...
179                                 [D,Y,i-len_PA (C),Gamma_k (h,4)];
180                         end
181                     end
182
183                 end
184
185                 if Gamma_k (h,4)==-1 % if  we have XA order, we follow ...
                        the similar steps, most of time just sway index (we ...
                        could make it concise it by doing a function )
186
187                     B=Gamma_k (h,2);
188                     Y=Gamma_k (h,1);
189                     C= PA (B,2);
190                     D= PA (B,3);
191                     i=Gamma_k (h,3);
192
193                     if len_PX (Y)==1 && len_PA (B)==1
194                         if PA (B,3)≠PX (Y,3)  % if both non-terminal ...
                                have length 1 and are different
195                             equal_or_not=false;
196                             return
197                         end
198                     else
199
200                         if len_PX (Y)≤i+len_PA (C)
201                             num_overlapk=num_overlapk+1;
202                             overlap_k (num_overlapk,:)=[Y,C,i,Gamma_k ...
                                    (h,4)];
203
204                         else
205                             number_of_ass=number_of_ass+1;
206                             subword_k (number_of_ass,:)=...
207                                 [Y,C,i,Gamma_k (h,4)];
208
209                             num_overlapk=num_overlapk+1;
```

```
                            overlap_k (num_overlapk,:)=...
                                [Y,D,i+len_PA (C),Gamma_k (h,4)];
                        end
                    end
                end
            end
        end

        if in_PA_or_PX==-1 % when the maximal length non-terminal is in ...
            X, we have the similar step as before
            % case1: the max length non-terminal does not appear in ...
                this assertion
            if (Gamma_k (h,2)≠nt_with_max_len && Gamma_k (h,4)==1)||... ...
                  % if it's in AX order
                    (Gamma_k (h,1)≠nt_with_max_len && Gamma_k ...
                        (h,4)==-1) % if it's in XA order

                i=Gamma_k (h,3);

                if Gamma_k (h,4)==1 && (0≤i...
                    &&len_PA (Gamma_k (h,1))≤...
                    i+len_PX (Gamma_k (h,2)))...% AX order &overlap case
                        ||Gamma_k (h,4)==-1 &&...
                        (0≤i&&len_PX (Gamma_k (h,1))≤...
                        i+len_PA (Gamma_k (h,2)))% XA order &overlap case

                    num_overlapk=num_overlapk+1;
                    overlap_k (num_overlapk,:)=Gamma_k (h,:);

                else
                    number_of_ass=number_of_ass+1;
                    subword_k (number_of_ass,:)=Gamma_k (h,:);
                end

            else % case2: the max non-terminal does appear in this ...
                assertion

                if Gamma_k (h,4)==-1 % if  we have XA order
                    B=Gamma_k (h,2);
                    Y=Gamma_k (h,1);

                    U= PX (Y,2);
                    V= PX (Y,3);
                    i=Gamma_k (h,3);

                    if 0≤i&&len_PX (Y)≤i+len_PA (B)  % case2.1.1: overlap
                        if len_PA (B)==1 && len_PX (Y)==1
                            if PA (B,3)≠PX (Y,3)  % if both terminal ...
                                have length 1 and are different
```

```
253                             equal_or_not=false;
254                             return
255                         end
256                     else
257                         if i<len_PX (U)
258                             num_overlapk=num_overlapk+1;
259                             overlap_k (num_overlapk,:)=...
260                                 [U,B,i,Gamma_k (h,4)];
261
262                             if 0≤len_PX (U)-i&&len_PA (B)≤...
263                                     (len_PX (U)-i)+len_PX (V)
264                                 num_overlapk=num_overlapk+1;
265                                 overlap_k (num_overlapk,:)=...
266                                     [B,V,len_PX (U)-i,-Gamma_k (h,4)];
267
268                             else
269                                 number_of_ass=number_of_ass+1;
270                                 subword_k (number_of_ass,:)=...
271                                     [B,V,len_PX (U)-i,-Gamma_k (h,4)];
272                             end
273
274                         else
275                             num_overlapk=num_overlapk+1;
276                             overlap_k (num_overlapk,:)=...
277                                 [V,B,i-len_PX (U),Gamma_k (h,4)];
278                         end
279
280                     end
281                 end
282
283                 if 0<i&&i+len_PA (B)<len_PX (Y)  % case2.1.2: Subword
284
285                     if i+len_PA (B)<len_PX (U)
286
287
288                         number_of_ass=number_of_ass+1;
289                         subword_k (number_of_ass,:)=...
290                             [U,B,i,Gamma_k (h,4)];
291                     elseif   i+len_PA (B)==len_PX (U)
292
293                         num_overlapk=num_overlapk+1;
294                         overlap_k (num_overlapk,:)=...
295                             [U,B,i,Gamma_k (h,4)];
296
297                     elseif i<len_PX (U)&&len_PX (U)<len_PA (B)
298
299                         num_overlapk=num_overlapk+1;
300                         overlap_k (num_overlapk,:)=...
301                             [U,B,i,Gamma_k (h,4)];
```

```
302
303
304                             num_overlapk=num_overlapk+1;
305                             overlap_k (num_overlapk,:)=...
306                                 [B,V,len_PX (U)-i,-Gamma_k (h,4)];
307
308
309                         elseif i==len_PX (U)
310                             num_overlapk=num_overlapk+1;
311                             overlap_k (num_overlapk,:)=...
312                                 [B,V,0,-Gamma_k (h,4)];
313
314
315                         else
316                             number_of_ass=number_of_ass+1;
317                             subword_k (number_of_ass,:)=...
318                                 [V,B,i-len_PX (U),Gamma_k (h,4)];
319                         end
320                     end
321
322                 end
323
324                 if Gamma_k (h,4)==1 % if  we have AX order
325                     B=Gamma_k (h,1);
326                     Y=Gamma_k (h,2);
327                     U= PX (Y,2);
328                     V= PX (Y,3);
329                     i=Gamma_k (h,3);
330
331                     if len_PX (Y)==1 && len_PA (B)==1
332                         if PA (B,3)≠PX (Y,3)  % if both terminal have ...
                                length 1 and are different
333                             equal_or_not=false;
334                             return
335                         end
336                     else
337                         if len_PA (B)≤i+len_PX (U)
338                             num_overlapk=num_overlapk+1;
339                             overlap_k (num_overlapk,:)=...
340                                 [B,U,i,Gamma_k (h,4)];
341                         else
342                             number_of_ass=number_of_ass+1;
343                             subword_k (number_of_ass,:)=...
344                                 [B,U,i,Gamma_k (h,4)];
345
346                             num_overlapk=num_overlapk+1;
347                             overlap_k (num_overlapk,:)=...
348                                 [B,V,i+len_PX (U),Gamma_k (h,4)];
349                         end
```

```
350                     end
351                 end
352
353
354             end
355
356         end
357
358
359     end
360
361     % remove the zero rows
362     subword_k =subword_k (any (subword_k,2),:);
363     overlap_k =overlap_k (any (overlap_k,2),:);
364
365     % Compacting
366     overlap_k=unique (overlap_k,'rows');  % remove repeated rows
367
368     overlap_k=sortrows (overlap_k,[4,1,2,3],...
369         {'descend','ascend','ascend','ascend'}); % sort the rows ...
            lexicographically
370
371     gamma_i=1; % first gamma in the first triple
372
373     while gamma_i+2≤size (overlap_k,1) && size (overlap_k,1)≥3  % ...
        repeat until we scan through all the triples
374
375         if isequal (overlap_k (gamma_i,[1,2,4]),...
376                 overlap_k (gamma_i+1,[1,2,4]))...
377                 && isequal (overlap_k (gamma_i,[1,2,4]),...
378                 overlap_k (gamma_i+2,[1,2,4]))  % check if the ...
                    non-terminals in the three assertions are the same
379
380             % we first get the length of non-terminal in the first entry
381             if overlap_k (gamma_i,4)==1
382                 len=len_PA (overlap_k (gamma_i,1));
383             else
384                 len=len_PX (overlap_k (gamma_i,1));
385             end
386
387             % same notation as is the proof so it is easy to understand
388             i= overlap_k (gamma_i,3);
389             j= overlap_k (gamma_i+1,3);
390             k=overlap_k (gamma_i+2,3);
391
392             if len≥k+j-i  % check if the triple satisfy the condition
393                 overlap_k (gamma_i+1,3)=i+gcd (i+j,i+k); % if yes, ...
                    replace the second and the third assertion
394                 overlap_k (gamma_i+2,:)=[];
```

```
395             else
396                 gamma_i=gamma_i+1; % if not we move on
397             end
398
399         else
400             gamma_i=gamma_i+1;% if this triples do not have same ...
                    non-terminates,  we move on
401         end
402
403     end
404
405     num_overlapk=size (overlap_k,1); % count the number of overlap ...
            assertions
406
407     % store both overlap assertion ans subword assertions
408     Gamma_k=zeros (max_k*4*m*n* (m+n),4);
409     Gamma_k (1:number_of_ass+num_overlapk, :)=[subword_k;overlap_k];
410     number_of_ass_k=size (Gamma_k,1);
411
412 end
413 end
```

9.6 Algorithm F for Theorem 2.25

Matlab name	Maths name	Description
`CSA` (input)	CS_A	First set of productions in a composition system.
`CSX` (input)	CS_X	Second set of productions in a composition system.
`k` (output)	k	The largest integers such that first k characters of CS_A and CS_X are the same.
`prefix_checker`		A function take an integer i', and check if the first i' characters of roots in CS_A and CS_X are the same.

```
function k = find_largest_k (CSA,CSX)
% The function find_largest_k takes two slp systems: PA and PX
% and output a k, the larges integers such that W (A)[:k] = W (X)[:k]
m =size (CSA,1);
n=size (CSX,1);

%first amend the negative indices and abbreviated productions
[len_PA,CSA] = compute_length (CSA);
[len_PX,CSX] = compute_length (CSX);

%j0 is the minimal length of w (A) and W (X)
j0=min (len_PA (m), len_PX (n));

% perform binary search
if prefix_checker (j0)==1

    k=j0;
else

    i0=0;
    in=i0;
    jn=j0;

    while in+1≠jn &&jn≠0
        i_prime=floor ( (in+jn)/2);

        if prefix_checker (i_prime)==1
            in=i_prime;
        else
            jn=i_prime;
        end

    end
```

```
34
35      k=in;
36  end
37
38      function first_ip_char_equal_or_not=prefix_checker (ip)
39
40
41          %add W (A)[:,i] as root
42          CSA_with_W_A_ip=[CSA;m+1,m,0,ip,m,0,0];
43
44          %add W (X)[:,i] as root
45          CSX_with_w_X_ip=[CSX;n+1,n,0,ip,n,0,0];
46
47
48          % convert them to straight-line programs
49          PA=cs_to_slp (CSA_with_W_A_ip);
50          PX=cs_to_slp (CSX_with_w_X_ip);
51
52          % put them in normal form
53          PN_A=slp_to_nf (PA);
54          PN_X=slp_to_nf (PX);
55
56          %check if the root produce the same word
57          first_ip_char_equal_or_not=...
58              check_two_slps_equal (PN_A,PN_X);
59      end
60
61  end
```

9.7 Algorithm G for Lemma 3.5

Matlab name	Maths name	Description
`CS0` (input)	CS_0	A set of productions in a composition system
`CS1` (output)	CS_1	The program will outputs a compressed composition system $CS1$ such that its root is the inverse of the root of $CS0$

```
function CS1=inverse_pro (CS0)
% the function inverse_pro takes an compressed composition system CS0
% and then output a compressed composition system CS1 such that its ...
    root is the inverse of the root of CS0

no_of_rows =size (CS0,1);
[len,CS0] = compute_length (CS0,no_of_rows);
CS1=zeros (no_of_rows,7); % CW1 will store the new sets of production

for i = 1:no_of_rows
    if CS0 (i,2)==0
        % if we have a non-terminal that is sent to a terminal, we ...
            simply replace the terminal by its inverse
        CS1 (i,:) =[ i,0,0,0,0,0,-CS0 (i,7)];
    else
        % otherwise we change the terminals and their truncations as in ...
            the proof
        % We avoided using negative indices, but it could also be ...
            corrected by using function compute_length
        CS1 (i,:) =[ i,...
            CS0 (i,5),...
            len (abs (CS0 (i,5)))-CS0 (i,7), ...
            len (abs (CS0 (i,5)))-CS0 (i,6),...
            CS0 (i,2),    ...
            len (abs (CS0 (i,2)))-CS0 (i,4),    ...
            len (abs (CS0 (i,2)))-CS0 (i,3) ];
    end
end

end
```

9.8 Algorithm H for Theorem 3.7

Matlab name	Maths name	Description
`P` (input)	P	A set of productions in a compressed straight-line program
`CS` (output)	CS	The s^{th} non-terminal in CCS produce the reduced word of the s^{th} non-terminal in P
`amend_inverses`		This function is a slight variation of the function `inverse_pro`; it takes a compressed composition system $CS0$ as an input, and outputs a composition system $CS1$, which stores every non-terminal in $CS0$ with its inverse immediately after it (Hence, there will not be any inverse in the production of the non-terminals in $CS1$)
`P (s, 1)`	A_s	The first entry of the s^{th} row in P (which represents the s^{th} non-terminal)
`CS (s, 1)`	X_s	The first entry of the s^{th} row in CS (which represents the s^{th} non-terminal)
`X_i_bar`	$\overline{X_i}$	The inverse of the first truncation in the production of A_s
`X_j`	X_j	The second truncation in the production of A_s

```
1  function CS1=amend_inverses (CS0)
2  % The amend_inverses is a slight variation of the function of inverse_pro
3  % It takes a compressed composition system CS0 as an input,
4  % it output a composition system CS1, which stores every non-terminal ...
       in CS0 and its inverse immediately after it
5  % There will not be any inverse in the production of the non-terminals ...
       in CS1
6
7  no_of_rows =size (CS0,1);
8  [len,CS0] = compute_length (CS0);
9
10 CS1=zeros (2*no_of_rows,7);
11
12 for i = 1:no_of_rows
13     if CS0 (i,2)==0
14         % if  we have a non-terminal is sent to a terminal,
15         % we make a copy of the non-terminal,
16         % and immediately after, we construct a copy of it's inverse
17         CS1 (2*i-1,:) =[ 2*i-1,0,0,0,0,0,CS0 (i,7)];
18         CS1 (2*i,:) =[ 2*i,0,0,0,0,0,-CS0 (i,7)];
19
20     else
21         % if  we have a non-terminal is sent to two truncations,
22         % we need to make a copy of the non-terminal, we need to change ...
```

```
            the indices, as now the previous non-terminal are in the ...
               different place
        % but we ensure that only positive indices can appear in the ...
            production rule.
        % next, we create the production of its inverse as the next ...
            non-terminal

        CS1 (2*i-1,:) =[ 2*i-1,...
            change_index (CS0 (i,2)),...
            CS0 (i,3),...
            CS0 (i,4),...
            change_index (CS0 (i,5)),...
            CS0 (i,6),CS0 (i,7)];

        CS1 (2*i,:) =[ 2*i,...
            change_index (-CS0 (i,5)),...
            len (abs (CS0 (i,5)))-CS0 (i,7), ...
            len (abs (CS0 (i,5)))-CS0 (i,6),...
            change_index (-CS0 (i,2)),    ...
            len (abs (CS0 (i,2)))-CS0 (i,4),    ...
            len (abs (CS0 (i,2)))-CS0 (i,3)   ];
    end
end

    function updated_index=change_index (ind)
        % the function change_index takes a index of terminal ind in ...
            CS0 as input
        % then it outputs the index of the corresponding non-terminal ...
            in CS1

        updated_index= abs (ind)*2; %  the corresponding index of ...
            inverse of non-terminals is twice of the index of the ...
            non-terminals
        if ind>0 % the corresponding index of of non-terminals is one ...
            less than the corresponding index of inverse of non-terminals
            updated_index=updated_index-1;
        end
    end
end
```

```
function CS=free_reduction (P)
% the function free_reduction takes a compressed straight-line program ...
    P as an input
% and output CS such that every non terminal in CCS_red produce a ...
    reduced word

no_of_rows =size (P,1);
CS = zeros (no_of_rows,7);
```

```
7
8  % we record the length so we can check if there is any truncation ...
       produce an empty word in the production
9  % we could also use this to correct negative indices appear in the ...
       algorithm
10 len_cs=zeros (no_of_rows,1);
11
12 for n = 1:no_of_rows
13
14     if P (n,2)==0 %if we have a non-terminal is sent to a terminal, we ...
           just make a copy of it in CS
15         CS (n,:) = [n,0,0,0,0,0,P (n,3)];
16         len_cs (n)=1;
17     else
18         A_i=P (n,2);
19         A_j=P (n,3);
20
21         %we double the production rules, so the last two productions ...
               are the A_i and its inverse
22         X_i_bar  = amend_inverses (CS (1:abs (A_i),:));
23
24         if A_i<0  %check if A_i was an inverse, if it was, then we want ...
               X_i_bar to be A_i, which is the second to last non-terminal, ...
               hence we need to remove the last terminal
25             X_i_bar (2*abs (A_i),:)=[];
26         end
27
28         %similar compute X_j
29         X_j= amend_inverses (CS (1:abs (A_j),:));
30         if A_j>0
31             X_j (2*A_j,:)=[];
32         end
33
34         %check how many characters cancel out
35         k=find_largest_k (X_i_bar,X_j);
36
37         len_cs (n)=len_cs (abs (A_i))+len_cs (abs (A_j))-2*k;
38
39         if len_cs (abs (A_i))-k==0  %if one of the non-terminal in the ...
               production is empty, make it our first truncation for ...
               convenience
40             CS (n,:) = [n,A_j,k,inf,A_i,0,0];
41         else
42             %Otherwise we construct the production as the proof.
43             CS (n,:) = [n,A_i,0,len_cs (abs (A_i))-k,A_j,k,inf];
44         end
45     end
46
47 end
```

9.9 Algorithm J for Theorem 3.9

Matlab name	Maths name	Description
`P` (input)	P	A straight-line program P
`s` (input)	s	An index of the non-terminal in P
`is_identity` (output)	I	A boolean value which tells whether A_s represents identity

```
function is_identity = word_prob_in_free_gp(P,s)
% The function word_prob_in_free_gp takes a straight line program P and ...
    an index of the non-terminal s
% it out put a boolean which tells whether this terminal is identity

is_identity=false;
CS_red=free_reduction(P(1:abs(s),:)); % CS_red has root that is ...
    reduction of s-th terminal in P
len=compute_length(CS_red);

if len(s)==0  % if the reduction of this non-terminal has length 0, ...
    then it is identity.
    is_identity=true;
end

end
```

9.10 Algorithm K for Theorem 4.3

Matlab name	Maths name	Description
`phi` (input)	Φ	An automorphism of a finitely generated free group
`W` (input)	W	A word in the free-by-cyclic group G_Φ
`is_identity` (output)	I	A boolean value which tells whether W represents identity
`W_p`	W'	The production for the root in the straight-line program
`no_of_a_i`	l'	The number of characters from $\mathcal{L}_m$ in W (this equals the length of)
`no_of_t_passed`	N_t	An array whose i^{th} entry stores the number of times t passed from the left and $\bar{t}$ passed from the right for i^{th} character from from $\mathcal{L}_m$ in W
`max_passed`	M_p	Maximum value in N_t, which we used to determine the number of non-terminals required in the straight-line program.
`no_of_gen`	m	The number of generated in the free group
`max_len`	M_l	The maximum length of the image word of Φ on a generator in the free group
`P`	P	The straight-line program with root being W'

```
function  is_identity = word_prob_in_free_by_cyclic_gp(phi,W)
% the function word_prob_in_free_by_cyclic_gp takes an automorphism ...
    phi, which determines a cyclic group and, a word W in the ...
    free-by-cyclic group
% the automorphism in F_m,phi ,is represented by a matrix with m rows, ...
    where the i-th row represent i-th generator of F_m and image of the ...
    phi on i-th generator, some rows might have dummy zeros at the end ...
    just to make sure all rows have the same length
% e.g. if we have F_3 = <a_1, a_2, a_3>, and phi(a_1)= a_1 a_2 a_3, ...
    phi(a_2)= a_1 a_2 and phi(a_3)= -a_1, then phi will be represent as
%       1   1   2   3
%       2   1   2   0
%       3   -1   0   0
% The word, W, denote its length as L, is represented by 2 by L matrix. ...
    where the  i-th column represent the i-th character of W.
% if the i-th character if the generator a_j, then the i-th column is ...
    going to be transpose(j,0), otherwise, if the i-th character is the ...
    generator t^p where p is 1 or -1, then the i-th column is going to ...
    be transpose(0,p)
% e.g. if we have  F_3 = <a_1, a_2, a_3>, W =  a_1 t a_2 t^(-1) a_3 t, ...
    then W will be represented as
%       1   2    0   3   0
%       0   0   -1   0   1

```

```
14 % it output an boolean value, is_identity, which tells if W is identity ...
       in the group
15
16
17
18 % We start by freely reducing W
19 current_char = 1; % first element of the pair
20
21 % we go through every single possible pair
22 while current_char <  size(W,2)
23     if W(:,current_char) == -W(:,current_char+1)
24         W(:,current_char:current_char+1)=[];
25
26         if current_char>1
27             current_char = current_char -1;  % we might have repeated ...
                   reduction at the same place
28         end
29     else
30         current_char = current_char +1;
31     end
32 end
33
34 if ¬isempty(W) % we proceed if W is not already the identity
35
36     if sum(W(2,:)) ==0  % check if the power of t adds up to 0, output ...
           FALSE if not and stop the program
37
38         % Number of character in the form of a_i in W(this equals the ...
               length of the root of the straight line program)
39         no_of_a_i=sum(W(1,:)≠0);
40
41         no_of_t_passed=zeros(1,no_of_a_i); %  the i_th entry stores the ...
               number of times t or t^(-1) have moved pass a generator a_i
42         W_p=zeros(2,no_of_a_i); %  this will store the word W_p as in ...
               the proof
43         % now we start by moving all the t's to the right
44         % (alternatively, we could, for every a_i ,sum the power of t ...
               on the left and  t^(-1) on the right, which is also a ...
               polynomial process, but moving t's one by one can reduce the ...
               number of non-terminal we need to construct )
45         % majority of the t's got eliminate by t^(-1) on the way
46         pow_of_t=0; % this store the power of t we are moving
47         no_of_ai_passed=0;  % this stores the number of a_i  t have pased
48
49         for r = 1: size(W,2)
50             if W(2,r) == -1 % if we see a t^(-1), power get reduced by 1
51                 if pow_of_t >0
52                     pow_of_t =pow_of_t-1;
53                     W(2,r) = 0;
```

```
54                  end
55              elseif W(2,r) == 1 % if we see a t, power get increased by 1
56                  pow_of_t = pow_of_t+1;
57                  W(2,r) = 0;
58              else
59                  no_of_ai_passed = no_of_ai_passed+1;
60                  no_of_t_passed(no_of_ai_passed) = pow_of_t ; % if we ...
                        see an a_i, we stores the power of t
61              end
62          end
63
64          % the above might seem to be enough given the fact that we have ...
                powers of t summing to 1, but it is not enough when the ...
                first characters from {t,t^(-1)} appeared in W is not be t
65          % we will do the same thing as earlier (more efficient to the ...
                alternative method described below)
66
67          W( :, ¬any(W,1) ) = [];
68          no_of_ai_passed=0;
69          pow_of_t_bar=0;
70          for r = size(W,2):-1:1   % first backward loop we have seen so far
71
72              if W(2,r) == -1 % if we see a t^(-1), power get increased ...
                    by -1
73                  pow_of_t_bar =pow_of_t_bar+1;
74                  W(2,r) = 0;
75                  % we will not see any t on the way
76
77              else
78                  no_of_ai_passed = no_of_ai_passed+1;  % if we see an ...
                        a_i, we stores the power of t
79                  no_of_t_passed((end+1)-no_of_ai_passed) ...
80                      = no_of_t_passed((end+1)-no_of_ai_passed) + ...
                            pow_of_t_bar ;
81              end
82          end
83
84          W( :, ¬any(W,1) ) = [];
85
86          W_p(1,:) = W(1,:);
87          W_p(2,:) = no_of_t_passed+1;
88
89          % (alternatively to the previous loop, since we have all the ...
                t's on the right end, we could just find, for every a_i,  ...
                the number of t^(-1) on the right)
90          % index_of_ai= find(W(1,:)); % we record the indices of ...
                character in the form a_i
91          % for r = 1: no_of_A
92          %    a_i=index_of_ai(r);
```

```
93          %  right_tb= sum(W(2,a_i:end)==-1);
94          % W_p(:,r) = [W(1,a_i); W_p(2,r)+right_tb+1];
95          % end
96
97          max_passed = max(W_p(2,:)); % maximum number of times of t or ...
                t^(-1) pass a character a_i
98
99          no_of_gen= size(phi,1); % number of generators in the free group
100         max_len=size(phi,2); % the length of the longest word produced ...
                by the phi acting on the generator
101         no_of_row =no_of_gen*max_passed;
102         P=zeros(no_of_row,max_len);
103
104         phi(:,1) =phi(:,1)+no_of_gen;  % add no_of_gen, the number of ...
                generators, to every element in the first column
105
106         % first we define the non-terminals that will be sent to the ...
                terminals
107         for i =1:no_of_gen
108             P(i,1)=i;
109             P(i,3)=i;
110         end
111
112         add_by=sign(phi)*no_of_gen; % the number we need to add to ...
                every entry
113         no_of_nt = no_of_gen;% this stores the number of non-terminals ...
                that have already been defined
114         % next we defined the rest required non-terminals, note that we ...
                can just increase the indices of the matrix represented by phi
115         for i =1:max_passed -1
116                P(no_of_nt+1:no_of_nt+no_of_gen,:) = ...
117                 phi + add_by*(i-1);
118
119             no_of_nt= no_of_nt+no_of_gen;
120
121             % an alternative method to construct the slp
122             %for j = 1:no_of_gen
123             %    phi_aj =nonzeros(phi(j,:));
124             %    phi_aj=phi_aj.';
125
126             %    l= length(phi_aj);
127             %     s = sign(phi_aj);
128             %     P(no_of_nt+j,1:l)=phi_aj+s*(no_of_nt - no_of_gen);
129             % end
130
131         end
132
133         % now we add the last production, which represent A ->W' in the ...
                proof
```

```
134         last_prod = zeros(1,no_of_a_i+1);
135         last_prod(1)=no_of_row+1 ;
136 
137         for i=1: no_of_a_i
138             % this corresponds to the definition of A_{i,p} in the proof
139             last_prod(i+1)= sign(W_p(1,i)) * ...
140                 (abs(W_p(1,i))+(W_p(2,i)-1)*no_of_gen);
141         end
142 
143         % before we add the last production, we need to make sure the ...
                all the rows in the resulting slp have the same size
144         max_passed=max([no_of_a_i+1,max_len]);
145 
146         P = [P, zeros(no_of_row,max_passed-max_len)];
147         last_prod = [last_prod, zeros(1,max_passed-(no_of_a_i+1))];
148 
149         % we add the last production
150         P=[P;last_prod];
151 
152         % then we put it in normal form
153         P1= slp_to_nf(P);
154 
155         % finally, check if its root produce the identity word using ...
                the function word_prob_in_free_gp
156         is_identity= word_prob_in_free_gp(P1,size(P1,1));
157 
158     else
159         is_identity = false;
160     end
161 
162 else
163     is_identity = true;
164 end
165 end
```

Algorithms for computing straight-line programs and composition systems

Finally, here are the implementations that take compressed words[29] and output the word produced by the non-terminals. They were used to check the correctness of previous algorithms.

Algorithm for computing straight-line programs

```
function  w_A_s=compute_slp (pro_r,s)
% compute_slp take a SLP production matrix "pro_r" and an integer "s"
% it outputs, w_A_s the word for the s-th non-terminal

% Define a string array that will store the words of all non-terminals
W_Ai_arr =  strings ([1,s]);

for i = 1:s
    if pro_r (i,2)==0
        W_Ai_arr (i) = strcat ("a",int2str (pro_r (i,3)));
    else

        if pro_r (i,2)>0
            s1=W_Ai_arr (pro_r (i,2));
        else
            % take the inverse
            s1=invert (char (W_Ai_arr (abs (pro_r (i,2)))));

        end

        s2='';
        if pro_r (i,3)>0

            s2=W_Ai_arr (pro_r (i,3));
        elseif pro_r (i,3)<0
            s2=invert (char (W_Ai_arr (abs (pro_r (i,3)))));

        end

        W_Ai_arr (i)=strcat (s1,s2);

    end

end

```

[29] One of them takes a straight-line program as the input while the other one takes a composition system as the input

```
36      function inverse=invert (nt)
37          % the function invert takes a word as input and output its inverse
38          % if it's negative, flip it, and put a minus sign
39          inverse=char (strcat ("a",join (  ...
40              string ( (-1)*str2double (...
41              flipud (split (nt (2:end) ,"a")))),"a" )) );
42      end
43
44  w_A_s=W_Ai_arr (s);
45
46  end
```

Algorithm for computing composition systems

```
1   function  w_A_s=compute_cs (pro_r,s)
2   % compute_cs take a SLP production matrix "pro_r" and an integer "s"
3   % it outputs w_A_s, the word for the s-th non-terminal
4
5
6   % Define a string array that will store the words of all non-terminals
7   W_Ai_arr =  strings ([1,s]);
8
9   for i = 1:s
10
11      if pro_r (i,2)==0
12
13          % no need to change anything here when it's send to terminal
14          W_Ai_arr (i) = strcat ("b",int2str (pro_r (i,7)));
15          % int2str:change variable type to string
16      else
17
18          % case where  non-terminal = non-teminal[i:j]. nonterminal[k,l]
19          s1='';
20          if pro_r (i,3)≠pro_r (i,4)
21              if pro_r (i,2)>0
22                  s1= char (W_Ai_arr (pro_r (i,2)));  % change the type ...
                        of variable so we can truncate it
23              else
24                  s1=invert (char (W_Ai_arr (abs (pro_r (i,2)))));
25              end
26
27              len = length (strfind (s1,'b'));  % the length of the ...
                    string is the number of b's appeared
28
29              m =0;
30              if pro_r (i,4) ≠ inf
```

```
31                  m = len - pro_r (i,4);   % m is the number of char we ...
                        need to remove at the end
32              end
33
34              n= pro_r (i,3); % we need to truncate first n characters in ...
                    the word
35
36              while n > 0
37                  s1= s1 (2:end);
38                  if s1 (1:1)=='b'
39                      n = n-1;                        % n is reduced by ...
                            one if we truncated the first character ...
                            completely (including indexes in Matlab)
40                  end
41              end
42
43              while m > 0
44
45                  if s1 (end)=='b'
46                      m = m-1;                        % m is reduced by ...
                            one if we truncated the last character ...
                            completely (including indexes in Matlab)
47                  end
48                  s1= s1 (1:end-1);
49              end
50          end
51
52
53          s2='';
54          if pro_r (i,6)≠pro_r (i,7)
55
56              if pro_r (i,5)>0
57                  s2= char (W_Ai_arr (pro_r (i,5)));  % change the type ...
                        of variable so we can truncate it
58              else
59                  s2=invert (char (W_Ai_arr (abs (pro_r (i,5)))));
60              end
61
62              len = length (strfind (s2,'b'));
63
64              m =0;
65              if pro_r (i,7) ≠ inf
66                  m = len - pro_r (i,7);   % m is the number of char we ...
                        need to remove at the end
67              end
68
69              n= pro_r (i,6); % we need to truncate first n characters in ...
                    the word
70
```

```
71              while n > 0
72                  s2= s2 (2:end);
73                  if s2 (1:1)=='b'
74                      n = n-1;                        % n is reduced by ...
                            one if we truncated the first character ...
                            completely (including indexes in Matlab)
75                  end
76
77              end
78
79              while m > 0
80
81                  if s2 (end)=='b'
82                      m = m-1;                        % n is reduced by ...
                            one if we truncated the first character ...
                            completely (including indexes in Matlab)
83                  end
84                  s2= s2 (1:end-1);
85              end
86
87          end
88
89          W_Ai_arr (i)=strcat (s1,s2);
90      end
91  end
92
93  w_A_s=W_Ai_arr (s);
94
95      function inverse=invert (terminal)
96                  % the function invert takes a word as input and output ...
                        its inverse
97          % if it's negative, flip it, and put a minus sign
98          inverse=char ( strcat ("b",join (  ...
99              string ( (-1)*str2double (flipud ( split (   ...
100             terminal (2:end)    ,"b")))) ,  "b" )) );   % if it's ...
                    negative, flip it
101     end
102
103 end
```

9.11 Proof for Lemma 2.15

Lemma 2.15. *Let m and n be two non-negative integers. In a computer system, the time it takes to store m is $\mathcal{O}(\log(m))$, while the time it takes to compute the addition $m+n$ is $\mathcal{O}(\log(\max\{m,n\}))$.* □

Proof. First, suppose m is non-zero. The computer stores m as its binary expansion, i.e.

if $m = b_0 + 2b_1 + 4b_2 + \cdots + 2^p b_p$ with $b_p \neq 0$, then m is stored as $b_p b_{p-1} \dots b_0$ (For example, 11 is stored as 1011). Since storing one binary digit takes 1 unit time, the time steps require to store m is

$$p + 1 \leq \log_2 2m = \log_2 m + 1 = \mathcal{O}(\log(m)).$$

Recall that $\mathcal{O}(|k|g) = O(g)$ if k is a non-zero constant, so the base of log is omitted here as changing the base of log is the same as multiplying it by a constant. In the case when $m = 0$, we require 1 time step to store it (which is a constant time). Hence the first part of lemma follows.

Now suppose one of m, n is non-zero. Without loss of granularity, m is positive. Suppose $m = b_0 + 2b_1 + 4b_2 + \cdots + 2^p b_p$ with $b_p \neq 0$ and $n = c_0 + 2c_1 + 4c_2 + \cdots + 2^q c_q$ with $c_q \neq 0$ unless possibly when $n = 0$. Then the computers compute the addition $m + n$ by writing n below m in their binary expansions and then add one column at a time (See Table 7 for the example of $11+27$). There are at most $\log_2(\max\{m, n\})+2$ columns, for every column we perform addition of at most 3 binary digits[30] and stores one binary digit, which takes 4 unit time together. Therefore in total at most $4(\log_2(\max\{m, n\}) + 2)$ time steps are required. On the other hand, when m and n are both 0, we require 3 time steps to add them (which is a constant time). Hence, the second part of the lemma follows. □

			1	0	1	1
+		1	1	0	1	1
	1	0	0	1	1	0

Table 7: Binary addition of $11 + 27 = 38$.

[30] We might have an additional 1 carried from the previous column.

Bibliography

References

[1] Brian H Bowditch. *A course on geometric group theory.* Mathematical Society of Japan, Tokyo, 2006.

[2] Max Dehn. Über unendliche diskontinuierliche gruppen. *Mathematische Annalen*, 71 (1):116–144, 1911.

[3] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for lempel-ziv encoding. In *Scandinavian Workshop on Algorithm Theory*, pages 392–403. Springer, 1996.

[4] Christian Hagenah. Gleichungen mit regulären randbedingungen über freien gruppen. 2000. PhD Thesis. University of Stuttgart.

[5] Derek Holt, Markus Lohrey, and Saul Schleimer. Compressed decision problems in hyperbolic groups. arXiv 1808.06886, 2018.

[6] GEORGE HYUN. Hyperbolicity and the word problem. 2013. University of Chicago, Research Experiences for Undergraduates.

[7] Markus Lohrey. Word problems on compressed words. In *International Colloquium on Automata, Languages, and Programming*, pages 906–918. Springer, 2004.

[8] M Lothaire. *Combinatorics on words. Encyclopedia of mathematics and its applications*, volume 17. Addison-Wesley, Reading, Mass, 1983.

[9] Roger C Lyndon and Paul E Schupp. *Combinatorial group theory.* Springer, 2015.

[10] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *European Symposium on Algorithms*, pages 460–470. Springer, 1994.

[11] Saul Schleimer. Polynomial-time word problems. *Comment. Math. Helv*, 83:741–765, 2008.