Comp + Maths
COC255
B424047

Decision Procedures for Fixed Points of Morphisms

by

Simon Yan Lung Yip

Supervisor: Dr Daniel Reidenbach

Department of Computer Science
Loughborough University

May/June 2017

Acknowledgements

I want to express my thanks to my supervisor, Dr Daniel Reidenbach who has given me countless advice and support throughout this year. I am very fortunate to have a supervisor who cared so much about my work and always giving me concise feedback to my questions.

"I had a very enjoyable time working with you on this project and I am very lucky to have you as my supervisor before I graduate. Thank you very much" - Simon Yan Lung Yip

Abstract

In this project, the main objective was to understand and implement Holub's Algorithm which is the first ever polynomial algorithm that decides for any given word, whether it is a fixed point of a non-trivial morphism or not. This report introduces the topic of morphisms and terminologies needed to understand the underlying theory of Holub's Algorithm. In Holub's Algorithm, we have identified a step where under certain conditions, it may try to handle multiple letters simultaneously by the algorithm and we have modified this step to ensure it would not happen, to make it suitable for implementation. As this modification only impose limitations on the step, the essence of the process is the same. Therefore, we continue to refer this as Holub's Algorithm. A detailed description of the algorithm was given in the report.

We have also discovered that one of the steps in Holub's Algorithm was performing unnecessary computation which made the algorithm inefficient. We proposed an improved method that replace this step which resolve this problem. A proof was written to provide the correctness of the improved method and an example was given to illustrate the improvement between the improved method and the original method. Two programs were created to implement the original algorithm which contains the original method and the improved algorithm which contains the improved method. A thorough explanation of both programs were given in the report. Various tests were carried out to compare the performance and correctness of the two programs and a detailed analysis of the results were presented in the report.

Methods of reducing the existing upper bound of the number of morphically primitive words for any fixed length n was discovered and presented near the end of the report. Using these methods, we successfully determined the total number of morphically primitive words of length 1 to 7 and reduced the upper bound for all morphically primitive words of length n.

Contents

1 Introduction

Morphisms are symbols substitutions. Given a set of symbols, a morphism h will map each symbol to a word. The special property of morphisms is that we can take a word as the input and we can map every letter in the word onto their corresponding image and concatenating the images of all letters in the word. For example, let Δ be alphabets and Δ^* the set of all words over Δ. Given a word $w = ababcc$ and a morphism $h : \Delta^* \longrightarrow \Delta^*$ where $h(a) = abc, h(b) = \varepsilon, h(c) = e$. Then h will map $ababcc$ to $abcabcee$, (i.e. $h(ababcc) = abcabcee$). Think of it as a mathematical function but instead of mapping numbers to numbers, we are mapping symbol to symbol(s). We say that a word w is a fixed point of a morphism h if $h(w) = w$, and a non-trivial morphism is a morphism that does not map each symbol to itself.

In this project, we aim to implement the first ever polynomial algorithm that decides for any given word, whether it is a fixed point of any non-trivial morphism or not. This algorithm is known as Holub's Algorithm, which was created by Štěpán Holub in 2009 [7].

For this project, the main objectives are as follows:

- To read and understand Holub's algorithm [7] and present a detailed description of the algorithm along with its underlying theory in the report.
- To learn a new programming language and implement Holub's Algorithm with the new language.
- Thoroughly explain the implementation in the report.
- To learn about set partition and implement an enumeration algorithm that generate set partitions for a given number. This is necessary to establish the correctness of the implementation.
- Performance tests will be carried out for the implementation and an analysis of its results to be included in the report.
- To learn Latex and write a full project report.

In addition to these objectives, the report also included:

- A discussion of the performance issues of the existing algorithm and a proposal of an improved version of the algorithm, which improves the performance of implementation. A proof of correctness of this improvement is to be included in the report.
- The implementation of the improved algorithm with the newly learnt programming language.
- Explanation of the new implementation in the report.
- Learn multithreaded programming and include multithreaded aspects in order to improve the run time of the implementation.
- Discover methods of reducing the existing upper bound of the total number of morphically primitive words and present them in the report. Morphically primitive words are words that are not a fixed point of any non-trivial morphism.

The report first presents the necessary definitions, theorems and terminologies in order for the reader to understand the problem that Holub's Algorithm is trying to solve. A detailed description of Holub's algorithm is presented and a detailed example to aid the understanding of the algorithm is given. Then, a discussion of the performance issues of the existing algorithm is given and we propose the improved version of the algorithm along with its proof. Next, a thorough description of both implementations is given and results of the testing is presented. Finally, we present the investigation on the total number of morphically primitive words where various methods and workings to reduce the current upper bound of estimating the total number of primitive words of length n are introduced. For readers who are interested in this topic, please read Reidenbach and Schneider's "Morphically primitive words" [4]. The report finishes with the final conclusion of this project.

It is advised to read Holub's paper where the algorithm was first presented [7] for further insights and to have some background knowledge about Formal Language Theories before reading this report.

2 Definitions

2.1 Basic Definitions

In this section we will introduce basic terms and notations which will be used to throughout the report. Some terms and notations used share similarity to those by Reidenbach and Schneider [4].

Let $\mathbb{N} := 1, 2, 3...$ be a set of natural numbers and let Δ and Σ be alphabets. A word w (over an alphabet Δ for example) is a finite sequence of symbols from Δ. The length of a word w is denoted by $|w|$ and $|w|_a$ denotes the number of occurrences of the symbol a in w. The empty word is denoted by ε which has the length of 0. The set of symbols occurring in w is denoted by $symb(w)$, e.g., for the word $w := cababc$, we have $symb(w) = \{a, b, c\}$. We use the symbol $\cdot$ to denote the concatenation of letters i.e. for the concatenation of letters a, b, we can represent this as $a \cdot b$ which give a word ab. The set of all letters over Σ is denoted by Σ^*. In other words, Σ^* is the set of all finite words that result from concatenating any letters within the alphabet Σ.

2.2 Morphisms

In this section, we will present some definitions about morphism, fixed-points and primitivity of words. A morphism is string substitution where given a set of symbols, we map each symbol with a word over another set of symbols. Not only a morphism can take individual symbols as inputs but it can also take words and we can map every letter in the word onto their corresponding image and we concatenate them together.

Definition 1. Let Δ and Σ be alphabets. A mapping $h : \Delta^* \longrightarrow \Sigma^*$ is called morphism if it is compatible with the concatenation, i.e, $\forall v, w \in \Delta^*, h(v) \cdot h(w) = h(v \cdot w)$.

Example 1. Let the morphism $h : \{a, b, c\}^* \longrightarrow \{a, b, c\}^*$ be given by

$$h(x) = \begin{cases} abb & \text{if } x = a, \\ c & \text{if } x = b, \\ \varepsilon & \text{if } x = c. \end{cases}$$

Then $h(abbca) = abbccabb$.

We say that a word w is a fixed point of a morphism h if $h(w) = w$. w is always a fixed point of the identity morphism h. The identity morphism h maps each symbol to itself.

Example 2. Let the word $w = abba$ and the morphism $h : \{a, b\}^* \longrightarrow \{a, b\}^*$ by given by

$$h(x) = \begin{cases} a & \text{if } x = a, \\ b & \text{if } x = b. \end{cases}$$

Then $h(abba) = abba$.

We can see that $h(w) = w$, so w is a fixed point of morphism h and h is the identity morphism. There will always exist an identity morphism for any given word. Therefore, identity morphisms are trivial.
In this report, we are only interested in finding nontrivial morphisms. A word w is a fixed point of a nontrivial morphism h if $h(w) = w$ and h is not the identity morphism of w.

Definition 2. A word $w \in \Sigma^*$ is a fixed point of a nontrivial morphism h if $h(w) = w$ and, for some $x \in symb(w)$, $h(x) \neq x$.

Example 3. Let the morphism $h : \{a, b, c\}^* \longrightarrow \{a, b, c\}^*$ by given by

$$h(x) = \begin{cases} ab & \text{if } x = a, \\ \varepsilon & \text{if } x = b, \\ c & \text{if } x = c. \end{cases}$$

This is a nontrivial morphism because there exist at least one mapping such that $h(x) \neq x$. The mapping are $b \to \varepsilon$ and $a \to ab$. For this example, the word $w = abcc$ is a fixed point of h, $h(abcc) = abcc$.

2.3 Morphically Primitive Words

In this section, we introduce two terms, morphically primitive words and morphically imprimitive words. They are an alternative notion of the primitivity of word which is based on morphisms rather than concatenation (or power). The terms were first introduced by Reidenbach and Schneider [4], for detailed proof and explanation, please refer to [4].

We uses the term morphically imprimitive to describe a word that is a fixed point of a nontrivial morphism.

Definition 3. A word w is morphically imprimitive if there exist morphisms h,h' and a word v such that $|v| < |w|$, $h(w) = v$ and $h'(v) = w$.

Example 4. Let the word $w = abbabb$, w is morphically imprimitive since there exist morphisms $h, h' : \{a, b\}^* \longrightarrow \{a, b\}^*$

$$h(x) = \begin{cases} a & \text{if } x = abb, \end{cases}$$

$$h'(x) = \begin{cases} abb & \text{if } x = a, \\ \varepsilon & \text{if } x = b, \end{cases}$$

Here, $h(w) = v$ give us $h(abbabb) = aa$ so $v = aa$ and $h'(v) = w$ gives us $h'(aa) = abbabb$. We can see that $|v| < |w|$. Therefore, the word w satisfies the condition for morphically imprimitive word.

Also in [4], it had shown that a word w is morphically imprimitive if there exists a nontrivial morphism h that maps $h(w) = w$. If there **does not exist** a nontrivial morphism that maps $h(w) = w$, we say that the word w is morphically primitive.

Definition 4. A word w is morphically primitive if it is not morphically imprimitive.

Example 5. The word $w = abbab$ is morphically primitive as there does not exist any nontrivial morphism $h : \{a, b\}^* \longrightarrow \{a, b\}^*$ that maps $h(w) = w$.

3 Holub's Algorithm

3.1 Terminology

In this section, we will introduce some basic definition and terminology used for Holub's Algorithm. Some definitions were obtained from Holub' paper [7] where the algorithm was first introduced. For further details and proofs of these definitions and terminology, please refer to [7].

Definition 5. A letter $a \in \Sigma$ is a mortal letter if $h(a) = \varepsilon$ for a morphism h. We denote a set of mortal letters of a morphism h by M_h.

Definition 6. A letter $b \in \Sigma$ is expanding if $h(b) = xby$ with $xy \in M_h^*$ for a morphism h. We denote a set of expanding letters of a morphism h by E_h.

Definition 7. A morphism h is a stable morphism if h maps all mortal letters to the empty word. (i.e $M_h \subseteq symb(w)$ and $\forall a \in M_h, h(a) = \varepsilon$)

Definition 8. A factor of a word w is any word v such that $w = u_1 v u_2$ for some words u_1, u_2.

Definition 9. A morphic factorisation is the factorisation of a word w induced by a morphism h. For $h(w) = w$, the morphic factorisation of w would be a k-tuple of factors $< f_1, f_2, ..., f_k >$ such that $w = f_1 f_2 ... f_k$ and we have an expanding letter $b \in symb(E_h)$ such that $h(b) = f_i$.

Theorem 1. A word w is morphically imprimitive if and only if there is a stable morphism h such that $h(w) = w$ and at least one factor in the morphic factorisation that has length of at least two. In other word, at least one factor that contains a mortal letter.

Example 6. Let a word $w = abcdcdeffabeff$, and a morphism $h : \{a, b, c, d, e, f\}^* \to \{a, b, c, d, e, f\}^*$ be given by

$$h(x) = \begin{cases} ab & \text{if } x = a, \\ \varepsilon & \text{if } x = b, \\ \varepsilon & \text{if } x = c, \\ cd & \text{if } x = d, \\ eff & \text{if } x = e, \\ \varepsilon & \text{if } x = f. \end{cases}$$

In this example, the mortal set is $\{b, c, f\}$, the expanding set is $\{a, d, e\}$ and the morphic factorisation is $< ab, cd, cd, eff, ab, eff >$. All mortal letters map to the empty word and with $h(w) = w$, there exist at least one factor in the morphic factorisation that has a length of at least two. Therefore, the word is morphically imprimitive.

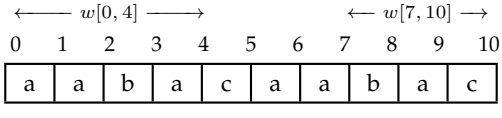
Example 7. Let a word $w = aabbcccabba$, and a morphism $h : \{a, b, c\}^* \to \{a, b, c\}^*$ be given by

$$h(x) = \begin{cases} a & \text{if } x = a, \\ b & \text{if } x = b, \\ c & \text{if } x = c. \end{cases}$$

In this example, our expanding set is $E_h = \{a, b, c\}$, and the morphic factorisation is $< a, a, b, b, c, c, c, a, b, b, a >$. There does not exist a factor in the morphic factorisation that has length of at least two. h is the identity morphism and since there does not exists any nontrivial morphism fixing this word, the word is morphically primitive. If every factor of the morphic factorisation has length one, we say that the morphic factorisation is trivial.

Definition 10. Interpositions exists in-between each letter of the word and at the boundaries of the word. Using two interpositions i, j, we can denote $w[i, j]$ to be an interval between i, j, such that $i < j$ of word w. The interval $w[i, j], j - i = 1$ denotes a letter in the word.

Example 8. In Figure 1, we see that the interpositions are labelled between the letters. Figure 1 also shows the intervals $w[0, 4]$, $w[7, 10]$ and the number of occurrence of all letters in w.



$$|w|_a = 6, |w|_b = 2, |w|_c = 2.$$

Figure 1: Showing the interpositions of a word and the number of occurrences of each letter

Definition 11 We define the set L and the set R to contain certain interpositions of the word. L denotes the set of left interpositions and R denotes the set of right interpositions.

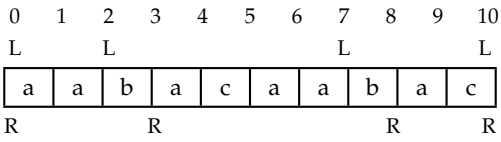


Figure 2: Showing the interposition contain set L and set R.

Definition 12. Let w be a word and x be a letter in the word. The longest common prefix of all intervals that end with x, $x \in symb(w[i,j]), i < j$ and $x = w[j, j+1]$ is denoted as l_x and the longest common suffixes of all intervals that start with x, $x \in symb(w[i,j]), i < j$ and $x = w[i-1, i]$ is denoted as r_x. The words l_x and r_x do not include the actual letter x (i.e. $x \notin symb(l_x)$ and $x \notin symb(r_x)$). Using the longest common prefix and suffixes of x, we define the neighbourhood of x as $n_x = l_x x r_x$.

Example 9. Consider the word $w = caabaabaac$. We have $symb(w) = \{a, b, c\}$.
The neighbourhood of letter a is itself, $n_a = a$ since there does not exist a longest common prefixes or suffixes for the letter a in the word. Similarly, letter c has no longest common prefixes and suffixes. Only letter b has a neighbourhood that has a length greater than 1, namely $n_b = aabaa$, with $l_b = aa$ and $r_b = aa$. Figure 3 and 4 show the neighbourhoods of $w = caabaabaac$.

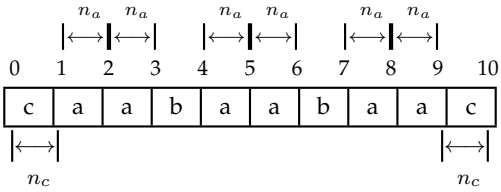


Figure 3: Showing the neighbourhoods of 'a' and 'c'

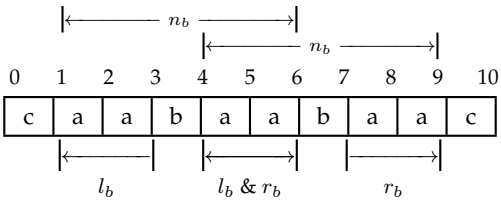


Figure 4: Showing the neighbourhoods of 'b'

Definition 13. Given an interval $w[i,j]$ where $i < j$, we define

$$\mu(w[i,j]) = \{p \mid p \in symb(w[i,j]) \wedge \forall q \in symb(w[i,j]), |w|_p \leq |w|_q\}$$

The function $\mu(w[i,j])$ will return the set of letters contained within the interval $w[i,j]$ which has the minimum number of occurrences in the word. Using the example in Figure 4, $|w|_b = 2$, $|w|_a = 6$ and $|w|_c = 2$, so $\mu(w[0,10]) = \{b, c\}$.

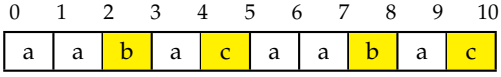


Figure 5: Showing letters with minimum occurrences of the whole word.

Definition 14. We denote $E(L,R)$ to be a set which contains only the expanding letters that can be found with available interpositions in set L and R. $E(L,R)$ is not constant in Holub's Algorithm as the interposition in set L and R are always increasing. This set is obtained by using $\mu(w[i,j])$ to find letters with the minimum occurrence and choosing the leftmost letter in $\mu(w[i,j])$ for all $i < j, i \in L, j \in R$.

Definition 15. We denote E to be a set of expanding letters obtained from the previous set of interpositions in set L and R.

In Holub's Algorithm, the set E is empty at the beginning. When $E(L,R) \neq E$, this implies that a new expanding letter is found in $E(L,R)$ using the current set L and R. We store $E = E(L,R)$ and new interpositions will be added to set L and R due to this new expanding letter (detailed process will be given in section 3.2). With new interpositions in set L and R, $E(L,R)$ might yield a new set of expanding letters such that $E(L,R) \neq E$. The algorithm continue to find new expanding letters and adding new interpositions to set L and R until no more expanding letters can be found in the word.

With the method of finding expanding letters in Holub's paper [7], it is possible to find more than one expanding letter at a time and the expanding letters found are to be handled simultaneously by the algorithm. However, it is not practical to implement this, so we have turned the process of finding expanding letter into

only finding **one new expanding letter during one iteration**. It is possible to impose this limitation because the algorithm **does not remove** any interpositions from set L and R which implies that it **does not remove** any expanding letters that have been found previously. Also, posing limitations does not change the method of finding expanding letter so we can continue to refer this as Holub's Algorithm.

In the situation where two new expanding letters a, b can be added to $E(L, R)$ in one iteration, choosing expanding letter a will not prevent b from being chosen in later iterations because we do not remove the interpositions $i \in L, j \in R$ that found the expanding letter in the first place. Therefore, we will be able to find the other expanding letter in some other iteration.

Example 10. Given a word $w = efababefcdcd$, $L = \{0, 2, 6, 8, 12\}$, $R = \{0, 1, 6, 7, 12\}$ and $E = \{e\}$:

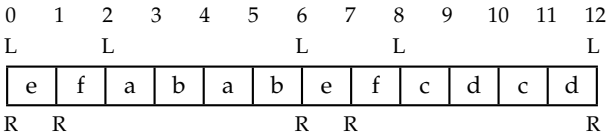


Figure 6: Showing the interpositions in set L and set R after finding expanding letter e

To find $E(L, R)$, we check all intervals $i < j, i \in L, j \in R$ and select the leftmost letter contained in $\mu(w[i, j])$. We can see that $\mu(w[2, 6])$ give a letter a and $\mu(w[8, 12])$ give a letter c. Therefore, we have $E(L, R) = \{e, a, c\}$. However, since we are processing one new expanding letter at a time, we ignore the letter c for now and only add one new expanding letter a to $E(L, R)$ so $E(L, R) = \{e, a\}$. After adding new interpositions for the new expanding letter, we get the following:

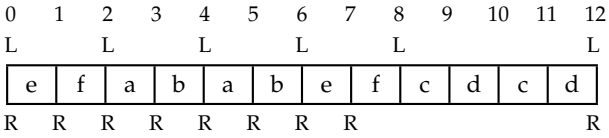


Figure 7: Showing the interpositions in set L and set R after finding expanding letter a

In Figure 7, we can see that the interval $\mu(w[8, 12])$ remains and c is be included as the new expanding letter. Processing expanding letter a first will not affect expanding letter c since the algorithm is designed to find all expanding letter and it does not matter what order we find them in.

(Note: In Holub's Algorithm, this step perform many unnecessary checks between the intervals and another method can be used for this step, please read section 3.4 for a detailed explanation.)

3.2 Main Algorithm

The version of Holub's Algorithm in this section is designed for processing one new expanding letter in one iteration unlike the algorithm in Holub's paper [7], which also allow finding multiple new expanding letters in one iteration and processing them simultaneously. The main idea of Holub's algorithm is to find all expanding letters in the word and label interpositions of the word as left or right using the expanding letters. The interpositions that we label as left of the expanding letters are stored in the set L. The positions that we label as right of the expanding letters are stored in the set R. The interpositions in set L and R will be used to find the morphic factorisation of the word.

Below is Holub's Algorithm which is illustrated in steps. Reader can refer to Holub's paper [7] for the original explanation.

1. Input the word w.
2. Add $0, |w| \in L$ and $0, |w| \in R$. (Denote this step as condition a)
3. Using set L and R and E, we iterate through **all intervals** $w[i, j]$ such $i < j, i \in L, j \in R$, find the leftmost letter of $w[i, j]$ contains in $\mu(w[i, j])$ and store this in set $E(L, R)$. For each leftmost letter, we check if the letter exists in set E. If the letter does not exists in E, this implies that $E(L, R) \neq E$ and we have a new expanding letter x. If more than one letters were found that does not exists in E, we only add the first new expanding letter to $E(L, R)$ and ignore all others.

   However, if $E(L, R) = E$, this implies that no new expanding letter can be found in the word and we have found all expanding letter. Therefore, we move to step 9.
   There exists another method of finding expanding letter which is more efficient for implementation, for more details please refer to section 3.4
4. We have confirmed that $E(L, R) \neq E$ and x is the new expanding letter. We denote $E(L, R)$ to be the new expanding set E, so $E := E(L, R)$.
5. Obtain all the intervals $w[i, i + 1]$ that only contain the expanding letter x.
   If $w[i, i + 1] = x$, store $i \in L$ and $i + 1 \in R$. (Denote this step as condition b)
6. Find all the neighbourhood of x so $n_x = w[i, j]$ for some $i, j,\ i < j$ in the word.
   Store $i \in R$ and $j \in L$. (Denote this step as condition c)
7. Now we proceed to synchronisation of neighbourhoods of expanding letters.
   In this step, we want to make sure that within the interval of the neighbourhood $n_x = w[i, j]$, the same number of interpositions and its respective positions have been added to the L and R **for all** neighbourhoods of x.
   After the neighbourhoods of x are checked, we synchronise the neighbourhoods for all other expanding letters for some $b \in E$.
   Formally put, if $w[i, j] = w[i^{'}, j^{'}] = n_x$ for some $x \in E$, then
   - For $i + k \in L$ where $i \leq i + k \leq j \Rightarrow i^{'} + k \in L$
   - and $i + k \in R$ where $i \leq i + k \leq j \Rightarrow i^{'} + k \in R$

   If new interpositions are added to either set during synchronisation of the expanding letters, we repeat the synchronisation for **all letters in** E starting from the expanding letter x until no new interpositions are added.
8. Now that all necessary interpositions are added to L and R for expanding letter x, go back Step 3 to search for expanding letters.
9. Now that all possible expanding letters have been found, we can determine whether the word is morphically primitive or morphically imprimitive.
   - If E is equal to the set of letters occurring in w, (i.e. if $E = symb(w)$), the word is morphically primitive.
   - Otherwise (i.e. if $E \neq symb(w)$). the word is morphically imprimitive.
10. End of Algorithm

After Holub's Algorithm has finished, we can use the set L and R to find the morphic factorisation of the word. This process was first introduced in Holub's paper [7] and it is analysed and adapted to the following algorithm.

1. First, we split the word w into $w = z_0a_1z_1a_2z_2...z_{k-1}a_kz_k$ where $a_1...a_k$ are the expanding letters in the word, k is the total number of expanding letters in w, so $k = |w|_E$ and $z_0...z_k$ are the mortal letters between the expanding letters in the word.
2. Loop through the expanding letters from 1 to $k-1$ where $k = |w|_E$.
   - $z_i = q_i \cdot p_{i+1}$ where q_i denote the suffix of a and p_{i+1} denote the prefix of the next a. We want to find the interposition that separate $q_i \cdot p_{i+1}$, let denote this as v.
   - To find v, we define $r \in L, w[r, r+1] = a_{i+1}$. Then we want to find v where $v \in R$ such that $v \leq r$.
   - Now, we define $w[s, s+1] = a_i$. We obtain $q_i = w[s+1, v]$ and $p_{i+1} = w[v, r]$.
   - We save q_i and p_{i+1} in some array.
   - Note that $z_0 = p_1$ which is the prefix from start of the word to the first expanding letter and $z_k = q_k$ which is the suffix from the last expanding letter to the end of the word.
3. Return each factor as $(z_0a_1q_1, p_2a_2q_2, ..., q_ka_kz_k)$

3.3 Detailed Example of using Holub's Algorithm

In this section, we will go through Holub's Algorithm step by step along with an example. Let a word $w = abccdabcd$. First, we add $0, |w|$ to L and R. This step is denoted as step 2 or condition a of the algorithm. The $()$ represents newly added interpositions.

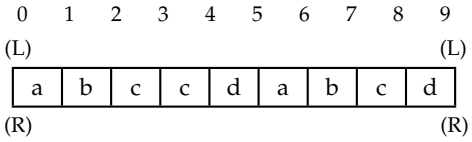


Figure 8: Showing the interpositions added to L and R after condition a.

Now, the algorithm enters into a loop. Using L and R, we check every possible intervals where $i \in L, j \in R, i < j$ and find the leftmost letter in the interval with the minimum number of occurrences in the word. Currently, since this is our first iteration of the algorithm, we do not have any letter within the set E and only one valid interval which is $w[0, 9]$. This is denoted as step 3 or condition e of the algorithm.

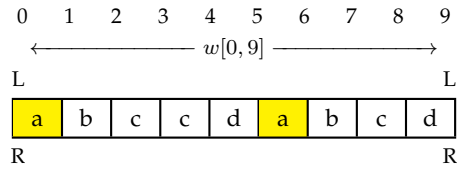


Figure 9: Highlighting letters that have leftmost occurrence in the interval $w[0, 9]$ with minimum occurrence of w.

We use the function $\mu(w[0, 9]) = \{a, b, d\}$ to obtain the set of letters with minimum occurrence in the word. The leftmost letter in $\mu(w[0, 9])$ is a. Since set E is empty, so a is the new expanding letter. The current expanding letters obtained from set L and R is $E(L, R) = \{a\}$ since $w[0, 9]$ is the only valid interval. With a as the new expanding letter, $E(L, R) \neq E$ so $E := E(L, R)$.

Next, we obtain all positions of letter a in the word so $w[i, i+1]$, and store $i \in L$ and $i+1 \in R$. This step is denoted as step 5 or condition b of the algorithm.

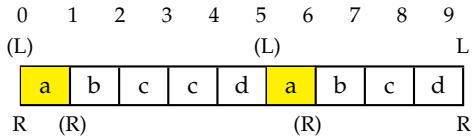


Figure 10: Showing the interpositions added to L and R after condition b.

Next, we locate all neighbourhoods of x in the word. We store $i \in R$ and $j \in L$ for $n_x = w[i, j]$. This step is denoted as step 6 or condition c of the algorithm.

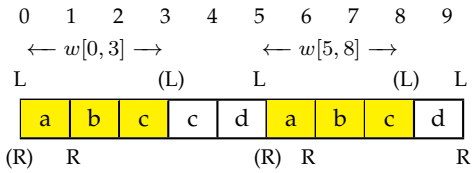


Figure 11: Showing neighbourhood of a and interpositions added after condition c.

Now, we proceed to the synchronisation of neighbourhoods of a. Neighbourhoods of a are located at $w[0,3]$ and $w[5,8]$ in the word. Using Figure 11, we can see that between neighbourhoods of n_a (i.e. $w[0,3]$ and $w[5,8]$), we have the same number and same location of interpositions between the neighbourhoods. Therefore, we do not need to add any more interpositions and the neighbourhoods are synchronised.

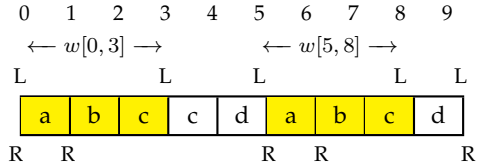


Figure 12: Showing neighbourhood of a and interpositions added after condition d.

After the synchronisation of neighbourhoods of a, we proceed to synchronising other letters in E. However, there is only one letter in E so we return to step 3 of the algorithm to search for expanding letters in the word.

We check all intervals such that $i < j, i \in L, j \in R$ and we hope to find another new expanding letter that does not exist in the current E. Using Figure 12, we can see that for most intervals where $i = 0$ or $i = 5$, the leftmost letter of these intervals contained in are $\mu(w[i,j]) = \{a\}$. We add a to $E(L,R)$, so $a \in E(L,R)$.

The noteworthy intervals are $w[3,4] = cd$ and $w[8,9] = d$. Using the formula μ we have $\mu(w[3,4]) = \mu(w[8,9]) = d$. Therefore, letter d is the leftmost letter in $w[3,4]$ and $w[8,9]$ with the minimum occurrences in w. Since d does not exist in E so the new expanding letters is d. We add d to $E(L,R)$. $E(L,R) = \{a,d\}$ and since d is a new expanding letter, we define $E := E(L,R)$. The neighbourhood of letter d is $n_d = cd$ and we add new interpositions to L and R for conditions b and c.

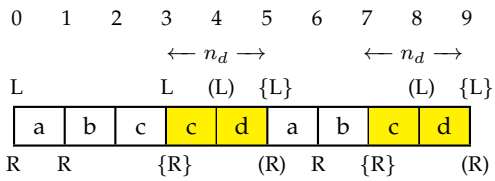


Figure 13: Showing interpositions added after condition b and c for letter d.

In Figure 13, the $()$ represents the interposition added after condition b and $\{\}$ represents the interposition added after condition c.

Now we perform neighbourhood synchronisation of letter d. The neighbourhoods of d are located at $w[3,5]$ and $w[7,9]$ in the word. From Figure 13, we can see that 2 interpositions were added to L and 2 interpositions were added to R. However, in neighbourhood $w[3,5]$, three interpositions exists in L but in neighbourhood $w[7,9]$ only two interpositions exists in L. The number of interpositions between the neighbourhoods are not the same. Therefore, we need to add one more interposition to the set L in $w[7,9]$. Since $w[3,5] = w[7,9] = n_d$ then $3 \in L \Rightarrow 7 \in L$, so interposition 7 is added to L. No new interposition is added to R as the neighbourhoods contain the same number and same location of interposition of R.

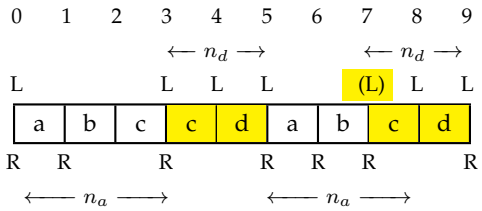


Figure 14: Showing interpositions added after synchronisation of letter d.

Now, we have the same number and same location of interpositions for all neighbourhoods of letter d. So, we proceed to synchronisation of the other letters in E. Currently, $E = \{a,d\}$. We have completed the synchronisation of letter d, so we proceed to the synchronisation of letter a. From Figure 14, we can see the neighbourhoods of n_a are over the intervals $w[0,3]$ and $w[5,8]$. There are two missing interpositions at $w[0,3]$, one in L and one in R. There is one missing interposition in R at $w[5,8]$.

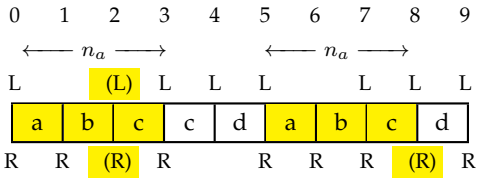


Figure 15: Showing interpositions added after synchronisation of letter a.

In Figure 15, we have added the missing interpositions for both intervals $w[0,3], w[5,8]$. Since we have added new interpositions into L and R, we need to repeat the synchronisation for all letter in E.

Starting with the latest expanding letter d, we can see that comparing the neighbourhoods of d ($w[3,5]$ and $w[7,9]$), there is a missing interposition in R in the interval $w[3,5]$.

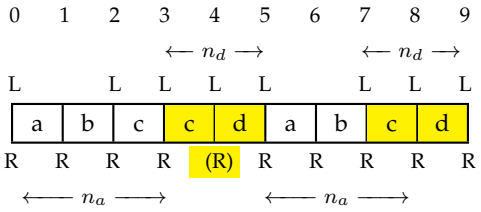
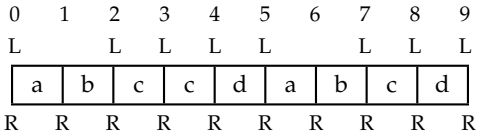


Figure 16: Showing interpositions added after synchronisation of letter d.

We continue to check the neighbourhood synchronisation of E until we have made sure no more interpositions are added to the sets. In Figure 16, we can see all neighbourhood have been synchronised, so we can go back to step 3 of the algorithm to search for expanding letters in the word.

Using the intervals from before (i.e. $\mu(w[0,3]) = \mu(w[5.8]) = \{a\}$ and $\mu(w[3,4]) = \mu(w[8,9]) = \{d\}$), we obtain $E(L,R) = \{a,d\}$. From Figure 16, we can see that $\mu(w[2,3]) = \mu(w[3,4]) = \mu(w[7,8]) = \{c\}$. Since c does not exist in E, so the new expanding letter is c and we define $E := E(L,R)$. Same as before, we add interpositions obtained from condition b and c, and proceed to synchronise the neighbourhood of all letters in E. No new interpositions are added and no more interval can be found that satisfy the conditions of step 3, so we proceed to step 9 of the algorithm.

With all the possible expanding letters found, we can determine if the word is morphically primitive or imprimitive. We have $E = \{a,c,d\}$ and since $E \neq symb(w)$, the word $w = abccdabcd$ is morphically imprimitive. Figure 17 shows the final results of set L, R, E, a morphic factorisation of the word and a morphisms h such that $h(w) = w$.



Morphic Factorisation $= (ab, c, c, d, ab, c, d)$ A possible morphism $h : \{a,b,c,d\}^* \to \{a,b,c,d\}^*$ given by

$$h(x) = \begin{cases} ab & \text{if } x = a, \\ \varepsilon & \text{if } x = b, \\ c & \text{if } x = c, \\ d & \text{if } x = d, \end{cases}$$

Figure 17: Showing interpositions and a non-trivial morphism h where the word $= abccdabcd$ is a fixed point of h.

3.4 Improvement of existing algorithm

Back in section 3.2 at step 3, we stated that there is another method for finding expanding letters which is more efficient for implementation. In this section, we discuss why the current method is inefficient and introduce a new method for finding expanding letters.

To find the set $E(L, R)$ (the current set of expanding letters), we search through all possible intervals $w[i, j]$ such that $i < j, i \in L, j \in R$. However, it is not necessary to search through all intervals. For example, if some interval $w[i, j]$ contains a letter $x \in E$, then $x \in E(L, R)$. Therefore, it is not necessary to search through any wider interval with the same i and a larger j since the letter x will be included in the wider interval and the results will be $x \in E(L, R)$, unless another expanding letter in E with a smaller occurrence is included in the interval.

Lemma 1. $\forall x \in E$, then $x \in E(L, R)$

Proof. In step 5 of Holub's Algorithm, all intervals that only contain the expanding letter x will have its start interposition stored in L and its end interposition stored in R, so if $x = w[i, i + 1]$, then $i \in L, i + 1 \in R$. Since we do not remove any interpositions added to L and R in Holub's Algorithm. $\forall x \in E$, there exists intervals such that $x = w[i, i+1], i \in L, i+1 \in R$. This means when the algorithm is searching for new expanding letters and choosing the leftmost letter contain in $\mu(w[i, i + 1])$, all letters $x \in E$ will also be in $E(L, R)$. □

Lemma 1 have shown that all expanding letters in E will also be in $E(L, R)$. This implies that if $E \neq E(L, R)$, we need to search for an interval $w[i, j]$ such that the leftmost letter in $w[i, j]$ with the minimum occurrence in the word is not in E. We can assume that $E(L, R)$ contains every letters that exists in E and focus on searching for a new expanding letter.

While these are straightforward observations, they never-the-less have an impact on how we implement the algorithm and reduce processing time. We are now about to change the algorithm quite substantially and in order to prove the correctness of the claim we need the following Lemmas from Holub's paper [7].

Lemma 2. For each letter $b \in symb(n_a)$, we have $|w|_b \geq |w|_a$.

Lemma 3. We say that letters a and b are twins if $b \in symb(n_a)$ and $a \in symb(n_b)$. We denote this as $a||b$ and has the following properties:

(1) $a||a$,

(2) If $a||b$, then $|w|_a = |w|_b$,

(3) If $|w|_a = |w|_b$ and $a \in symb(n_b)$, then $a||b$,

(4) If $a||b$, then $n_a = n_b$.

Lemma 4. Let E to be the set of expanding letters of the morphic factorisation of w, induced by a stable morphism h. Let $i < j, i \in L, j \in R$. If $a \in \mu(w[i, j])$, then either $a \in E$, or there is a letter $a' \in symb(w[i, j])$ such that $a' \in E, a||a'$ and $a \in symb(h(a'))$.

Proof. *Please refer to Holub's paper [7] for detailed proof.* □

In other words, Lemma 4 shows that either the least frequent letter occurring within the interval or one of its twins has to be expanding, not both.

Now, we have the following proposition: If there exist a new expanding letter that is not in set E, then there exist an interval that contains the new expanding letter and none of the existing expanding letters in set E are within that interval.

Proposition 1. *If $E(L, R) \neq E$, then $p \in L, q \in R$ such that $\exists x \in symb(w[p, q]) : x \in E(L, R) \wedge (\forall y \in E : y \notin symb(w[p, q]))$.*

Proof. This proposition can be proved by contradiction. Let us assume that there does not exists an interval such that the interval contains the new expanding letter x and none of the existing expanding letter y appear in the interval. This implies that the new expanding letter $x \in E(L, R)$ must only exists within an interval that contains an existing expanding letter $y \in E$. Let consider the four possible cases of this situation:

Case 1: We assume that the new expanding letter $x \in E(L, R)$ is to the right of existing expanding letter $y \in E$. Let assume that x is in the neighbourhood of y so $x \in symb(n_y)$ and there exists an interval such that it contains the letter y and x.

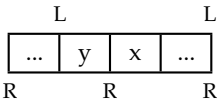


Figure 18

As we can see in Figure 18, the interpositions around letter y (i.e. first L and second R) come from step 5 of the algorithm, when $y \in E$, $w[i, i+1], i \in L, i+1 \in R$. By step 6 of the algorithm, $n_y = w[i, j], i \in R, j \in L$. The first and last interpositions are in R and L represent the neighbourhood of y. Lastly, the last interposition is also in R because we are assuming that there exists an interval which contains the letter y and x.

Since $x \in symb(n_y)$, we know that $|w|_x \geq |w|_y$ by Lemma 2. In order for $x \in E(L, R)$, we must have $|w|_x > |w|_y$ and $y \notin symb(n_x)$. Otherwise, x cannot be an expanding letter. We can prove this by contradiction.

Assume that $|w|_x = |w|_y$. Since $x \in symb(n_y)$ then by Lemma 3 property 2, we have $x||y$. Since $x||y$, this implies that $y \in symb(n_x)$ by the definition of twins in Lemma 3. If $x||y$ then by Lemma 4 only one of its twins has to be expanding. However, we want both letter to be expanding. Therefore, x cannot be an expanding letter under this condition $|w|_x = |w|_y$. Assume that $x \in symb(n_y), y \in symb(n_x)$, then that letter x and y are twins by Lemma 3 and by Lemma 3 property 2, it is impossible for this condition $|w|_x > |w|_y$ to exists. Therefore by contradiction, x is an expanding letter if $|w|_x > |w|_y$ and $y \notin symb(n_x)$.

Now that we have proved that if $x \in symb(n_y)$ and x is to the right of an existing expanding letter $y \in E$, $x \in E(L, R)$ is true under these condition $|w|_x > |w|_y$ and $y \notin symb(n_x)$. The neighbourhood of new expanding letter x does not contain the existing expanding letter y or in other words, the letter x also exists outside of the neighbourhood of y. Since x exists outside of the neighbourhood of y, this implies that there must exists an interval that does not contain $y \in E$. We can see this by using Holub's Algorithm. In step 5 (or condition c) of Holub's Algorithm $n_y = w[i, j]$ and $i \in R, j \in L$. Since $x \notin n_y$ and x is to the right of y. Therefore, the interposition $j \in L$ of n_y exists at the left of some letter x outside n_y and we can find some interposition $s \in R$ through some other neighbourhoods of $t \in E$ or $|w| \in R$ to obtain an interval where none of the existing expanding letters in E are within that interval. Therefore, this case is proved by contradiction.

Case 2: We assume that the new expanding letter $x \in E(L, R)$ is to the right of existing expanding letter $y \in E$. Let assume that x is not in the neighbourhood of y so $x \notin symb(n_y)$ and there exists an interval such that it contains the letter y and x.

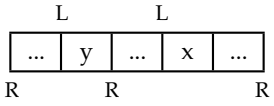


Figure 19

As we can see in Figure 19, the interpositions around letter y came from step 5 of the algorithm. The first and second last interpositions show the neighbourhood of y. Lastly, last interposition is in R because we are assuming that there exists an interval which contains the letter y and x.

Since $x \notin symb(n_y)$, we can use the part in case 1 where x exists outside of the neighbourhood of y. In step 5 (or condition c) of Holub's Algorithm, the neighbourhood of y, $n_y = w[i, j]$ and $i \in R, j \in L$. From Figure 19, we can see that there must exists an interposition $i \in L$ at the left of letter x because of the expanding letter $y \in E$. We can find some interposition $j \in R$ through some other neighbourhoods of $y \in E$ or $|w| \in R$. Therefore, there exists an interval such that $y \in E, y \notin symb(w[i, j])$ and $x \in symb(w[i, j])$ and this case is proved by contradiction.

Case 3: We assume that the new expanding letter $x \in E(L, R)$ is to the left of existing expanding letter $y \in E$. Let assume that x is in the neighbourhood of y so $x \in symb(n_y)$ and there exists an interval such that it contains the letter y and x.

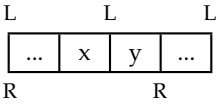


Figure 20

As we can see in Figure 20, the interpositions around letter y come from step 5 of the algorithm. The first and last interpositions show the neighbourhood of y. Lastly, first interposition is also in L because we are assuming that there exists an interval which contains the letter y and x.

Same as case 1, in order for $x \in E(L, R)$, we must have $|w|_x > |w|_y$ and $y \notin symb(n_x)$. Otherwise, x cannot be an expanding letter. We have proved this by contradiction in case 1. Since x exists outside of the neighbourhood of y, this implies that there must exists an interval that does not contain $y \in E$. As x is to the left of y and by step 5 of Holub's Algorithm, the interposition $i \in R$ of n_y exists at the right of letter x. We can find some interposition $i \in L$ through some other neighbourhoods of $t \in E$ or $0 \in L$. Therefore, there exists an interval such that $y \in E, y \notin symb(w[i, j])$ and $x \in symb(w[i, j])$ and this case is proved by contradiction.

Case 4: We assume that the new expanding letter $x \in E(L, R)$ is to the left of existing expanding letter $y \in E$. Let assume that x is not in the neighbourhood of y so $x \notin symb(n_y)$ and there exists an interval such that it contains the letter y and x.

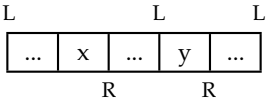


Figure 21

As we can see in Figure 21, the interpositions around letter y come from step 5 of the algorithm. The second and last interpositions show the neighbourhood of y. Lastly, first interposition is in L because we are assuming that there exists an interval which contains the letter y and x.

This is same as case 3, since x exists outside of the neighbourhood of y, this implies that there must exists an interval that does not contain $y \in E$. As x is to the left of y and by step 5 of Holub's Algorithm, the interposition $i \in R$ of n_y exists at the right of letter x. We can find some interposition $i \in L$ through some other neighbourhoods of $t \in E$ or $0 \in L$. Therefore, there exists an interval such that $y \in E, y \notin symb(w[i,j])$ and $x \in symb(w[i,j])$ and this case is proved by contradiction.

These four cases have been proven by contradiction. Therefore, Proposition 1 is true. □

From the proposition and lemmas, the new method of finding expanding letter is as follows:

$\forall x \in E$, then x is also in $E(L,R)$. If $E(L,R) \neq E$, we need to find a new expanding letter that is not in E. We search for an interval $w[i,j]$ such that $w[i,j]$ does not contain any letters in E. When we found an interval $w[p,q]$ contain letters in E, we ignore that interval and any wider intervals with the same p and larger q as we would not find any new expanding letters with intervals that contain letters in E. The leftmost letter in $w[i,j]$ with the minimum occurrences in w will be the new expanding letter in $E(L,R)$. If there does not exists such an interval $w[i,j]$, then $E(L,R) = E$ and we have obtained all expanding letters in the given word.

Example 11. In this example, we compare Holub's original method of finding the expanding letter and our new method of finding expanding letter. Using Figure 12, $w = abccdabcd$. With Holub's original method of finding the expanding letter, it will first obtain the following intervals:

1. $w[0,1] = a, w[0,5] = abccd, w[0,6] = abccda, w[0,9] = abccdabcd$,
2. $w[3,5] = cd, w[3,6] = cda, w[3,9] = cdabcd$,
3. $w[5,6] = a, w[5,9] = abcd$
4. $w[8,9] = d$.

Next, we have to use the formula μ on all these intervals to find letters with the leftmost occurrence each interval. However, we can see that a lot of these intervals will produce the same leftmost letter. $E(L,R) = \{a,d\}$. With our method, we only need to search through these intervals:

1. $w[0,1] = a$,
2. $w[3,5] = cd$.

As we can see, we have decreased the number of intervals we have to check compared to Holub's original method. First, we check the interval $w[0,1]$. $w[0,1]$ contains a letter in E, so we ignore this interval and all other wider intervals that start with the interposition 0. Next, we check the interval $w[3,5]$ and found that this interval does not contain any letters in E. The leftmost letter of the interval $w[3,5]$ from the set $\mu(w[3,5])$ is d. The letter d is the new expanding letter and $E(L,R) = \{a,d\}$.

We can clearly see from our example that new method performs less computation then Holub's original method of finding expanding letters. This suggest that the improved method will have a better performance compared to the original method when they are implemented in computer programs.

4 Implementation

4.1 About the Program

The implementation of the algorithm is written using the programming language C and there are two versions of the implementation. One version contains Holub's Algorithm with the original method of finding expanding letters which we denote as the Original Version. The other version contains Holub's Algorithm with the improved method of finding expanding letters which we denote as the Improved Version.

Both versions only differ in finding the expanding letters, so all other functionality of both program are the same. In both versions, we only have a set E to store the expanding letter. The set E represents both $E(L, R)$ and E because we do not remove expanding letters or remove any interpositions in set L and R, so it is more logical to only have E and add the new expanding letter to the set rather replacing E with $E(L, R)$. If there does not exists an leftmost letter x in some interval $\mu(w[i, j])$ such that $x \notin E$ then this simply implies that $E(L, R) = E$, which means all expanding letters have been found.

In the program, there are several core functionalities (options) available.

1. Enter a single word and run through Holub's Algorithm with that word.
2. Insert a single file and run through Holub's Algorithm with the strings in the file.
3. Insert multiple files and run through Holub's Algorithm with these files.
4. A set partition generator which will generate all partitions of length n then run Holub's Algorithm with all partitions to find the total number of morphically primitive words for words of length n (Detailed explanation given in Section 5).

Within these core functions, there are some other functionalities for the user to choose.

1. When inserting files, the user can choose either to print the result of Holub's Algorthm for each individual string to the screen.
2. The program offers to save the results of Holub's Algorithm to a text file (or several text files depending on the size of the result).
3. The user can choose what output to be saved into the text file.

These minor functionalities are not available for option 1 as the user only inputs a single word.
For all options except option 1, the program will display the total number of strings that Holub's Algorithm went through, the total number of morphically primitive word and the total number of morphically imprimitive word.
At the end of each options, the user will be given an option to restart the program or to end the program.

4.2 Functionality

In this section, we will go through each function of the program along with some explanation of the code.

4.2.1 Option 1 - Single word check

For the first option, the user can enter a word of maximum length of 99. Then the program will run Holub's function which is the implementation of Holub's algorithm with that word (details about Holub's function will be explained in later sections).
Holub's function will output a number 0 or 1. If the output is 0, then the program will print out "the word is morphically primitive" otherwise it will print out "the word is morphically imprimitive". Also, Holub's function has the option to print or not print out the results depending on the input given to the function.
For this option, the 'print' input to the function is fixed at '-1' which will print out the word, set L and set R in the same format as the figures in this report, and a possible morphic factorisation for that word. The program will prompt the user to restart or end the program at the end.

4.2.2 Option 2 - Insert a single file

For the second option, the user will need to enter the name of the text file to be read by the program. If the text file is in another folder, use the syntax "./folder_name/filename.txt".
The program will ask the user whether to display each individual results of the algorithm. If the user choose to display the results, it will print out the morphic factorisation of each string in the text document to the screen. Next, the program will ask if the user wants to save the results from Holub's function into a text file. If they choose to save, the program will prompt the user to enter a name for the text file, then it will ask the user to

select which results from Holub's function to be saved.
The user has the following options:

1. The user can save all primitive words from the file.
2. The user can save all imprimitive words from the file.
3. The user can save both primitive and imprimitive words. If this option is selected, the user will need to choose to whether to label each word as "primitive" or "imprimitive".

Now, the program will read the text file and run Holub's function for each string in the file. (Note that if the file does not exist within the directory it will return an error)
An indicator will keep track of the number of primitive and imprimitive words in the file. If the total number of processed word is divisible by 20000 (i.e. mod 20000 = 0), the program will print out the current number of word it has went through. This is necessary in order to keep track of the data if we are processing a large file.
Finally, after the file has been read, it will print out the total number of words, the number of primitive and imprimitive words in that file. Lastly, it will prompt the user to restart or end the program.

4.2.3 Option 3 - Insert multiple files

Similar to option 2, option 3 reads strings from text file but option 3 allows the user to insert multiple files to the program. The user can select up to 18 text files. However, the save functionality is not available in this option because the resulting text file will be too large to open.
If the total number of processed words is divisible by 20000, it will print the current total number of word. After the program has read all the files, it will print out the total number of all strings read from all file, the number of primitive and imprimitive words of all files. Lastly, it will prompt the user to restart or end the program.

4.2.4 Option 4 - Finding all primitive word of length n

Set partitions are used to generate all words of length n in canonical form. Using these generated words, we can find the total number of morphically primitive word of length n by running Holub's function and work out the total number of morphically primitive word of length n (Detailed explanation on set partitions and canonical form will be given in Section 5). For this option, there are two set partition algorithms for generating set partitions available for the user. The first algorithm is taken from book, "Combinatorial Algorithms" by Albert Nijenhuis and Herbert S. Wilf [3]. The second algorithm is taken from an paper by M. C. ER [2] which is the fastest set partition generating algorithm in existence.
For each word generated by the set partition algorithm, it will run Holub's function with the generated word. When the set partition algorithm is finished, it will print out the number of words set partition had generated, the number of primitive and imprimitive words. Lastly, it will prompt the user to restart or end the program.

4.3 Program Code for Holubs' Algorithm

In the program, Holub's Algorithm is named as Holub's function. In the function, we first start initialising and collecting data from the given word. This way, it will make the implementation faster as the program only need to refer to the memory for information about the word.
After the data collection is completed, the function move onto running Holub's algorithm. Finally, after the function has finished running the algorithm, the program use the results (i.e set E, set L and set R) to find the morphic factorisation of the word and obtain a possible morphism h for the given word. Finding the morphic factorisation will occur if the user wanted to display the results onto the screen. Otherwise, this process is skipped in order to save computation time.

Detailed information about the code and data structure for data collecting, algorithm and morphic factorisation of Holub's function can be found in section 4.3.1.

4.3.1 Data Collection of the word

When the program calls Holub's function $int\ holubAlg(char*str, intl, intp)$ (inputs are word w, length of word $|w|$ and print variable), it will collect all of the information from w before running the actual algorithm. We store the information of w into an array of structures.

```
struct data {
  char letter;    //Store a letter of the word
  int numoccur;   //Store the number of occurrences in the word
  int* positions;   //Store the positions where the letter appear in the word
  int leftlimit;    //Store the length of the longest common prefix of the letter
  int rightlimit;   //Store the length of the longest common suffix of the letter
  int inE;      //Store the elemental position of this letter that exist in the array called Eset
};
```

We can see the format of the stored information above. Please note that positions and interpositions are different. For example, position 0 of the word = a and $w[0,1] = a$. The reason we use positions because we use arrays in the programs and arrays uses positions to link to data in the program. Recall **Defintion 12**, the neighbourhood of a letter x is $n_x = l_x x r_x$. By storing the length of l_x and r_x, we can determine the intervals of all neighbourhoods of x in w.

We define the array of structure as $letterdata$ in the function (we will refer to this as $letterdata\ array$). To add data to the array, we define the function $addnewletter$.

```
void addnewletter(struct data *info, char word, int location, int position){
  info[location].letter = word;
  info[location].numoccur = 1;
  info[location].positions = (int*)malloc(sizeof(int));
  info[location].positions[0] = position;
  info[location].inE = -1;  //Default -1, letter not in set E.
}
```

This function will add the default information about the letter to the array. The inputs are: An array of $data$, a letter, the element of array to be added (i.e. $info[0]$ is element 0 of the array) and the first position of the letter in word. The memory of array of positions ($info[location].position$) is set as dynamic because the size of the array is depended on the number of occurrences of the letter in the word. Dynamic memory allocation allows the program freely to allocate the amount of memory needed for each letter in the structure array and the memory is freed when the function ends.

The code below shows that beginning of data collection process.

```
  int uniq = 1; //initial letter count = 1 the initial one we put in
  struct data *letterdata = malloc(2*sizeof(struct data));  //dynamic data
  addnewletter(letterdata, word[0], 0, 0);  //add first letter in word, in array 0, position 0

  for(int i = 1; i < lenw; i++) {
    for(int j = 0; j < uniq; j++){
      if(letterdata[j].letter == word[i]){  //If the letter exist in the struct data.
        letterdata[j].numoccur++;
        int temp = letterdata[j].numoccur;
        //Increase the memory size for position array
        letterdata[j].positions = realloc(letterdata[j].positions, temp * sizeof(int));
        letterdata[j].positions[temp-1] = i; //Store next position of the word (i)
        break;
      }
      else if( j == uniq-1){
        uniq++;   //Increase the num of unqiue letter
        letterdata = realloc(letterdata, uniq * sizeof(struct data) ); //Increase the memory
        addnewletter(letterdata, word[i], j+1, i);  //Add new letter to letterdata
        break;
      }
    }
  }
```

The array of structure in the code is defined as a dynamic data which allows the program to freely increase its memory size as needed. The data collection utilise two $for\ loops$, the first loop iterate through the word and the second loop iterate through the $letterdata\ array$. Due to the fact that $letterdata\ array$ is empty, we need to add the first letter to the $letterdata\ array$ outside both loops so $letterdata\ array$ has the initial size of 1 which is denoted by the variable $uniq$.

Within the second loop, it checks whether the current letter exists in letterdata array. If the letter exists, we need to increase the number of occurrences, adjust the memory for the $position\ array$ of the letter and add position i into $letterdata\ array$. If the letter does not exist in $letterdataarray$, which mean we have encounter

a new letter in the word. Therefore, we increase the variable $uniq$ and the memory of $letterdataarray$ and add the new letter to $letterdataarray$.

Next, we collect the length of the longest common prefix and suffix of each letter. To do this, we iterate the $letterdata\ array$ with a $for\ loop$. In the loop, we get the number of occurrences for the current letter and use the pointer $getposition$ to get the $position\ array$ for the current letter x of the $letterdata\ array$. If there is only one occurrence of the letter, then the neighbourhood of x is the whole word. If there is more than one occurrence, we proceed to find the longest common prefix (stored as $leftlimit$) and longest common suffix (stored as $rightlimit$) of the letter. The process of finding both $leftlimit$ and $rightlimit$ are almost the same.

The code below shows how the program finds the $leftlimit$.

```
while (leftfind == FALSE){  //Finding the correct left neighbourhood
  for (int k = 0; k < getnumocc-1; k++){
    int s = (getposition[k]-getleftlimit);    //Neigbourhood of first occurrence of letter
    int t = (getposition[k+1]-(getleftlimit)); //Neigbourhood of second occurrence of letter
    if(s < 0){  //If the start letter went beyond the beginning of the word
      leftfind = TRUE;
      break;
    }
    if(word[s] != word[t]){ //When letters the between the two neigbourhood are not the same
      leftfind = TRUE;
      break;
    }
  }
  if(leftfind == FALSE){  //Letters between the two neigbourhood are the same
    getleftlimit++;   //Increase the size of left neighbourhood, checking the next letter.
  }
}
```

To find the $leftlimit$, we iterate through the $position\ array$ in the $letterdata\ array$. For each iteration, we get the current position s and the next position t and subtract these value by $getleftlimit$. The variable $getleftlimit$ is the number of positions to the left of the letter x that we are checking. Using s and t, we will be able to compare the letter on the left of both positions by some number $getleftlimit$. At the end of the iteration, we increase the value of $getleftlimit$ if we cannot find a letter that is different from each other or until value of s goes out of bounds for the word.

The process for finding $rightlimit$ is the same as finding $leftlimit$. The only different is that instead of iterate from the first position, we from iterate backward from the last position to the first position in the $position\ array$. Also, instead of checking if the start letter went beyond the beginning of the word, we check if the last letter went beyond the length of the word.

When we have obtained a value for $getleftlimit$ and $getrightlimit$, we subtract them by 1 and add them to the $letterdata\ array$ for that letter. Reset the values and continue finding the '$leftlimit$' and '$rightlimit$' for all letter in $letterdata\ array$.

4.3.2 Holub's Algorithm

With the data collection process completed, we move on to the algorithm. First, we introduce new structures which are used in this part of the function.

```
struct Estr {
  char letter;  //Expanding letter
  int datapos;  //Stores the elemental position of the letter in the letterdata array.
  int datastart;
  int dataend;
};
struct LRdata{
  int L;
  int R;
};...
int Lset[lenw+1]; int Rset[lenw+1]; //Both array has the size of the word + 1 (just in case)
int Lsize = 2;  int Rsize = 2; //Since the arrays are not filled up, we create limiters for them.
```

The $Estr$ is used to store information about the expanding letter. The variable $datapos$, $datastart$ and $dataend$ variables will be used later for morphic factorisation. An array of $Estr$ (denoted as $Eset$ in the program) is the set E of the algorithm. The $LRdata$ structure array (denoted as $newsetdata$ in the program) list all interpositions in the word and the two variables L and R act as an indicator which tell the program whether the interposition is added to set L and set R. The arrays $Lset$ and $Rset$ stores the actual value of interpositions in the set L and R respectively.

The code below adds $0, |w|$ to the sets. This step is denote as condition a or step 2 in the algorithm.

```
//Condition a
Lset[0] = 0;   Rset[0] = 0;
Lset[1] = lenw;    Rset[1] = lenw;
newsetdata[0].L = 1;   newsetdata[0].R = 1;
newsetdata[lenw].L = 1;    newsetdata[lenw].R = 1;
```

Now, we move on to finding an expanding letter. This process is iterative so it is implemented within a *while loop* which will endlessly repeat step 3 to 9 of the algorithm until all possible expanding letters are found and we have determined whether the given word is morphically primitive or imprimitive. We denote this *while loop* as the *'core' while loop*.

In this part of the function, we will use the $Lset$, $Rset$ array (which is set L and set R respectively) to find intervals in the word. We initialise two variables $Lpoint$ and $Rpoint$ to indicate the position in the $Lset$ and $Rset$ array. We initialise some three more variables, $minocc$, $minchar$ and loc. (We will explain the use of these variables as we go along).

In the Original version, we introduce three more variables, *'ctempocc'*, *'ctempchar'* and *'ctemploc'* to temporarily store data during the search. At the beginning of the *'core' while loop*, we reset the value of the variables to ensure the data from previous iteration is not carried over. Next, we have *while loop* with the condition, $Lpoint < Lsize - 1$ (the value of *'Lpoint'* is less then the size of the $Lset$ array). This *while loop* effectively iterates through $Lset$ array (set L). Let denote this as the *L while loop*.

Within the *L while loop*, first we check if the value of interposition of $Lset$ array is greater than the value of interposition of $Rset$ array. If this is true, this violate the condition $i < j, i \in L, j \in R$ of the algorithm. We increase the value of *'Rpoint'* which will get us the next available interposition in $Rset$.

```
while(Lpoint < Lsize-1){  //Lsize-1 becuase there's no need to check the last L of the set
  if(Lset[Lpoint] >= Rset[Rpoint]){ //If L > R, we need to increase R so that R > L.
    Rpoint = Rpoint + 1;
  }
...
```

If the condition is false, we denote the interpositions between $Lset[Lpoint]$ and $Rset[Rpoint]$ to be a valid interval and iterate through each letter in the interval to find the leftmost letter with the minimum occurrence in the word (effectively performing $\mu(w[i, j]$ where $i = Lset[Lpoint], j = Rset[Rpoint])$. The leftmost letter with the minimum occurrence in the word of the interval $w[i, j]$ is stored in *ctempchar* with the value of its occurrences stored in *ctempocc* and its elemental position in the *letterdata array* stored as *ctemploc*.

```
...
else {
  int startpoint = Lset[Lpoint];    //Startpoint of the interval in the word.
  int intervalsize = Rset[Rpoint];  //Endpoint of the interval in the word.
  for (int i = startpoint; i < intervalsize; i++){
    char tempchar = word[i];
    int temploc = 0;
    for(int k = 0; k < uniq; k++){
      if( tempchar == letterdata[k].letter ){
        temploc = k;  //Find the location of the letter in the struct array
        break;
      }
    }
  int tempocc = letterdata[temploc].numoccur;    //Get the num of occurence for that letter
  if( ctempocc > tempocc){              //Compare it to the previous minimum occurrence
    ctempocc = tempocc;               //Update the new minimum occuring letter in this interval
    ctempchar = tempchar;
    ctemploc = temploc;
  }
}
```

Next, the program will decide if the leftmost letter is the new expanding letter we are looking for. If this is the first iteration (the size of E is zero), we temporarily store the values of variables into $minchar$, $minocc$ and loc and increase the value of $intervalfind$ by 1 and break out of the *L while loop*. If $E \neq \varepsilon$, we iterate through $Eset$ and check if the leftmost letter is in the array $Eset$. If the letter is in $Eset$, we reset the *ctempocc* and continue onto the next available interval by increasing *'Rpoint'*. Unless $Rpoint$ reached the last element of $Rset$, we reset $Rpoint$ to zero and increase $Lpoint$ by 1.

After *L while loop* is completed, if the value of $intervalfind$ equal to zero, this implies that the loop did not find an new expanding letter for E. Therefore, we break out of the *'core' while loop* and return Holub's function as 1 which implies that the word is morphically imprimitive.

In the Improved version, we initialise the boolean variables $Efind$ and $Letterfind$. They are indicators which we will use in this part of the function. Similar to the Original version, we check if the value of interposition in $Lset$ array is greater than the value of interposition of $Rset$ array then we increase $Rpoint$ by 1 to get the next interposition in the $Rset$ array. Otherwise, we iteration through each letter in the interval where $i = Lset[Lpoint], j = Rset[Rpoint]$ and check if each the letter is in the array $Eset$ (set E).

If the letter is in E, we set $Efind$ to true. $Efind$ is an indicators for when we have located an expanding letter in the interval. If $Efind$ is true, we set the boolean value $Letterfind$ to false, reset the $minocc$ to maximum and stop iterating through the letters in the interval. $Letterfind$ is the indicator to check if we found a letter that is not in E in the interval and $minocc$ stores the minimum occurrence of the letter.

```
else {
  int startpoint = Lset[Lpoint];     //Startpoint of the interval in the word.
  int intervalsize = Rset[Rpoint];  //Endpoint of the interval in the word.
  for (int i = startpoint; i < intervalsize; i++){  //Going through each letter in the interval
    checking for an existing expanding letter.
    for(int j = 0; j < Esize; j++){
      if(word[i] == Eset[j].letter){        //Check if the letter exist in the expanding set
        Efind = TRUE;
        break;
      }
    }
  }
  if(Efind == TRUE){       //Found expanding letter
    Letterfind = FALSE;    //The interval doesn't contain a new expanding letter
    minocc = lenw+1;     //Reset the minimum occurence data to pervent any error carry toward
    break;
  }
...
```

If $Efind$ is false, we set $Letterfind$ to true and compare the value of occurrences of this letter with the value of $minocc$. $minocc$ contain the current minimum occurrence of some letter in $'minchar'$ which we have found in the interval (If there was none, its default value will be greater then the length of the word). If the current value is smaller than the previous value, we update $minocc$ with the current number of occurrence and store the letter into $minchar$. The code below shows this part of the process.

```
...
else{
  char tempchar = word[i];  //The letter is not an expanding letter.
  int temploc = 0;
  for(int k = 0; k < uniq; k++){
    if( tempchar == letterdata[k].letter ){
      temploc = k;  //Find the location of the letter in the struct letterdata array
      break;
    }
  }
  int tempocc = letterdata[temploc].numoccur; //Get the num of occurence for that letter
  if( minocc > tempocc){  //Compare it to the previous minimum occurrence
    minocc = tempocc; //Update the new minimum occuring letter in this interval
    minchar = tempchar;
    loc = temploc;
  }
  Letterfind = TRUE;  //Currently this interval doesn't contain any expanding letter
}
```

After we have iterated through the interval, we check if $Letterfind$ is true. If $Letterfind$ is true after the iteration of letters in the interval is completed, then we increase $intervalfind$ by 1, meaning we have found an interval that does not contain any letter from E and we break out of the L $while\ loop$. Otherwise, we increase $Lpoint$ by 1 to check the next interval.

If $intervalfind$ = 0, this implies that there does not exist a new expanding letter so no more expanding letter can be found. The word is morphically imprimitive if the size of $Eset$ is not equal to size of the $letterdata\ array$. The function will set the return value $primitive = 1$ and break our of the $'core'$ $while\ loop$.

If the size of $Eset$ is equal to size of the $letterdata\ array$ (i.e. $E = symb(w)$), this means the word is morphically primitive and the function will set the return value $primitive = 0$ and break out of the $'core'$ $while\ loop$. (The $E = symb(w)$ check is performed at the end of the $'core'$ $while\ loop$ because it is impossible to have new expanding letter if every unique letter of the word are in E. Once, we have processed the last unique letter, there is no need to perform the search for new expanding letter).

If $intervalfind \neq 0$, then the new expanding letter will be the letter in $minchar$ and we add this letter to $Eset$. Next, we add the interpositions for condition b and c (or step 5 and 6) of the algorithm.

```
getposition = letterdata[loc].positions;
int limit = letterdata[loc].numoccur; //Using the total occurrence as a limit for looping
int leftneigh = letterdata[loc].leftlimit;  //The prefix and suffix length for that expanding letter
int righneigh = letterdata[loc].rightlimit;

//Condition b
for (int i = 0; i < limit; i++){
  newsetdata[getposition[i]].L = 1;
  newsetdata[getposition[i]+1].R = 1;
}
//Condition c
for (int j = 0; j < limit; j++){
  newsetdata[getposition[j]+righneigh+1].L = 1;
  newsetdata[getposition[j]-leftneigh].R = 1;
}
```

Now, we begin the synchronisation of the neighbourhoods. For this part of the function, it is implemented in the following way. First, we create a loop which will iterate all the expanding letter in $Eset$ from the newest to the earliest one. In each iteration, we get its $positions\ array$ and the length of the longest common prefix and suffix ($leftlimit$ and $rightlimit$) from $letterdata\ array$. Using these data, we are able to get the all the neighbourhoods of the current expanding letter in the word.

Now, we iterate each interposition of all neighbourhoods and check whether the interposition exists in the $Lset$ array (set L) and in the $Rset$ array (set R). If it exists in the set, we record the position of where it occurred in neighbourhood and store this into an array.

After we have iterated all the neighbourhoods of this letter in the word, we would have created a template which tells us all the locations where an interposition should exist in L and R in the neighbourhood interval. These templates are named as $Larray$ and $Rarray$ in the program. The code below shows the creation of these templates.

```
//Finding expanding interval pattern for neighbourhood
for (int i = 0; i < limit2; i++){ //Loop through the number of occurrence of the expanding letter
  int wordpos = getposition[i]-leftneigh; //Starting position for the interval of the current
    position of the expanding letter in the word.
  for(int j = 0; j < lengthneigh; j++){   //Loop through size of interval
    if(newsetdata[wordpos+j].L == 1){   //Checking if there is an L for that position.
      Larray[Larraypoint] = j;    //Add the current interval literation to the L templete (not the
    position of the letter in the word)
      Larraypoint = Larraypoint + 1;  //Increase the Larray limit.
    }
    if(newsetdata[wordpos+j].R == 1){
      Rarray[Rarraypoint] = j;
      Rarraypoint = Rarraypoint + 1;
    }
  }
  //Sort the templete, since we are adding new values at the end of the array.
  Larraypoint = sortset(Larray, Larraypoint);
  Rarraypoint = sortset(Rarray, Rarraypoint);
}
```

Using these template, we iterate through the location in the neighbourhood and check if the interposition exist in $Lset$ array (set L) and in $Rset$ array (set R). If the interposition does not exist in the set, we add the interposition to the set and we set the boolean variable $'repeat'$ to true. The code below shows this process.

```
//With the templete done, we add in the missing Ls and Rs.
for (int i = 0; i < limit2; i++){
  int wordpos = getposition[i]-leftneigh;
  for(int j = 0; j < Larraypoint; j++){
    int Lwordpos = wordpos+Larray[j];   //interval templete: starting word position + to the where
                                        //there exist an L within the interval templete
    if( newsetdata[Lwordpos].L == 0){
      newsetdata[Lwordpos].L = 1;   //Add the new position of L to the word.
      repeat = TRUE;  //Since we have added an L to the word, there is a possibility that it
                    //affect the other expanding interval, so we repeat the whole section later.
    }
  }... //Same for set R
```

When we have checked all the neighbourhoods of the current expanding letter, if $'repeat'$ is true, we restart the iteration for the expanding letter starting from the latest expanding letter added to $Eset$. After the synchronisation, the function will return to the beginning of $'core'\ while\ loop$ and attempt to find an expanding letters from the word again.

4.3.3 Finding Morphic Factorisation

After the function have determined whether the word is morpically primitive or imprimitive, it will move on to working out the morphic factorisation of the word if the user specified to show the results of each word in the program. This morphic factorisation algorithm taken from Holub's paper [7].

The algorithm itself is quite simple. Let us denote $w = u_0e_1u_1e_2u_2...u_{k-1}e_ku_k$ where e are expanding letters in the word and u contain the suffix of previous e and prefix of the current e, so $u_i = q_i \cdot p_{i+1}$. We want to find all the interposition which separate $q_i \cdot p_{i+1}$ for all expanding letters in the word. Using this interposition, we will be able to separate each expanding letter into their own prefix and suffix and obtain the morphic factorisation of the word. To find this interposition, we will use set L and R. Let $i \in L$, i is used for locating the start of an expanding letter in the word. We want to find $j \in R$ such that $j = \max\{a \mid a \leq i\}$. j is the interposition that signify the start of the prefix. For u_0, q_i does not exist, so $u_0 = p_1$ and for u_k, p_{k+1} does not exist, so $u_k = q_k$.

The code below shows how we obtain the factor for all expanding letters except the first expanding letter.

```
//Now getting all other prefixes of all expanding letter in the word
for (int k = temp1+1; k < Lsize-1; k++){
int temp2 = 0;
  for (int n = 0; n < Esize; n++){
    if(word[Lset[k+temp2]] == Eset[n].letter){
      Epos2 = Lset[k];
      EEpos2 = n;
      break;
    }
    if( n == Esize-1){
      temp2 = temp2 + 1;
      n = -1;
    }
  }
  k = k+temp2;
  for (int p = 0; p < Rsize; p++){
    if(Rset[p] <= Epos2){
      Eend = Rset[p];
    }
    if(Rset[p] > Epos2){
      break;
    }
  }
  //Storing the factorisation
  if(k == temp1+1 || Eset[EEpos1].datastart == -1){
    Eset[EEpos1].datastart = Estart;
    Eset[EEpos1].dataend = Eend;
  }
  Estart = Eend;
  Epos1 = Epos2;
  EEpos1 = EEpos2;
}
```

4.4 Set partition generators

In option 4 of the program, the user can select two different algorithm to generate set partitions of the set $\{1, 2, ..., n\}$ and perform Holub's Algorithm on these sets. The algorithm for first set partitions generator is based on the book written by "Albert Nijenhuis and Herbert S. Wilf" in 1978 [3] which is the first set partition algorithm implemented in the program. Later on, another second set partition generation algorithm was implemented which is based on the article by "M. C. ER" in "The Computer Journal" in 1988 [2] and it is known to be the fastest algorithm for generating all partitions of the set $\{1, 2, ..., n\}$.

In the program, for each word generated by the set partition algorithm, it will run Holub's function on the word. This method will save memory in the program as we do not store any set partition. The program allows the user to print out and save the results of the set partitions for length $n < 9$. Also for $n < 9$, both set partition generation algorithm are not multithreaded because a single threaded program is already quite fast to compute for the program.

For length $n > 9$, depending on the algorithm, the program will multithread itself into 6 threads (including the main thread) in order to divide the work load and improve the speed of the program during certain parts of the set partitions for n. Initially, both algorithms are designed to find all set partition after a given length n and some other perimeters. In order to make it suitable for multithreading, certain methods were discovered to break down set generation into sections. As a result, during multithreading, each thread will only generate certain part of the set partitions for n and the speed of the program is greatly improved.

5 Testing

In this section we investigate the correctness of the program and introduce some concepts in order to understand how the test is performed. Next, we compare the two implementations (Original Version and Improved Version) by running tests on the programs and analyse their performance.

5.1 Important Concepts

For words of length n, if we assume there is an infinite alphabet, it will be impossible to run any correctness test as there are infinitely many words. Therefore, it is necessary to impose restrictions which lead us to only deal with words in canonical form. Before we explain about words in canonical form, we must introduce the concept of renamings. A morphism r is a renaming if and only if r is injective and for every letter x, $|r(x)| = 1$ [4]. A word w is a renaming of a word v providing there is a renaming r mapping v onto w.

Example 12. The word $w = egfffg$ is a renaming of $v = abccb$ since there is a morphism r such that $r(a) = e, r(b) = g, r(c) = f$. r is a renaming mapping v onto w (i.e. $r(abccd) = egfffg$).

We can say that word $egfffg$ and $abccb$ are essentially the same, they have the same property but different letters. A word in canonical form means it is the lexicographically minimal word and it is the representative among all of it renamings [4]. For example, the word $abbca$ is the canonical form of the words $effge$ and $caabc$ etc. Since we consider words as preimages of morphisms, we can always rename the word with a different letter but property of the morphisms will not change. We are only interested in the property of the word. Therefore, do not consider any renamings and only deal with words in canonical form which is the representative of one type of words.

Reidenbach and Schneider [4] proposed that the number of word in length n that are in canonical form are equals to the number of all partitions of a set S into nonempty subsets where $|S| = n$. Using this proposition, we can use set partition generator algorithm to work out all words of length n in canonical form and the nth Bell number to work out the total number of partitions of a set of size n into nonempty subsets. By using the set partition generator algorithm and Holub's algorithm, we can find the number of morphically primitive words of length n that are in canonical form.

We denote B(n) as the nth Bell numbers which is the number of ways a set of size n can be partitioned into nonempty subsets [8] and mPrim(n) to denote total number of morphically primitive words of length n. we also denote Twice(n) as the Associated Stirling Number of the Second Form which is the current upper bound for estimating the total number of morphically primitive words of length n (Detailed explanation is given in Section 6).

5.2 Correctness Test

Now, we proceed to the correctness test of both implementation. As we have mentioned in section 4, option 4 of the program uses set partitions to generate all words of length n in canonical form and find the total number of morphically primitive words of length n from the generated words. A table of the number of morphically primitive words of length 1-12 was discovered by brute force method by Reidenbach and Schneider [4]. Using this table as an reference, both implementation are shown to have obtained the correct number of morphically primitive word of length 1-12. This provides a good evidence that both implementations are correct to length 12. By using multithreading concepts in both program, we were able to extend this table to length 15.

Table 1: Number of morphically primitive words of length 1-15

n	1	2	3	4	5	6
B(n)	1	2	5	15	52	203
mPrim(n)	1	1	1	3	11	32
Twice(n)	0	1	1	4	11	41
n	7	8	9	10	11	12
B(n)	877	4140	21147	115975	678570	4213597
mPrim(n)	152	625	3152	16154	90993	539181
Twice(n)	162	715	3425	17722	98253	580317
n	13		14		15	
B(n)	27644437		190899322		1382958545	
mPrim(n)	3398803		22561791		157493522	
Twice(n)	3633280		24011157		166888165	

While it is possible to obtain length greater then 15, the estimated time needed is not desirable. See Table 3 for further statistics of the time taken for both implementation to obtain the total number of morphically primitive words from length 3-15. As the length of n grows, the number of ways to partition a set of size n into nonempty subsets will grow exponentially and the time taken for the set partition algorithm will increase.

5.3 Comparison and Performance Tests

In this section, we compare the performance of both implementations (Original Version and Improved Version). We conducted two tests. The first test involve comparing the speed of the implementation against the number of input given. The second test involve comparing the speed of the implementation against a large input size.

5.3.1 Speed Comparison against number of inputs

Using the option 4 on both implementation, we measure the time taken for the program to find the total number of morphically primitive words of length n using set partitions generator algorithms. While there are two set partition algorithms to choose from, we have chosen the fastest algorithm by "M. C. ER" for this test. Table 2 show the time taken for the set partition algorithm to generate all words of length n.

Table 2: Time taken for fastest set partition algorithm to generate all words of length n.

n	3	4	5	6	7	8	9
Time Taken (sec)	0.001	0.004	0.005	0.005	0.006	0.006	0.010
n	10	11	12	13	14	15	
Time Taken (sec)	0.020	0.058	0.317	2.151	14.803	111.125	

Please refer to Table 1 for B(n) which is the total amount of input the implementations need to process for length n. Table 3 show the time taken for both implementations to compute the number of morphically primitive words of length n for set partition algorithms. Please note that for $n > 9$, both program will be multithreaded into six threads for certain set partitions generation of the algorithm in order to divide the work load and decrease the run time for the programs. We denote Original Version as Origver and Improved Version as Impver.

Table 3: Time taken for the two implementations to find the morphically primitive words of length n.

n	3	4	5	6	7	8	9
Time Taken for Origver (sec)	0.004	0.005	0.007	0.008	0.009	0.022	0.088
Time Taken for Impver (sec)	0.003	0.004	0.005	0.006	0.008	0.019	0.068
n	10	11	12	13	14	15	
Time Taken for Origver (sec)	0.289	1.312	8.828	72.054	687.802	5674.737	
Time Taken for Impver (sec)	0.167	0.966	6.214	45.878	375.737	3012.308	

We can see from Table 3 that the Improved Version is clearly faster than the the Original Version. This evidence that the improved algorithm perform faster then Holub's original algorithm. To work out the specific time taken for Holub's Algorithm to process large input of data, we can subtract the results in Table 3 from Table 2. Let denote the time taken for Holub's Algorithm in the Original Version as tho(n) and the time taken for Holub's Algorithm in the Improved Version as thi(n). tho(n)/thi(n) denote the increase in speed of thi(n) compared to tho(n) (This is ignored for time less than 1 second).

Table 4: Time taken for Holub's Algorithm to process words of length n.

n	3	4	5	6	7	8	9
tho(n) in seconds	0.003	0.001	0.002	0.003	0.002	0.016	0.078
thi(n) in seconds	0.002	0.000	0.000	0.000	0.002	0.013	0.058
tho(n)/thi(n)	-	-	-	-	-	-	-
n	10	11	12	13	14	15	
tho(n) in seconds	0.269	1.254	8.511	69.903	672.999	5424.208	
thi(n) in seconds	0.167	0.908	5.897	43.727	360.934	2901.183	
tho(n)/thi(n)	-	1.381	1.443	1.599	1.865	1.870	

As we can see from Table 4, from $n = 3$ to $n = 10$ the speed between the two implementation are around the same. Starting from $n = 11$, we can see the implementation of the improved algorithm is finishing faster than the Holub's original algorithm and from $n = 12$, we can see a clear difference in speed between the implementation. Using the results from $n = 11$ to $n = 15$, we can see that the speed of the improved implementation increases as n increase. From Table 4, we can conclude that the Improved version of Holub's Algorithm will be at least 1.3 time faster than the Original version of Holub's Algorithm for finding the morphically primitive word of length $n > 10$.

Lastly in terms of memory, both program will not use more than 2.00 MB of the system memory. While we are working with a large number of inputs, the program does not require large amount of memory to run. This show both programs have good memory management.

5.3.2 Speed Comparison against size of input

In this section, we measure the time taken for the program to process one large string of word between the two implementation. Using Shakespeare's play "Romeo and Juliet", by getting rid of the spaces and symbols in the text, we have one large string which is used to test the implementation. Initially, both implementations could not process the text because the amount of time needed was too long. More than two hours were spent on processing the text but both implementations could not finish processing it. The reason was due inefficient sorting and retrieving information about the interposition from arrays ($Lset$ array and $Rset$ array) which represent L and R in the algorithm.

$Lset$ and $Rset$ only stored interposition values in the array. To retrieve information about an specific interposition, we need to iterate through the array and check every value in the array to see if the desired interposition is within the array. To improve this method, we have introduced the new array structure $LRset$ which is a list of interposition of the word with the information of whether the each interposition is in L and R. With this method, we only need to input the value of interpositions into the array and the information about whether the interposition is in L and R can be easily retrieved.
We use the following plays as inputs for this test:

- Romeo and Juliet: 105848 characters.
- Merchant of Venice: 90620 characters.
- Much Ado about nothing: 91652 characters.
- Tempest: 73584 characters.
- Twelfth Night: 85750 characters.
- A modified version of Romeo and Juliet which is morphically imprimitive: 105756 characters.

With this new array, the time needed for the Improved Version to process the play "Romeo and Juliet" was virtually about **less than half a second** (around 0.3 seconds) which is a major improvement compare to before. However with the Original Version, it is still not possible to process the file even after 2 hours.

Next, we test the Improved Version with multiple files. Using option 3 of the implementation, we insert all plays into the program and process them. With the Improved Version, the program took still manage to took less than half a second to perform Holub's Algorithm on these files. It is not possible to test this with the Original Version because it took too long to process one file. The memory used during this test is the total size of the text file inserted into the program.

From the two comparison tests, it is evidenced that the improved version of Holub's Algorithm is faster than the original version of Holub's Algorithm from these speed comparison tests. With the improved Holub's Algorithm, we are able to process large size of inputs or long strings of letters whereas the original Holub's Algorithm could not. Both implementations do not need large amounts of system memory to run. We can conclude that implementation of the improved version of Holub's Algorithm is better than implementation of the original version of Holub's Algorithm.

5.4 Testing with known examples

Using the same word as in section 3.3, $w = abccdabcd$, we show a series of screenshots showing the set L and set R of each step in the algorithm.
First, Figure 22 will show the start of the program, the word and sets L and R from step 1 to step 5:

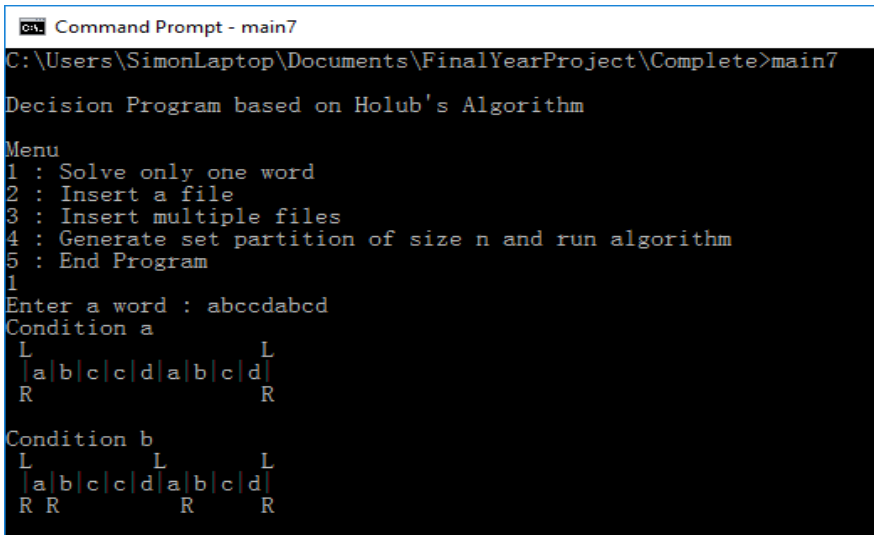


Figure 22

As we can see from Figure 22, the expanding letter chosen in this iteration is letter a and interposition $w[i, i+1]$ are added to set L and R. Next, we proceed to step 6 to 8 and return to step 3 to 6. We add interpositions at the start and end of n_a and showing the synchronisation of a and find the next expanding letter.

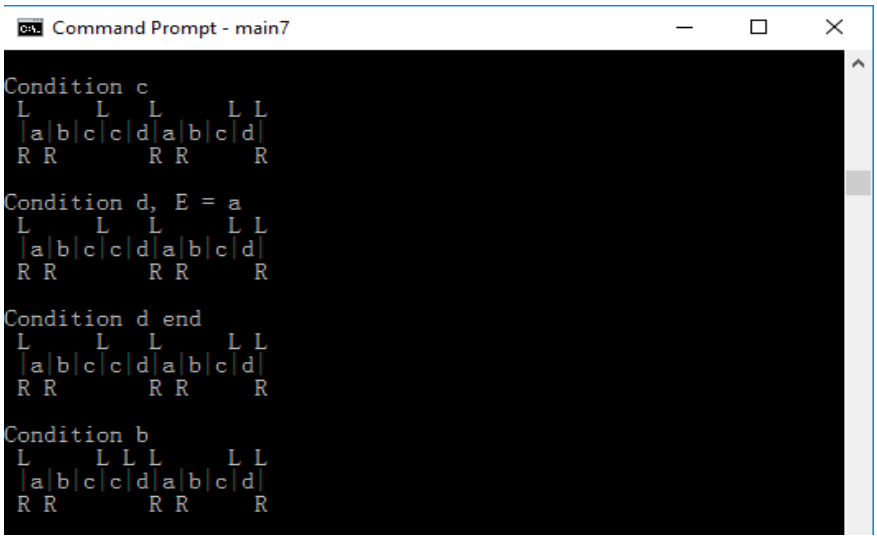


Figure 23

As we can see from Figure 23, $n_a = abc$ and the interpositions of $n_a = w[i, j], i \in R, j \in L$ are added. No interposition are added during the synchronisation. The new expanding letter is d and interposition $w[i, i+1]$ of letter d is added. Next, we add the interposition of n_d and proceed to synchronisation of d.

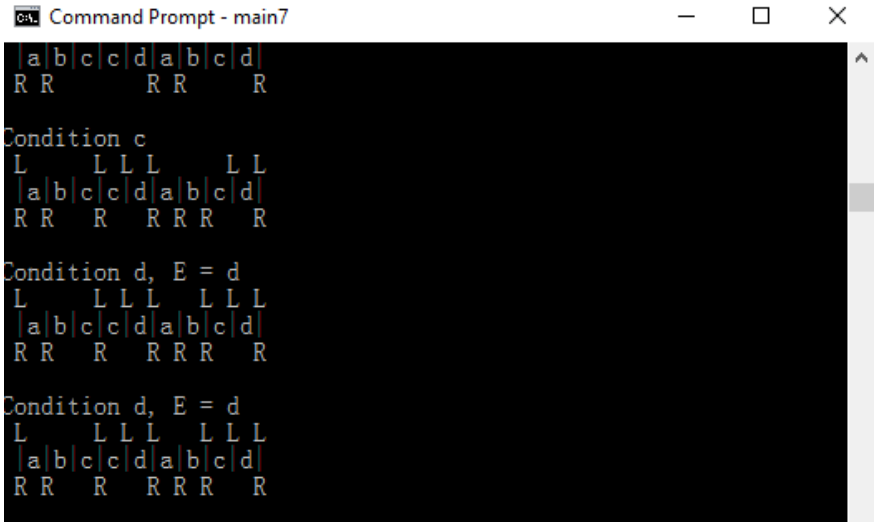


Figure 24

As we can see from Figure 24, we added the missing interposition into set L during first synchronisation of letter d. As an interposition was added to a set, we restart the synchronisation of E, which is why the synchronisation of letter d is repeated. Next, we proceed synchronise the next letter in E.

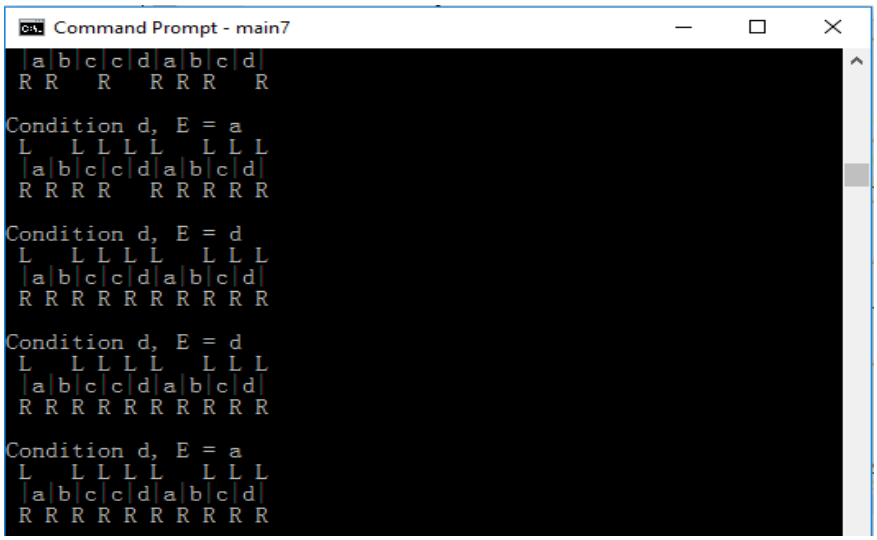


Figure 25

New interpositions have been added during synchronisation of letter a so the synchronisation restarted at d. Then, an interposition has been added to R during the synchronisation of letter d so it restarts again.
The synchronisation continues and Figure 26 shows the next process.

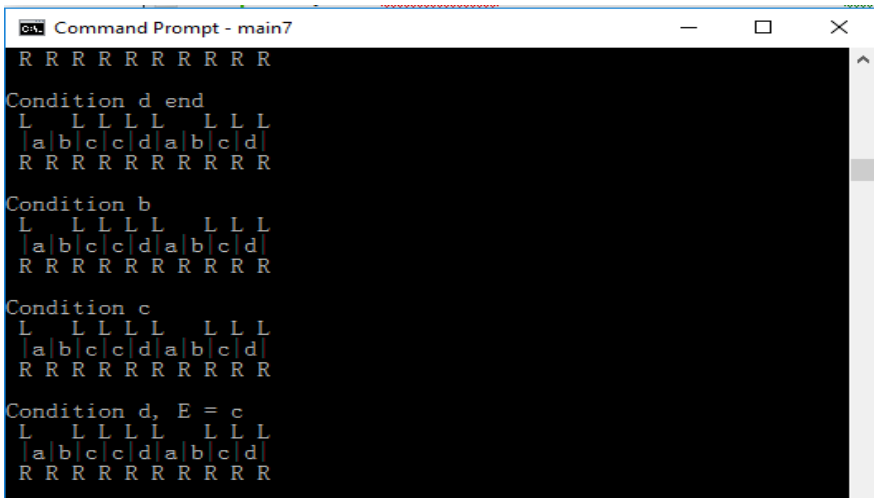


Figure 26

The synchronisation ends and the program proceeds to find the next expanding letter. The new expanding letter is c and we add the interposition needed for c. It is hard to from looking at L and R in Figure 26 because all the necessary interpositions already exists in L and R. However, the program had performed these steps as evidenced by the outputs in Figure 26.
We proceed to synchronisation of letters in E starting with letter c.

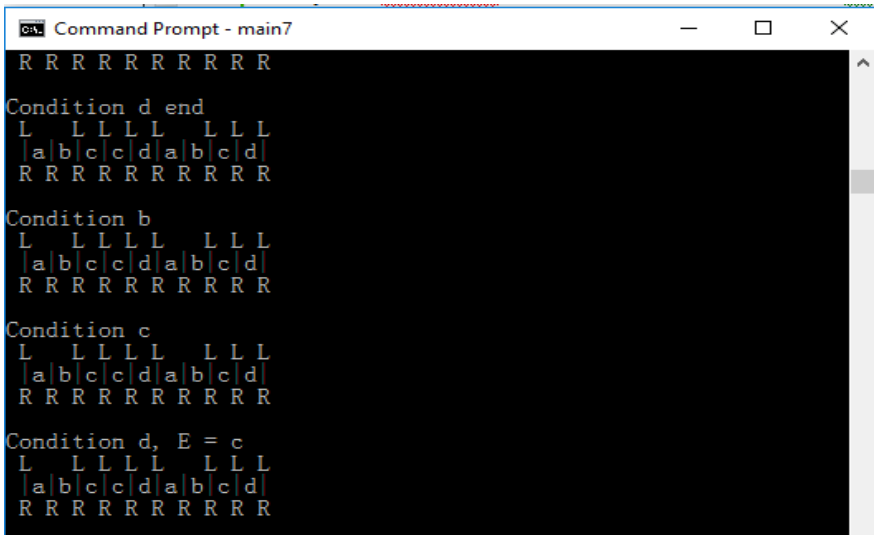


Figure 27

The synchronisation has ended and no new interpositions are added to L and R. There does not exist a new expanding letter so Holub's Algorithm ends. The program show the final result of set L and R, morphic factorisation of $w = abccdabcd$ and print out whether the word is morphically primitive or imprimitive.

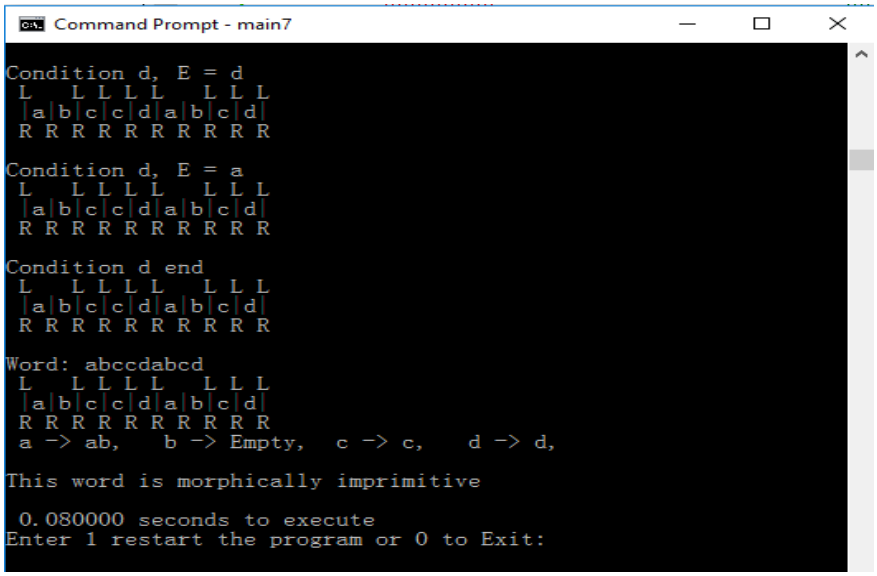


Figure 28

The word $w = abccdabcd$ is morphically imprimitive with a possible morphic factorisation. The program will also print out the amount of time spent by the program to run this word.

6 Total number of Primitive Words

To the present day, no equation has been formulated to work out the number of morphically primitive words for any fixed length n. In Reidenbach and Schneider "Morphically Primitive Words" [4], some bounds are given. $mPrim(n) \leq Twice(n)$ and $B(\lfloor n/2 \rfloor) \leq mPrim(n) \leq B(n-1)$. In the section, we introduce methods to reduce further the upper bound of estimating the number of primitive words of length n.

6.1 Terminology

First, we introduce some definitions to help us understand the topic. Please refer to [8] and [9] for detailed explanations and formulas. Bell number is the number of ways that a set of n elements can be partitioned into nonempty subsets and we denote this as B(n). For example, if $n = 3$, there are five ways to partition this into nonempty subsets: one subset = $\{1, 2, 3\}$, two subsets = $\{1, 2\}\{3\}, \{1, 3\}\{2\}, \{2, 3\}\{1\}$ and three subsets = $\{1\}\{2\}\{3\}$. Therefore, $B(3) = 5$.

In the example above, we see that can partition a set of n elements into k nonempty subsets. The number of ways that we can partition a set of n elements into k nonempty subsets is called the Stirling number of the Second Kind, denoted as $S(n, k)$. For example, if $n = 3$, $m = 2$, the number of ways that a set of 3 elements with a subsets of 2 is $\{1, 2\}\{3\}, \{1, 3\}\{2\}, \{2, 3\}\{1\}$. So, $S(3, 2) = 3$.

Table 5 show a table of Stirling number of the Second Kind $n = \{1, .., 10\}$ and $k = \{1, .., 10\}$. The values are obtained from [5].

Table 5: Stirling number of the Second Kind from 1 to 10

k/n	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	-	1	3	7	15	31	63	127	255	511
3	-	-	1	6	25	90	301	966	3025	9330
4	-	-	-	1	10	65	350	1701	7770	34105
5	-	-	-	-	1	15	140	1050	6951	42525
6	-	-	-	-	-	1	21	266	2646	22827
7	-	-	-	-	-	-	1	28	462	5880
8	-	-	-	-	-	-	-	1	36	750
9	-	-	-	-	-	-	-	-	1	45
10	-	-	-	-	-	-	-	-	-	1

Next, we introduce the r-associated Stirling number of the Second Kind [1]. An r-associated Stirling number of the Second Kind is the number of ways to partition a set of n elements into k nonempty subsets, with each subset containing at least r elements. This is denoted by $S_r(n, k)$. For example, if $n = 4, k = 2$, $S(4, 2) = 7$ with set partitions: $\{1, 2, 3\}\{4\}, \{1, 2, 4\}\{3\}, \{1, 2\}\{3, 4\}, \{1, 3, 4\}\{2\}, \{2, 4\}\{1, 3\}, \{2, 3\}\{1, 4\}, \{2, 3, 4\}\{1\}$. With $n = 4, k = 2, r = 2$, we have $S_2(4, 2) = 3$ with set partitions: $\{1, 2\}\{3, 4\}, \{2, 4\}\{1, 3\}, \{2, 3\}\{1, 4\}$. As we can see, each subsets have at least two elements. Table 6 shows a table of r-associated Stirling number of the Second Kind $n = \{1, .., 12\}$, $k = \{1, .., 6\}$ and $r = 2$. The values are obtained from [1] or [6].

Table 6: 2-Associated Stirling number of the Second Kind from 1 to 12

k/n	2	3	4	5	6	7	8	9	10	11	12
1	1	1	1	1	1	1	1	1	1	1	1
2	-	-	3	10	25	56	119	246	501	1012	2035
3	-	-	-	-	15	105	490	1918	6825	22935	74316
4	-	-	-	-	-	-	105	1260	9450	56980	302995
5	-	-	-	-	-	-	-	-	945	17325	190575
6	-	-	-	-	-	-	-	-	-	-	10395
Twice(n)	1	1	4	11	41	162	715	3425	17722	98253	580317

The sum of the r-associated Stirling number of the Second Kind for length n is Twice(n) in Table 1 which is the current upper bound of mPrim(n).

6.2 Methods for working out morphically primitive words

The method to work out the number of morphically primitive words of length n lies within the 2-associated Stirling number of Second Kind. As we know from Reidenbach and Schneider [4], $mPrim(n) \leq Twice(n)$. The problem at the moment is that there exists morphically imprimitive words within $Twice(n)$. Therefore, by working out the number of morphically imprimitive words within $Twice(n)$, we can reduce the upper bound for $mPrim(n)$ and hopefully lead to a formula for $mPrim(n)$ eventually.

Due to the time constraints of this project, detailed proofs are not written for these methods. Given a set partition $\{1,2\}\{3,4\}$, we can generate the word $aabb$ if we assume that each subset is number of letters, the numbers within the subset is the position in the word and the size of each subset is the number of repeated letters.

6.2.1 Method for partition of n with k = 2

Assuming that each subset has no elements less than 2, we work out all possible partitions of n into subset of k. For example, with $n = 6$, limiting to two subsets ($k = 2$), we have $6 = 5 + 1, 6 = 4 + 2$ and $6 = 3 + 3$. Let denote them as $(5,1), (4,2), (3,3)$ where the first element is the size of first subset and the second element is the size of second subset (i.e. $6 = 5 + 1$, so subset one has 5 elements and subset two has 1 element so we get $(5,1)$). We ignore partition $(5,1)$ because it has a subset with less than 2 element so it is not within $Twice(n)$.

The use of number partitions can be used to obtain the possible partitions from the size of the subsets for word of length n. The algorithm for number partitions generation can be found in [3].

The partition of 6 with $k = 2$ is $(4,2), (3,3)$ and using each element as a reference of the size of each subset, the set partition generated from these partitions will give us the value $S_2(6,2) = 25$. There exist morphically imprimitive words within these partitions. The words are $aba\ aba$, $aab\ aab$ and $abb\ abb$ for $(4,2)$ and $ab\ ab\ ab$ for $(3,3)$.

To work out the imprimitive words of $(4,2)$, let consider one subset is the expanding letter a and the other subset is the mortal letter b. The highest element is always the mortal set when $k = 2$. We have two a and four b for the partition of $(4,2)$. For a word that contain two different letters to be morphically imprimitive, a factor must contain an expanding letter and some mortal letters and all factors must be the same. Since all factors are the same, such a factor when $k = 2$ will only have one expanding letter and an unknown number of mortal letters. To find the number of mortal letters, we divide the total number of mortal letters with the number of expanding letters. For $(4,2)$, $4/2 = 2$ so we have two mortal letters in the factor. In the case where the elements are not divisible (i.e partition $(5,3)$), it is impossible to find a factor such that all the expanding letter will use up all the letters in the mortal set. Therefore, the partition will not generate any imprimitive words.

With the factor containing one expanding letter and two mortal letter, we need to find all possible patterns with these letters. One method of working this out would be by permutation. Let the $\{m_1, ..., m_n\}$ to be the number of repetitions for each letter and length of factor to be x, then

$$P_x^{(m_1,...,m_n)} = \frac{x!}{m_1!...m_n!}$$

This will give out the number of possible patterns of the factor. The length of the factor is the number of expanding letter plus the number of mortal letter in the factor, so $P_3^{(1,2)} = 3$ The factor are abb, bab and bba, when converted to canonical form, it become abb, aba and aab.

Another way of working out the possible patterns of the factor would be to use the Stirling number of Second Kind and 2-associated Stirling number of Second Kind. The formula is

$$S(x,2) - S_2(x,2).$$

This formula tells us the number of the set partition of x with subsets of 2 where one of the subset contains only one element (which is the expanding letter in the factor). This is equivalent to finding all possible patterns of the factor.

The value of possible factors available with the given letters is the number of morphically imprimitive words of given partition of k = 2. Therefore, with these formula, we are able to remove all morphically imprimitive word for all set partition with subset where k = 2.

6.2.2 Method for partition of n with k = 3

For partition of n with $k = 3$, a different method is used to find the imprimitive words, but it contains similar concepts to finding imprimitive words for $k = 2$. First, we need to generate number of n with $k = 3$ such that $r \geq 2$. For example, for $n = 8$, $k = 3$, we have partitions $(2,2,4), (2,3,3)$. There are three different cases we need to consider for number partitions of n when $k = 3$.

Case 1: Let assume that we have highest value elements that are dividable with only one other element in the partition. Let us use the partition $(2, 3, 4)$ as an example. Similar to $k = 2$, we use element 1 (value of 2) and element 3 (value of 4) to work out possible factors available for these values. Using the formula, we get $S(3, 2) - S_2(3, 2) = 3$. The three factors are abb, aba and aab, let denote them as expanding patterns. We know that the number of expanding letter is 2 from the partition, so there are two factors that contain the expanding letter. Let denote these factor as expanding factors. With the remaining 3 letters in element 2 not belonging in any expanding letter, they are treated as individual factors. In total, we have 5 factors, two expanding factors and 3 individual factors. As long as these are at least two expanding factors in the word, it is morphically imprimitive. To find all permutations of these factors, we use the permutation formula in $k = 2$:

$$P_t^{(f_1,...,f_n)} = \frac{t!}{f_1!...f_n!}$$

The value of t is the total number of factors and $f_1, ..., f_n$ is the number of repetitive factors we have. Now, with 2 expanding factors and 3 individual factors we have:

$$P_5^{(2,3)} = \frac{5!}{2!3!} = 10.$$

There are 10 possible permutations for these factors. Since, we have three different expanding patterns available, we multiply this by the value of possible permutations to obtain total number of morphically imprimitive word for partition $(2, 3, 4)$. The formula is as follows:

$$(S(x, 2) - S_2(x, 2)) \cdot (P_t^{(f_1,...,f_n)})$$

x is the total number of mortal letters divide with total number of expanding letter + 1. t is the total number of factors and $f_1, ..., f_n$ is the number of repetitive factors.

Case 2: Let assume that all elements in the partition are equal to each other. Let use the partition $(2, 2, 2)$ as an example. We use the formula we found in case 1. However, because all elements are equal to each other, the formula will produce repeated words. This is because our formula does not consider the canonical form of the word during permutation. Therefore, it will produce some words that has the same canonical form. To prevent this, we need to subtract these repeated words from the formula. The repeated pattern is cab when converted to canonical form, we get abc. Therefore $abc\ abc$ = $cab\ cab$. For $k = 3$ this is the only repeat case to consider. The formula for case 2 is to use the formula in case 1 and subtract the results by 1.

Case 3: Let assume that two elements are dividable by the other element in the set. Let use the partition $(2, 2, 4)$ as an example. With element 2 and 3 are divisible by the element 1 in the set, this suggests that there are other expanding letters that share mortal letters. In order to locate these letters and prevent repeating cases, first we use the formula in case 1 for all possible elements that are dividable with each other.

In our example, we have $(\underline{2}, \underline{2}, 4)$, the expanding factor is one expanding letter and one mortal letter. There are two expanding factors and four individual factors and the formula in case 1 given us

$$(S(2, 2) - S_2(2, 2)) \text{ x } (P_6^{(2,4)}) = 15.$$

With $(\underline{2}, 2, \underline{4})$, there are three available expanding patterns. The expanding factor has one expanding letter and two mortal letters. In total there are two expanding factors and two individual factors in the word, so

$$(S(3, 2) - S_2(3, 2)) \text{ x } (P_4^{(2,2)}) = 3 \text{ x } 6 = 18.$$

Next, we choose the lowest element in the partition and divide the elements in the set by that value, so $(2, 2, 4)/2 = (1, 1, 2)$. Using the remaining elements, we have a factor with two expanding letters and two mortal letter. Using $S(n, k)$, we have $S(4, 3) = 6$. There are 6 different expanding patterns available. Using the example, we obtain the following expanding patterns: $abcc$, $acbc$, $accb$, $cacb$, $ccab$ and $cabc$. When converted to canonical form, we have $abcc$, $abcb$, $abbc$, $abac$, $aabc$ and $abca$. Within these patterns, there are two unique letters. As long as one letter has a factor that contains all mortal letter, or both letter has a factor that contain equal amount of mortal letter, it is morphically imprimitive. While we have worked out all the pattern manually, there was not time to work out an formula for this. After we have obtained value for this, we proceed to remove excess word previous generated by $(\underline{2}, \underline{2}, 4)$ and $(\underline{2}, 2, \underline{4})$.

To remove these words, we need to divide the remaining element that was not used to create the expanding factor with number of expanding factors. Let denote that value as x. We used the permutation formula $P_{x+1}^{(x)} = y$, this will give the number of different patterns created with expanding factor and the remaining letters when the remaining letters are shared among the expanding factors. y^2 will give us the excess words in the partition. If there exists a different pattern of expanding factors, we multiply this by y^2 before subtracting it from the partition.

6.2.3 Method for partition of n with k = 4

For partition of n with $k = 4$, similar concepts are used. First, we need to generate number of n with $k = 4$ such that $r \geq 2$. Unfortunately due to time constrain, we were only able to identify two different cases to consider for number partitions of n when $k = 4$.

Case 1: Let assume that all elements are equal to each other (i.e. partition $(2, 2, 2, 2)$). We pick two elements to be the number of expanding letters and number of mortal letters and work out the expanding factor. All other elements are seen as individual factor and we permutate all the factors, so $(S(2, 2) - S_2(2, 2))$ x $P_8^{(2,2,2)} = 90$.

We then divide the result by number of extra subsets that was not used for the expanding factors so $90/2 = 45$. This is because we have two subsets that have equal elements and are not used for the expanding factor. Let denotes these subsets with letters c and d. Since they are not in the expanding factor, they are extra letters used for permutation. Since these letters have the same property, it does not matter if the letters c is changed to d and d is changed to c since the resulting canonical form will be the same. However, the permutation formula classify these two subset as different letters. Therefore, repeated words are generated. By dividing the result by number of extra subsets that have the same size will remove this problem.

The next problem is the repeating word $abc\,abc = cab\,cab$ which we introduced back in case 2 when $k = 3$. However, since we have 4 subsets instead of 3, we need to permutate these expanding factors with the other subset. We divide this value by half because we do not need the words that are not in canonical form, so $P_4^{(2,2)} = 6/2 = 3$ and $45 - 3 = 42$. Furthermore, we have repeated words because we can have two different expanding factors created by different subset. Let denote (2,2,2,2) to be subset 1, subset 2, subset 3 and subset 4 with value of 2. Since subset 1 and 2 are used to create the expanding factor ab, subset 3 and 4 can be used to create expanding factor cd. Let denote ab to be expanding factor 1 and cd to be expanding factor 2. There are two expanding factor 1 and two expanding factor 2. With permutation of four factors, $P_4^{(2,2)} = 6$. However, half of these permutation have the same canonical form (i.e. $ababcdcd$ = $cdcdabab$). This is because expanding factor 1 and 2 have the same pattern but different letters. Therefore, we divide the permutation by a half, $6/2 = 3$, so $42 - 3 = 39$.

Case 2: We have three elements that are equal to each other but the last element is not (i.e partition $(2, 2, 2, 3)$). Similar to case 1, We pick two elements to be the number of expanding letters and number of mortal letters to work out the expanding factor. All other elements are seen as individual factor and we permutate all the factors. Unlike case 1, we only need to consider repeating words that happens when three subsets have the same size (i.e. $abc\,abc = cab\,cab$). With the partition $(2, 2, 2, 3)$, we have

$$(S(2, 2) - S_2(2, 2)) \text{ x } (P_7^{(2,2,3)}) - (P_5^{(2,3)}) = 210\text{–}10 = 200.$$

With these following methods, we are able to successfully remove all morphically imprimitive words for set of length n with subset of $k = 2$ and remove some morphically imprimitive words for subset of $k = 3, .., 4$. Unfortunately, given the time constraints of the project, we were unable to write a mathematical formula for this reduction. However with the introduced methods, we can conclude that we have reduced the current upper bound of estimating the total number of morphically primitive words. With further research, it is possible to continue reduce this upper bound and obtain a formula to work out the total number of morphically primitive words of length n.

6.3 Workings

With the following methods from $k = 2, ..4$, we can work out the number of morphically primitive words up to n = 8. Using the methods we introduced above, we work out the number of morphically primitive from $n = 4, ...8$.

- For $n = 4$, with $k = 2$, we only have the number partition $(2, 2)$.
  With $(2, 2)$, we have one expanding letter and one mortal letter for the expanding factor. The number of morphically imprimitive word in $(2, 2)$ (denote this as $imprim(2, 2)$) is $imprim(2, 2) = S(2, 2) - S_2(2, 2) =$ 1–0 = 1. Total number of morphically primitive word (denote this as $prim(n)$) is $prim(4) = Twice(4) - 1 = 4$–1 = 3.

- For $n = 5$, number partition of 5 is $(5), (1, 4), (2, 3), (1, 1, 3), (1, 2, 2), (1, 1, 1, 2), (1, 1, 1, 1, 1)$.
  There does not exist a partition that contain two dividable elements. Therefore, $prim(5) = Twice(5) = 11$.

- For $n = 6$, the number partition of 6 with the minimum value of 2 in every element, we have $(2, 4)$, $(3, 3)$ and $(2, 2, 2)$.
  - When $k = 2$, we have the partition $(2, 4)$ and $(3, 3)$ with potential imprimitive words

* For partition $(2,4)$, we have one expanding letter and two mortal letter for the expanding factor (from $4/2 = 2$). $imprim(2,4) = S(3,2) - S_2(3,2) = 3–0 = 3$. (The words are *abbabb*, *babbab*, *bbabba*).
* For partition $(3,3)$, we have one expanding letter and one mortal letter for the expanding factor (from $3/3 = 1$). $imprim(3,3) = S(2,2) - S_2(2,2) = 1–0 = 1$. (The word is *ababab*).

– Now, when $k = 3$, we have the partition $(2,2,2)$ with potential imprimitive words.
Using the formula in $k = 3$, case 2,
$imprim(2,2,2) = ((S(2,2) - S_2(2,2))$ x $P_4^{(2,2)}) - 1 = 1$ x $6 - 1 = 5$. (The words are *ababcc*, *ccabab*, *abccab*, *cababc*, *abcabc*, *cabcab*).

$prim(6) = Twice(6) - imprim(2,4) - imprim(3,3) - imprim(2,2,2) = 41 - 3 - 1 - 5 = 32$

- For $n = 7$ with $k = 2$, there do not any partition that contain two dividable elements. So there is no morphically imprimitive word for $k = 2$.

– When $k = 3$, partition $(2,2,3)$ with two divisible elements.
Using the formula in $k = 3$, case 1, $imprim(2,2,3) = ((S(2,2) - S_2(2,2))$ x $P_5^{(2,3)}) = 1$ x $10 = 10$

$mprim(7) = Twice(7) - imprim(2,2,3) = 162 - 10 = 152$.

- For $n = 8$, number partition of 8 with the minimum value of 2 in every element is $(2,6)$, $(4,4)$, $(2,3,3)$, $(2,2,4)$, $(2,2,2,2)$

– When $k = 2$, we have partition $(2,6), (4,4)$.
$imprim(2,6) = S(4,2) - S_2(4,2), 7–3 = 4$ and $imprim(4,4) = S(2,2) - S2(2,2), 1–0 = 1$.

– When $k = 3$, we have partition $(2,2,4), (2,3,3)$

* With partition $(2,2,4)$, we use the incomplete method discussed in case 3 of $k = 3$. $(\underline{2},\underline{2},4)$, first we use the normal formula
$imprim(\underline{2},\underline{2},4) = (S(2,2) - S_2(2,2))$ x $P_6^{(2,4)}) = 1$ x $15 = 15$.
$imprim(\underline{2},2,\underline{4}) = (S(3,2) - S_2(3,2))$ x $P_4^{(2,4)}) = 3$ x $6 = 18$.
Now, the number of imprimitive words that shares mortal letter is $imprim(\underline{2},\underline{2},\underline{4}) = 24$. (Note the formula have not been worked out)
Now, we remove repeating words in $imprim(\underline{2},\underline{2},4)$ and $imprim(\underline{2},2,\underline{4})$.
$imprim(\underline{2},\underline{2},4) = 15 - (P_3^{(2)})^2 = 6$
$imprim(\underline{2},2,\underline{4}) = 3$ x $(6 - (P_2^{(1)})^2) = 6$
So, $imprim(2,2,4) = 6 + 6 + 24 = 36$
* With partition $(2,3,3)$, we use case 1 of $k = 3$ to get
$imprim(2,3,3) = (S(2,2) - S_2(2,2))$ x $P_5^{(2,3)} = 1$ x $10 = 10$.

– When $k = 4$, we have partition $(2,2,2,2)$. Using case 1 of $k = 4$, we first have: $imprim(2,2,2,2) = (S(2,2) - S_2(2,2))$ x $P_6^{(2,2,2)} = 1$ x $90 = 90$. Since there exists two elements with the same value that is not used for expanding factor: $imprim(2,2,2,2) = 90/2 = 45$. Then we remove the repeating words that occur when three elements have the same value, $imprim(2,2,2,2) = 45 - P_4^{(2,2)}/2 = 45 - 6/2 = 42$. Lastly, we remove the repeating words that occur when we have two expanding factor with the same pattern but with different letters, $imprim(2,2,2,2) = 42 - P_4^{(2,2)}/2 = 42 - 6/2 = 39$.

$mprim(8) = Twice(8) - imprim(2,6) - imprim(4,4) - imprim(2,2,4) - imprim(2,2,2,2) = 715 - 4 - 1 - 36 - 10 - 39 = 625$.

7 Conclusion

In this report, we have presented various definitions and terminologies to introduce the topic of morphisms and to understand the concepts of Holub's Algorithm.

We first examined Holub's Algorithm and found out that it was not practical to implement Holub's Algorithm directly. This is because the process of finding expanding letters in Holub's algorithm can lead to finding more than one new expanding letter in one iteration and the new expanding letters are to be handled simultaneously by the algorithm which was not practical to implement in a computer program. In order to implement Holub's Algorithm, we have limited the process of finding expanding letters to only find one new expanding letter in one iteration. This would not have change Holub's Algorithm because the process of finding expanding letters is still the same and the goal of Holub's Algorithm is to find all expanding letters.

We had presented the steps of Holub's Algorithm in detail and shown a way of finding the morphic factorisation of the given word from the results of Holub's Algorithm. An example was shown to illustrate the process of the algorithm for further understanding.

Next, we have further analysed the process of finding expanding letters and identified the inefficiencies of the method used in Holub's Algorithm. We proposed an improved method of finding expanding letters which eliminate the inefficiencies from Holub's Algorithm and provided a proof for the correctness of this method along with an example to illustrate the efficiency of the improved method compared to original method.

This proposed method is an improvement to Holub's Algorithm. Hence, two programs was created where each program contains a different version of the implementation of Holub's Algorithm. The first program contains Holub's Algorithm with the original method of finding expanding letters which is denoted as the Original Version in the report and the other program contains Holub's Algorithm with the improved method of finding expanding letter which is denoted as the Improved Version. Both programs are written in the programming languages C and they are thoroughly described in the report.

To verify the correctness of both versions of Holub's Algorithm, two set partition generating algorithms were implemented into both programs. Set partition generating algorithm are used for generating all words of length n in canonical form. Using these generated words, we were able to work out the total number of morphically primitive words of length n that are in canonical form. The total number of morphically primitive words from length $1 - 12$ were previously found by Reidenbach and Schneider [4]. Using these results, we are able to verify the correctness of both programs. Furthermore, through the use of multithreading, we were able to extend the results from up to length 15.

To obtain evidences that the performance of Improved Version is better then the Original Version, two speed comparison tests were performed. A speed comparison test against the number of inputs and the speed comparison test again the size of the inputs. The number of inputs test was done by measuring the time taken for the programs to work out the total number of morphically primitive words of length 3-15. As the length of the word increase, the number of generated word grow exponentially. For example, with length 11, the set partition will generate a total of 678570 words and with length 12, it will generate a total of 4213597 words. It was found that the improved Holub's Algorithm was at least 1.3 time faster compared to the original Holub's Algorithm when finding of the total number of morphically primitive word from length 11. As the length of the word increase (which implies the number of words increases), the time taken for the improved algorithm to finish decreases compared to the original algorithm.

Next, the size of the input test was done by using converting Shakespeare's play into one string of input by removing all symbols and spacing in the play and insert the play into the programs and measure the time taken to process the test. For example, the play "Romeo and Juliet" is converted into a string of 105848 characters. In result, the improved algorithm was able to process a string of 105848 characters within 1 second whereas the original algorithm could not process the string within an acceptable amount of time. This implies that the improved algorithm can be used to process very long length of words. If the time taken to process large number of inputs was improved, the improved algorithm can be used to further extend the results for working out the total number of morphically primitive words. From these comparison and performance tests, we concluded that the improved algorithm has a better performance wise compared to the original algorithm.

Lastly, methods were introduced to reduce the existing upper bound that estimate the number of morphically primitive words. The current upper bound was proposed by Reidenbach and Schneider [4]. The proposed methods are used to obtain a number of morphically imprimitive words that are included in the proposed upper bound. The number of morphically imprimitive words in the upper bound are removed, reducing the current upper bound and give a closer estimate of the number of morphically primitive words. The methods use number partitions to obtain all the possible size of subsets from set partitions and we can work out the number of imprimitive words that can be made from set partitions generated from these possible size of subsets. In the report, the method of finding the number of imprimitive words that are generated by

set partitions with two subsets was perfected but the methods for set partitions with three and four subsets were only partially completed. Nevertheless, the methods we have introduced in this report have successfully reduced the current upper bound proposed in [4] and they were able to find the number of morphically primitive words of length 3-7.

For future research, one may try to find the total number of morphically primitive word greater than length 15 using the improved algorithm proposed in the report since we have evidenced that the improved algorithm was able to handle long words with over 100000 characters or words consist of 26 different characters. Another direction of future research would be to use the underlying theory introduced in section 6 to continue to further reduce the current upper bound that estimate the number of morphically primitive words and attempt to create a mathematical formula which calculate the number of morphically primitive words of length n.

From this report, we have successfully met all the main objectives for this project and the additional objectives were also achieved to a high standard. Therefore, we can conclude that the project was a success.

References

[1] L. Comtet. *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Revised edition, 1974.

[2] M. C. ER. A fast algorithm for generating set partitions. *The Computer Journal*, 31(3):283–284, 1988.

[3] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithm For Computers and Calculators*. 2nd edition edition, 1978.

[4] Daniel Reidenbach and Johannes C Schneider. Morphically primitive words. *Theoretical Computer Science*, 410(10):2148–2161, 2009.

[5] N. J. A. Sloane. On-line encyclopedia of integer sequences, sequence: A008277. Internet: http://oeis.org/A008277, [accessed 16-April-2017].

[6] N. J. A. Sloane. On-line encyclopedia of integer sequences, sequence: A008299. Internet: http://oeis.org/A008299, [accessed 16-April-2017].

[7] Štěpán Holub. Polynomial-time algorithm for fixed points of nontrivial morphisms. *Discrete Mathematics*, 309:5069–5076, 2009.

[8] Eric W. Weisstein. Bell number. from mathworld–a wolfram web resource. Internet: http://mathworld.wolfram.com/BellNumber.html,[accessed 16-April-2017].

[9] Eric W. Weisstein. Stirling number of the second kind. from mathworld–a wolfram web resource. Internet: http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html, [accessed 16-April-2017].