Implementing an algorithm for finding minimal forbidden words

Casper Christensen, 20117142

Master's Thesis, Computer Science

May 2015

Advisor: Christian Nørgaard Storm Pedersen

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis describes the theory, implementation and evaluation of algorithms for finding minimal forbidden words in a string. A minimal forbidden word is a string s' that does not occur in a string s but all proper substrings of s' occur in s.

There exists an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm by Crochemore et al, 1998 [4] which uses a factor automaton and according to Barton et al, 2014 [3] no implementation is publicly available. There also exists an $\mathcal{O}(n^2)$-time and $\mathcal{O}(n)$-space algorithm by Phino et al, 2009 [6] based on the construction of a suffix array. Baron et al, 2014 [3] introduces an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm, which is an improvement of Phino et al, 2009 [6].

This thesis' contribute by implementing the algorithm proposed by Crochemore et al, 1998 and examine if the algorithm has a linear running time as in theory. The thesis will also compare the running time of the two algorithms by Crochemore et al, 1998 [4] and Barton et al, 2014 [3].

The experiments show that the implementation of Crochemore et al, 1998 have a practical running time of $\mathcal{O}(n)$-time. The experiments also show that the implementation of Baron et al, 2014, is running about a factor 2 faster than the implementation introduced in this thesis.

The code is freely available at:

`http://users-cs.au.dk/casper91/master_thesis`

Acknowledgements

I would like to thank my supervisor Christian N. Storm Pedersen for his help, feedback, guidance and keeping me motivated throughout the development of this thesis. Thanks to Julie Marie B. Jørgensen for her support and proof reading.

Casper Christensen,
Aarhus, May 9, 2016.

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**
- 1.1 Outline . . . 2

2 Fundamentals **3**
- 2.1 Words and Factors . . . 3
- 2.2 Languages and Automata . . . 3
- 2.3 Factor Automaton . . . 4
- 2.4 Suffix array . . . 8
- 2.5 Longest Common Prefix Array . . . 8
- 2.6 Minimal Forbidden Words . . . 9

3 Algorithms **11**
- 3.1 Using Factor Automaton - **MFW** . . . 11
- 3.2 Using Suffix Array - **MAW** . . . 13

4 Implementation **17**
- 4.1 Description . . . 17
- 4.2 Usages . . . 18
- 4.3 Data . . . 18
- 4.4 Correctness . . . 20

5 Experiments **21**
- 5.1 Experimental Setup . . . 22
- 5.2 DAWG . . . 23
- 5.3 Finding MFW . . . 24
- 5.4 MFW vs MAW - Time . . . 25
- 5.5 MFW - Spikes . . . 26
- 5.6 Input/Output . . . 29

6 Conclusion **31**
- 6.1 Future Work . . . 31

Bibliography **33**

Chapter 1

Introduction

Today a lot of string theory focus on finding patterns in strings. In particular, in bioinformatics, sequence comparison is important. In bioinformatics sequence comparison usually consist of comparing DNA, RNA or proteins in order to identify similarities. It can also be used in order to reconstruct genomes and much more.

In this thesis focus will be on the complementary problem. Instead of looking at what is in a sequence/string the opposite will be investigated by looking at what is **not** in a string.

The purpose of this thesis is to find the minimal forbidden words/strings in a string, which could be a Genome, DNA sequence etc. Forbidden refers to words that does **not** exist in a the given string and minimal refers to the words that are not just an extension of another forbidden word.

There exists an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all forbidden words with a fixed-sized alphabet using a factor automaton by Crochemore et al, 1998 [4]. According to Barton et al, 2014 [3] an implementation of this algorithm does not exist. There also exists an $\mathcal{O}(n^2)$-time and $\mathcal{O}(n)$-space algorithm based on the construction of suffix arrays Phino et al, 2009 [6]. Most recent an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm also based on construction of suffix arrays Baron et al, 2014 [3].

The purpose of this thesis is to implement the algorithm from Crochemore et al, 1998 [4], and then investigate if the practical running time of the algorithm is as expected according to the theory. Lastly there will be a comparison of how well it compete against the newer implementation of Baron et al, 2014 [3].

1.1 Outline

Chapter 2. Fundamentals: introduces the fundamentals used in this thesis. The fundamentals consist of all the basic theory used for understanding the algorithms described in chapter 3.

Chapter 3. Algorithms: describes the two algorithms **MFW** (using factor automaton) and **MAW** (using suffix arrays). **MFW** is the algorithm implemented in this thesis and will be compared in practice against the algorithm **MAW**.

Chapter 4. Implementation: describes the details of the implementation of **MFW**. The usage of this implementation will also be described. Lastly the correctness of the program will be tested.

Chapter 5. Experiments: describes the experiments done throughout this thesis. The main experiments will investigate if the practical running time of **MFW** is as expected by the theory. They will also be investigating how well the two algorithms compare against each other based on performances.

Chapter 6. Conclusion: summarizes all the important aspect of this thesis in a final conclusion.

Chapter 2

Fundamentals

In this chapter the fundamentals and preliminary theory for finding minimal forbidden words will be described.

2.1 Words and Factors

An **alphabet**, denoted by Σ, is a finite set of symbols/letters. The size of Σ is denoted by $|\Sigma|$ and is constant. We denote Σ^* as a set of finite words of the alphabet Σ.

A **word/string**, denoted by $s \in \Sigma^*$ is a sequence of symbols/letters from the alphabet Σ. The size/length of s is denoted by $|s|$ or mostly used in this thesis as n. The empty word, denoted ϵ, has length zero.

A **factor**, also called a substring, is denoted by $s[i..j] = a_i a_{i+1}..a_{j-1}a_j$ where $0 \leq i \leq j$ and $i \leq j \leq n-1$, where n is the length $|s|$. If i and j is the same we denote it as $s[i]$ or s_i, where i is the ith position/index of the string s starting at zero. As an example, let $s = abcdefg$, then $s[2..4] = cde$ and $s[1] = b$.

A **suffix** is a substring where $j = n-1$ in $s[i..j]$. This gives us $s[i..n-1] = a_i a_{i+1}..a_{n-1}$ where $0 \leq i \leq n-1$. If $0 < i$ we call it a proper suffix. As an example, let $s = abcde$, then $s[3..4] = de$ is a proper suffix.

A **prefix** is a substring where $i = 0$ in $s[i..j]$. This gives us $s[0..j] = a_0 a_1..a_j$ where $0 \leq j \leq n-1$. If $j < n-1$ we call it a proper prefix. As an example, $s = abcde$, then $s[0..2] = abc$ is a proper prefix.

2.2 Languages and Automata

A **language** L is a subset of Σ^*, i.e. a set of strings over the alphabet Σ. L is finite if the set contains a finite number of strings.

An **Automaton**, denoted by A, consists of a 5-tuple $(Q, \Sigma, q_0, \delta, F)$.

- Q is the set of states.
- Σ is the alphabet.
- q_0 is the initial state, where $q_0 \in Q$

- δ is the transition function: $\delta : Q \times \Sigma \rightarrow Q$, which means it can go from state $p \in Q$ to state $q \in Q$ using symbol $a \in \Sigma$ if and only if such transition $(p, a, q) \in \delta$.

- F is the accepting states, where $F \subseteq Q$

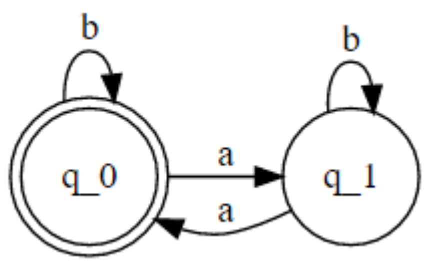
An automaton is deterministic if there for each state $p \in Q$ and each symbol $a \in \Sigma$ exists, at most, one state $q \in Q$, where $(p, a, q) \in \delta$.

An automaton reads a string, $w = a_0a_1..a_{n-1} \in \Sigma^*$, one character/symbol at a time starting at q_0. Then moving through a set of states $q_0, q_1..q_{n-1}$ where $q_i \in Q$ and $q_{i+1} = \delta(q_i, a_i)$, $1 \leq i < n - 1$. A string, $s \in \Sigma^*$ is accepted by the automaton if $q_{n-1} \in F$.

An automaton recognize a language $L \subseteq \Sigma^*$ if the set of all strings are accepted by the automaton.

As an example, let $\Sigma = \{a, b\}$ and $L \subseteq \Sigma^*$ be a language that consist of strings containing an even number of $a's$ e.g. {baa, aab, aba, etc}. Then we can build an automaton $Ea(L)$ that accepts this language. Such automaton can be seen on Figure 2.1.

Figure 2.1: An automaton accepting strings with an even number of a's.



2.3 Factor Automaton

The factor automaton of a string $s \in \Sigma^*$ is a minimal deterministic automaton that recognizes the langauge of the substrings of s. The factor automaton can be represented using a directed acyclic word graph (**DAWG**). In Chrochemore et al, 1994 [5] an algorithm for building a suffix-dawg is introduced, and it will briefly be described and explained in this chapter. This algorithm can be used to build a suffix automaton. The suffix automaton can then be changed into a factor automaton by setting all of its states to accepted states instead of only those that accept the suffixes. By doing so it is not necessarily a minimal automaton afterwards.

Figure 2.2 shows the suffix automaton of the string $s = ababb$. The first thing to do, in order to transform it into a factor automaton, is to set all states as accepted. After that it is easy to see that state 4 can be merged together with state 3 without destroying the purpose of the factor automaton. Figure 2.3 shows the result after merging the two states thus giving us a minimal factor

Figure 2.2: Suffix automaton of the string $s = abbabb$.

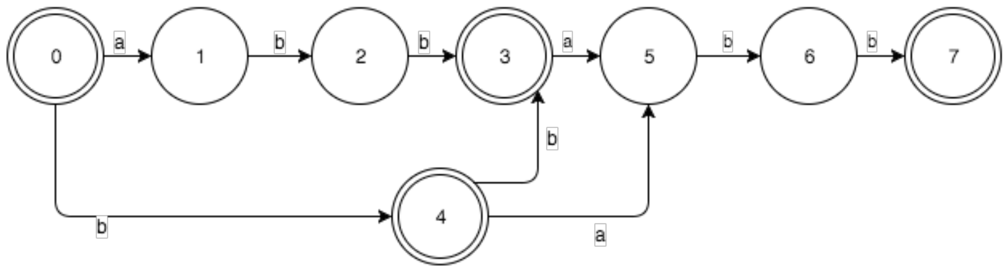
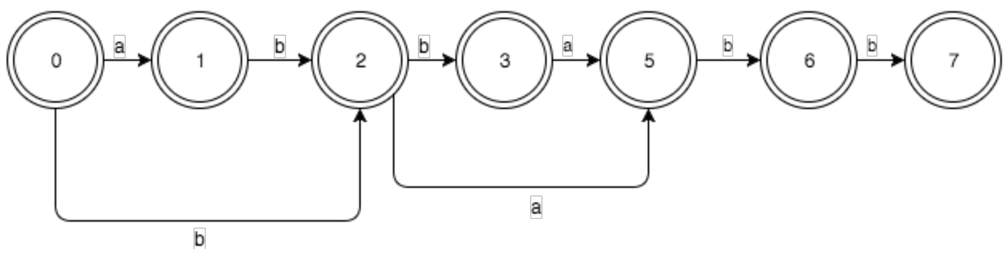


Figure 2.3: Factor automaton of the string $s = abbabb$, after making all states acceptable and minimizing the suffix automaton.



automaton. In this thesis the minimization step will not be used. Instead the suffix automaton with all states as accepting states will be used as a factor automaton.

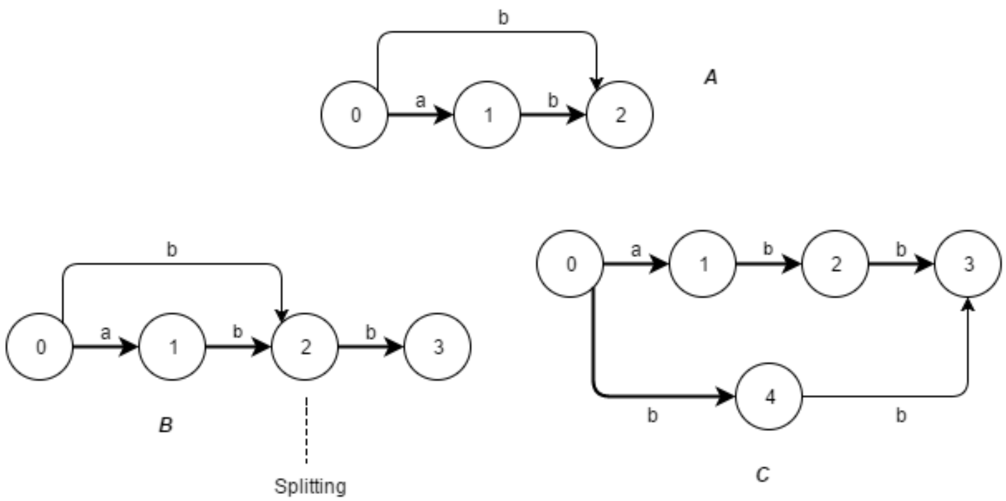
Chrochemore et al, 1994 [5] introduces two ways of constructing the **suffix-dawg**. The first is an offline algorithm which means it will build the data-structure based on the whole string. The other is an online algorithm which means that the data-structure is extended by one character at a time. In this thesis the online approach is used.

The algorithm **Factor-dawg** builds a representation of the factor automaton of a finite string s. In the algorithm two kind of transactions exist. A solid transaction cannot be changed and it is included in the longest path from the root to some state. A weak transaction may be changed during processing and can be used as a shortcut between states.

Another important part of the algorithm is splitting. Figure 2.4 shows an example of the transformation of the string ab into abb by appending the letter b. There are two substrings associated to the state 2, ab and b. In abb only the substring b is a suffix, therefore it is necessary to split state 2 into state 2 and 4.

All accepting states of the suffix automaton can be obtained by starting at the last state q and then go through all the suffixes using the suffix links generated during the process. The path will look something like: $q, suf(q), suf(suf(q)), .., q_0$, where q_0 is the initial state. This step is not done

Figure 2.4: Transformation of *ab* into *abb*. **A**: The initial automaton of *ab*. **B**: Shows the first step of algorithm. The splitting state is 2. **C**: Shows automaton of *abb* after the split.



in this thesis because it will be used as a factor automaton and then all states will be accepting states.

The complexity of the algorithm to build the factor automaton is $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space. The proof can be found in Chrochemore et al, 1994 [5].

Algorithm 1 Factor-dawg algorithm

```
FactorDawg(string s)
create initial_state
p := initial_state
suf[p] := nil

for i := 0 to |s| − 1 do
    a := s[i]
    create new state q
    make a solid transaction δ(p, a) := q

    w := suf[p]
    while w ≠ nil and δ(w, a) = nil do
        make a non-solid transaction δ(w, a) := q
        w := suf[w]
    end while

    v := δ(w, a)
    if w = nil then
        suf[q] := initial_state
    else if δ(w, a) = v is a solid transaction then
        suf[q] := v
    else
        SPLIT()
    end if

    p := q
end for
```

Algorithm 2 Factor-dawg SPLIT

```
SPLIT()
create new state q′
q′ has same transactions as v except that they are all non-solid
change δ(w, a) = q into a solid transaction δ(w, a) = q′
suf[q] := q′
suf[q′] := suf[v]
suf[v] := q′

w := suf[w]
while w ≠ nil and δ(w, a) is a solid transaction do
    change δ(w, a) = q into δ(w, a) = q′
    w := suf[w]
end while
```

2.4 Suffix array

A suffix array is a lexicographically sorted array of all suffixes of a string s denoted by SA. $SA[i]$ denotes the starting position of the ith suffix according to the lexicographical order.

As an example, let $s = bacaabb$. Table 2.4 shows a list of the suffixes of s, and i indicates the starting position of the suffix. Table 2.4 shows the suffixes in lexicographical order. Table 2.4 shows the resulting suffix array SA of s.

Suffix	i
bacaabb	0
acaabb	1
caabb	2
aabb	3
abb	4
bb	5
b	6
ϵ	7

Table 2.1: Suffixes of $s = bacaabb$.

Suffix	i
ϵ	7
aabb	3
abb	4
acaabb	1
b	6
bacaabb	0
bb	5
caabb	2

Table 2.2: Suffixes of $s = bacaabb$ in lexicographical order.

i	0	1	2	3	4	5	6	7
$SA[i]$	7	3	4	1	6	0	5	2

Table 2.3: The suffix array SA of s.

2.5 Longest Common Prefix Array

The longest common prefix array denoted by LCP is an array which contains the lengths of the longest common prefixes between all pairs of consecutive suffixes in a suffix array SA. $LCP(s_1, s_2)$ denotes the longest common prefix between the two suffixes s_1 and s_2. $LCP[0..n-1]$ is defined as: $LCP[0] =$ *undefined*. Undefined means that it has no value, but in general it is defined as 0. The rest is defined as: $LCP[i] = s[SA[i-1]..n-1, s[SA[i]..n-1]]$ for $1 \leq i < n$. This means that $LCP[i]$ contains the length of the longest common prefix of the ith lexicographical suffix and the predecessor in a suffix array.

As an example, $s = bacaabb$. Table 2.4 shows the suffix array SA for the string s. Table 2.5 shows the suffix array along with the corresponding suffixes. Next the LCP array is constructed by comparing consecutive suffixes according to the suffix array.

i	0	1	2	3	4	5	6	7
$SA[i]$	7	3	4	1	6	0	5	2
	ϵ	a	a	a	b	b	b	c
		a	b	c		b	b	a
		b	b	a		c		a
		b		a		a		b
				b		a		b
				b		b		
						b		

Table 2.4: SA with the corresponding suffixes.

i	0	1	2	3	4	5	6	7
$LCP[i]$	0	0	1	1	0	1	2	0

Table 2.5: LCP array of s.

2.6 Minimal Forbidden Words

Often when looking at string comparisons the problem is to find patterns or substrings in a given string. In this thesis the problem is the other way around. Instead of looking for substrings and patterns in a string, the absence of strings in a string will be investigated. The focus will be on the minimal forbidden words. The reason for only looking at the minimal forbidden words, is that a letter or character from the alphabet can be appended to a minimal word in order to construct a new string, that is not contained in the original string.

Let s be a string over the alphabet Σ, then a non-empty string $x = x_0x_1...x_{n-1}$ is a minimal forbidden word if and only if the two following conditions hold:

- x is not a substring of s
- both $x_0x_2...x_{n-2}$ and $x_1x_2...x_n - 1$ are substrings of s

This means that the string x is not in the original string s because otherwise it would not be absent. Secondly, all proper prefixes and proper suffixes is in the original string. Instead of checking all suffixes and prefixes it is sufficient just to check the longest proper prefix and longest proper suffix.

As an example, $s = abbabb$, then the minimal forbidden words of s is $\{aa, ba, bbb\}$. The string aaa is also a forbidden word of s, but it is not minimal because it can be constructed by appending the letter a to the minimal forbidden word aa.

Chapter 3

Algorithms

In this chapter the two algorithms for finding minimal forbidden words will be described. The first algorithm is the one introduced in Chrochemore et al, 1998 [4], which takes a factor automaton, and produce an automaton accepting the minimal forbidden words. The other algorithm is the one introduced in Barton et al, 2014 [3], which is based on the construction of suffix arrays.

3.1 Using Factor Automaton - MFW

In this section the algorithm introduced in Chrochemore et al, 1998 [4] will be explained, and the complete algorithm for finding minimal forbidden words produced in this thesis, will be put together.

The algorithm **MF-Trie** takes a factor automaton along with the suffix links and produces a new automaton, which accepts the minimal forbidden words. It is done by iterating all the states of the factor automaton using a BFS (Breadth-first search), and in each state iterating all of the symbols in the alphabet.

Algorithm 3 MF-trie algorithm

MFTrie(factor_automaton A(Q,Σ,q_o,δ), suffix_links *suf*)

for each state $p \in Q$ in BFS starting from q_0 **do**
  for each alphabet $a \in \Sigma$ **do**
    if $\delta(p, a)$ undefined **and** ($p = q_0$ **or** $\delta(suf(p), a)$ defined **then**
      set $\delta'(p, a) = new\_sink$
    else
      set $\delta'(p, a) = q$
    end if
  end for
end for

Recalling that a minimal forbidden word of s is a string s', which is **not** in the string s, but the longest prefix and longest suffix of s' **is** in s. At a state p there exists a substring s' (a path from q_0 to p) and a symbol a. Then the transition $\delta(p, a)$ has to be undefined, otherwise the string $s' + a$ would be in the string s. Then the prefix has to be in s, and by the definition of a factor automaton, the substring at state p is in fact in the string s. Then looking at the suffix of s' using the suffix link, the transition $\delta(\mathit{suf}(p), a)$ has to be defined, otherwise the suffix of s' will not be in the string s.

The complexity of the algorithm to build automaton accepting the minimal forbidden words is $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space. The proof can be found in Chrochemore et al, 1998 [4].

The algorithm **MFW** takes a string and finds all the minimal forbidden words. It is a combination of using the algorithm for building a factor automaton, along with the algorithm for transforming it into an automaton which accepts all the minimal forbidden words. In this thesis the implementation does not build the automaton which accepts all the minimal forbidden words, but instead it produces a list of all the minimal forbidden words of a given input string.

The complexity of the combined algorithm for building the factor automaton and finding the minimal forbidden words is still linear in time and space, but with a higher constant in Big-O notation.

Chapter 4 will describes more of the actual implementation of the **MFW** algorithm that has just been introduced.

Algorithm 4 MFW algorithm

MFW(string s)

Build factor automaton and suffix links of s.

list_of_minimal_forbidden_words := *empty*
for each state $p \in Q$ in BFS starting from q_0 **do**
    for each alphabet $a \in \Sigma$ **do**
        if $\delta(p, a)$ undefined **and** ($p = q_0$ **or** $\delta(\mathit{suf}(p), a)$ defined **then**
            append (path_to $p + a$) to list of minimal forbidden words
        end if
    end for
end for

return list_of_minimal_forbidden_words

3.2 Using Suffix Array - MAW

In this section the basics of the algorithm introduced in Barton et al, 2014 [3] will be explained. The algorithm is using a suffix array and a longest common prefix array. As mentioned previously a minimal forbidden word is a string s' of a string s, where s' is not a substring of s and all proper prefixes and proper suffixes are substrings of s.

A minimal forbidden word $x[0..m-1]$ of a string $s[0..n-1]$ is a substring whose proper substrings all occur in s. Two of them can be describes as $x_1 = x[1..m-1]$ and $x_2 = x[1..m-2]$, also written as using x_1, $x_2 = x_1[0..|x_1|-2]$. These two substrings can be used to characterize the minimal forbidden words.

Consider the set of letters that occur just before $x1$ and $x2$:

$$B_1(x_1) = \{S[i-1] : i \text{ is the starting postion of an occurrence of } x_1\}$$

$$B_2(x_2) = \{S[i-1] : i \text{ is the starting postion of an occurrence of } x_2\}$$

Lemma 1. *Let x and S be two strings. Then x is a minimal forbidden word if and only if $x[0] \in B_2(x_2)$ and $x[0] \notin B_1(x_1)$*

Lemma 2. *Let x be a minimal forbidden word of length m of string S of length n. Then there exists an integer $i \in [0..n-1]$ so that $S[SA[i]..SA[i]+LCP[i]] = x_1$ or $S[SA[i]..SA[i]+LCP[i+1]] = x_1$, where $x_1 = x[1..m-1]$.*

The proofs for Lemma 1 and 2 can be found in [3].

By using Lemma 2, all minimal forbidden words can be computed by looking at the substrings $s_i = S[SA[i]..SA[i]+LCP[i]]$ and $s_{i+1} = S[SA[i]..SA[i]+LCP[i+1]]$, for all $i \in [0..n-1]$. Then constructing the sets $B_1(s_j)$ and $B_2(s_j)$, for all $j \in [0..2n-1]$. By Lemma 1 the minimal forbidden words can be found by looking at the difference between $B_2(s_j)$ and $B_1(s_j)$, for all $j \in [0..2n-1]$.

In order to find all minimal forbidden words in linear time, the sets B_1 and B_2 has to be computed efficiently. This is done by visiting the suffix array and longest common prefix array twice, first in a top-down pass and afterwards in a bottom-up pass. The sets B_1 and B_2 are represented as an array of length $2n$ with a bit-vector of length $|\Sigma|$. 0 means that the ith letter is not in the set and 1 means that the ith letter is in the set. While iterating the SA and LCP, an *Interval* array is maintained. This array stores the letter encountered before the prefix of length l of $s[SA[i]..n-1]$ as a bit-vector of Σ. In the first iteration from top to bottom all letters that come before the occurrence of s_i and s_{i+1}, whose starting position appear before i in SA. In the last iteration from bottom to top the sets B_1 and B_2 will be finished by storing the letters that come before the occurrences, whose starting position appear after i in SA. In order to be efficient a stack called *LiFoLCP* is maintained to store the LCP values of the substrings, that are prefixes of the substring currently in scope. After the two sets have been computed, the sets will be compared. If there is a difference between the two sets, then by using Lemma 1, a minimal forbidden word can be constructed.

Algorithm 5 Top-down algorithm

```
Top-Down-Pass(s, n, SA, LCP, B_1, B_2, Σ)
Interval[0.. max_{i∈[0..n−1]} LCP[i]][0..|Σ| − 1] := 0
LiFoLCP.push(0)
for i := 0 to n − 1 do
    if i > 0 and LCP[i] < LCP[i − 1] then
        while LiFo.top()> LCP[i] do
            proxa := LiFo.pop()
            Interval[proxa][0..|Σ| − 1] := 0
        end while
        if LiFo.top() < LCP[i] then
            Interval[LCP[i]] := Interval[proxa]
        end if
        B_1[2i − 1] := Interval[proxa]
        B_2[2i − 1] := Interval[LCP[i]]
    end if
    if SA[i] > 0 then
        u := s[SA[i] − 1]
        value := LiFoLCP.top()
        while Interval[value][u] = 0 do
            Interval[value][u] := 1
            value := LiFoLCP.next()
        end while
        Interval[LCP[i]][u] := 1
        B_1[2i][u] := 1
        B_1[2i + 1][u] := 1
        B_2[2i][u] := 1
        B_2[2i + 1][u] := 1
    end if
    if i > 0 and LCP[i] > 0 and SA[i − 1] > 0 then
        v := s[SA[i − 1] − 1]
        Interval[LCP[i]][v] := 1
    end if
    B_2[2i] := Interval[LCP[i]]
    if LiFo.top() ≠ LCP[i]  then
        LiFoLCP.push(LCP[i])
    end if
end for
```

Algorithm 6 Bottom-up algorithm

```
Bottom-Up-Pass(n, SA, LCP, B_1, B_2, Σ)
Interval[0.. max_{i∈[0..n−1]} LCP[i]][0..|Σ| − 1] := 0
LiFoLCP.push(0)
for i := n − 1 to 0 do
    proxa := LCP[i] + 1
    proxb := 1
    if i < n − 1 and LCP[i] < LCP[i + 1] then
        while LiFoLCP.top() > LCP[i] do
            proxa := LiFoLCP.pop()
            LiFoRem.push(proxa)
        end while
        if LiFoLCP.top() < LCP[i] then
            Interval[LCP[i]] := Interval[proxa]
        end if
    end if
    for each k ∈ Σ, where B_1[2i][k] = 1 do
        value := LiFoLCP.top()
        while Interval[value][k] = 0 do
            Interval[value][k] := 1
            value := LiFoLCP.next()
        end while
        Interval[LCP[i]][k] := 1
    end for
    B_2[2i] := B_2[2i] | Interval[LCP[i]]
    B_2[2i + 1] := B_2[2i + 1] | Interval[LCP[i + 1]]
    B_1[2i + 1] := B_1[2i + 1]] | Interval[proxb]
    proxb := proxa
    B_1[2i] := B_1[2i] | Interval[proxa]
    while LiFoRem not empty do
        value := LiFoRem.pop()
        Interval[value][0..|Σ| − 1] := 0
    end while
    if LiFoLCP.top() ≠ LCP[i] then
        LiFoLCP.push(LCP[i])
    end if
end for
```

Chapter 4

Implementation

In this chapter the important aspects of the implementation is described. The code and data files used in this thesis can be found at:

`http://users-cs.au.dk/casper91/master_thesis`

In order to better compete and compare the performance against **MAW** [3], which is written in **C**, the **MFW** implementation is written in **C++**. **MAW** only uses the alphabet containing $\{A, C, G, T, N\}$ so the same is done for **MFW**. All other characters than $\{A, C, G, T, N\}$, will be changed into an N.

4.1 Description

The implementation of **MFW** consists of 2 main parts, which is an implementation of the factor automaton introduced in Chapter 2.3, along with the implementation of MF-Trie introduced in Chapter 3.1. Then of course the main program **MFW** ties it all together as shown in Chapter 3.1.

The implementation of the factor automaton has been done according to the algorithm without any modifications. The structure of the factor automaton is implemented by using a transition matrix in order to make lookup constant. In the program this is expressed as a 2-dimensional matrix, using **vectors**, containing pairs. The pairs tell which vertex it is pointing to, and if the edge is solid or not.

The implementation of MF-Trie is a little bit different than the algorithm, since the algorithm is building an automaton accepting the minimal forbidden words. In this thesis the focus is to find the minimal forbidden words. Therefore the implementation do not build the automaton, but instead it finds the minimal forbidden words and print them to a file. Each state has been extended to hold the path from the initial state, and to some state p while iterating the states in order to keep track of the substrings.

The complete implementation of **MFW** consists of combining the parts together. When working with large data it takes time to copy the data and pass it to functions. In order to not make this a bottleneck, most of the functions uses references instead of copying the data.

The **helper__class** consists of helpful methods such as:

findSigma
: Method used in order to determine the sigma/alphabet used.

getTime
: Method used for measuring running time.

readInput
: Method used for reading the input file.

4.2 Usages

The code and how to use **MAW** can be found at:

`https://github.com/solonas13/maw`

In order to use **MFW** it is necessary to compile the program using `make` or `make mfw` from the extracted folder. **MFW** can be run using the command `./mfw` with a set of options. The options are:

-i
: Used for specifying the path to the input file. This is mandatory. The input file should be in **fasta** [1] format and can only take a single fasta sequence.

-o
: Used for specifying the output file. This is optional.

-k
: Used for specifying the minimum length of the minimal forbidden words. This is optional. If **-K** is not set, the length will be equal to **-k**.

-K
: Used for specifying the maximum length of the minimal forbidden words. This is optional. If **-k** is not set, the length will be equal to **-K**.

As an example: `./mfw -i genome.fa -o genome.out -k 8 -K 12`. This will find all minimal forbidden words of length l, where $8 \leq l \leq 12$, in `genome.fa` and write them to `genome.out`.

4.3 Data

The data used in this thesis consists of different genomes, which are listed in Table 4.1. All the data is obtained from the NCBI database found at:

`http://www.ncbi.nlm.nih.gov/nuccore/`

Nucleotide	**Abbreviation**	**Genome reference**	**Size(bp)**
Arabidopsis thaliana chromosome 1	at_1	NC_003070.9	30427671
Arabidopsis thaliana chromosome 1 bottom arm	at_1_ba	AE005173.1	14668883
Arabidopsis thaliana chromosome 1 top arm	at_1_ta	AE005172.1	14221815
Arabidopsis thaliana chromosome 2	at_2	NC_003071.7	19698289
Arabidopsis thaliana chromosome 3	at_3	NC_003074.8	23459830
Arabidopsis thaliana chromosome 4	at_4	NC_003075.7	18585056
Arabidopsis thaliana chromosome 5	at_5	NC_003076.8	26975502
Caenorhabditis briggsae chromosome I	cb_1	NC_013489.1	11274843
Caenorhabditis briggsae chromosome II	cb_2	NC_013486.1	14512975
Caenorhabditis briggsae chromosome III	cb_3	NC_013490.1	13544562
Caenorhabditis briggsae chromosome IV	cb_4	NC_013487.1	15290274
Caenorhabditis briggsae chromosome V	cb_5	NC_013488.1	16004101
Caenorhabditis briggsae chromosome X	cb_10	NC_013491.1	20608032
Drosophila melanogaster chromosome 2L	dm_2l	NT_033779.5	23513712
Drosophila melanogaster chromosome 2R	dm_2r	NT_033778.4	25286936
Drosophila melanogaster chromosome 3L	dm_3l	NT_037436.4	28110227
Drosophila melanogaster chromosome 3R	dm_3r	NT_033777.3	32079331
Drosophila melanogaster chromosome X	dm_x	NC_004354.4	23542271
Bacillus anthracis str. Ames chromosome	ba	NC_003997.3	5227293
Bacillus subtilis subsp. subtilis str. 168 chromosome	bs	NC_000964.3	4215606
Escherichia coli str. K-12 substr. MG1655	ec	NC_000913.3	4641652
Lactococcus lactis subsp. lactis Il1403 chromosome	ll	NC_002662.1	2365589
Mycoplasma genitalium G37	mg	NC_000908.2	580076
Myxococcus stipitatus DSM 14675	ms	NC_020126.1	10350586
Pseudomonas aeruginosa DNA, strain: NCGM257	pa	NZ_AP014651.1	7090694
Streptomyces coelicolor A3(2)	sc	AL645882.2	8667507
Oryza sativa (japonica cultivar-group) chromosome 3	os_3	DP000009.2	36340990
Schistosoma mansoni strain Puerto Rico chromosome 2	sm_2	HE601625.1	34464480

Table 4.1: Data used in this thesis.

4.4 Correctness

The correctness of the implementation is tested by comparing it against the implementation of Barton et al, 2009[3], which here is denoted by **MAW**. More precisely, the number of minimal forbidden words of lengths 8, 13, 18 and 23 were counted for some of the genomes listed in Table 4.1.

Table 4.2 shows the number of minimal forbidden words for the selected genomes. The length of the minimal forbidden words are denoted by $\mathbf{M_8}$, $\mathbf{M_{13}}$, $\mathbf{M_{18}}$ and $\mathbf{M_{23}}$. The results shows the same for both algorithms, suggestion that the implementation of **MFW** is correct.

		MFW				MAW			
Sequence	**Size (bp)**	$\mathbf{M_8}$	$\mathbf{M_{13}}$	$\mathbf{M_{18}}$	$\mathbf{M_{23}}$	$\mathbf{M_8}$	$\mathbf{M_{13}}$	$\mathbf{M_{18}}$	$\mathbf{M_{23}}$
ba	5227293	30	2639373	27155	325	30	2639373	27155	325
bs	4215606	4	2104434	9503	64	4	2104434	9503	64
ec	4641652	168	2390324	11029	339	168	2390324	11029	339
ll	2365589	406	943147	9346	260	406	943147	9346	260
mg	580076	8660	150453	951	42	8660	150453	951	42

Table 4.2: Number of minimal forbidden words of lengths 8, 13, 18, 23 in the selected genomes using MFW and MAW.

Chapter 5

Experiments

The previous chapters have been describing and explaining the theory and implementation of finding minimal forbidden words. In this chapter the experiments and the result will be presented. The experiments consist of comparing the theoretical running time against the practical running time, and comparing **MFW** with **MAW**. The experiments are divided into sections. Each section presents the experiment along with the result.

DAWG
: This experiment will investigate the running time for constructing the factor automaton which is used to find the minimal forbidden words. The purpose of the experiment is to check whether the practical running time of the algorithm implemented in this thesis, agrees with the theoretical running time.

Finding MFW
: This experiment will investigate the running time for finding the minimal forbidden words. The purpose of the experiment is to check whether the practical running time of the algorithm implemented in this thesis, agrees with the theoretical running time.

MFW vs MAW - Time
: This experiment will investigate how well the implementation of this thesis' algorithm for finding minimal forbidden words performs compared to **MAW**.

DAWG - Spike
: This experiment will investigate the sudden increase in the running time for building the factor automaton which is seen in Figure 5.1.

Before explaining each experiment further, the experimental setup will be described.

5.1 Experimental Setup

All experiments have been run on a MAC computer with an Intel Core i5 2.6 GHz CPU and 8 GB RAM running OS X El Capitan 10.11.2. The code has been compiled using $g++$ (`Apple LLVM version 7.3.0 (clang-703.0.29)`) with optimization flag `O3`.

Each test has been run 5 times. The points in the plot then corresponds to an average/mean of the 5 runs. This is done to even out spikes while running the test and give a more representative result.

The running time has been measured using **time.h**. Table 5.1 shows the measured running times of each experiment.

Not all the minimal forbidden words found will be printed. Only the minimal forbidden words of length between 2 and 8 is printed to the output file. This will be the same for both **MFW** and **MAW**.

Nucleotide	**Building DAWG**	**Finding MFW**	**MFW_Total**	**MAW**
at_1	36,6	28,9	65,5	25,1
at_1_ba	16,5	12,7	29,2	12,6
at_1_ta	16,0	11,5	27,5	11,1
at_2	20,7	16,5	37,2	15,6
at_3	30,7	21,5	52,3	19,3
at_4	19,6	15,7	35,3	15,4
at_5	33,8	25,2	59,0	21,7
cb_1	13,3	9,6	22,9	8,4
cb_2	15,9	12,6	28,5	11,0
cb_3	15,0	11,7	26,6	10,2
cb_4	16,5	13,2	29,7	11,6
cb_5	17,4	13,7	31,1	12,4
cb_X	27,8	18,2	46,0	12,4
dm_2l	31,4	22,0	53,4	18,9
dm_2r	32,5	23,5	56,0	20,8
dm_3l	35,0	27,8	62,8	22,5
dm_3r	37,4	30,3	67,7	26
dm_x	30,3	21,7	52,0	18,8
ba	5,9	3,4	9,3	4,0
bs	4,0	2,7	6,7	3,1
ec	5,2	2,9	7,1	3,4
ll	2,0	1,5	3,5	1,7
mg	0,5	0,4	0,9	0,4
ms	12,1	7,5	19,6	8,9
pa	7,2	4,9	12,1	5,3
sc	8,4	6,3	14,7	6,5
os_3	40,6	36,8	77,4	29,8
sm_2	37,8	38,4	76,2	29,0

Table 5.1: Running time (seconds) of the algorithms on the input.

5.2 DAWG

In this section the the theoretical running time will be compared to the practical running time of the algorithm implemented in this thesis for building the DAWG, which represents the factor automaton. The theoretical running time of the algorithm is linear $O(n)$.

Figure 5.1 shows the running time for building the factor automaton on the input sequences from Table 4.1. Since the algorithm has theoretically linear running time, a linear line would be expected.

The graph shows as expected a linear line, but with a slight increase of the running time around the sequences of length 20 million. This will be looked further into in section 5.5. Nevertheless this is a very good indication that the implementation is actually running linear in the input size.

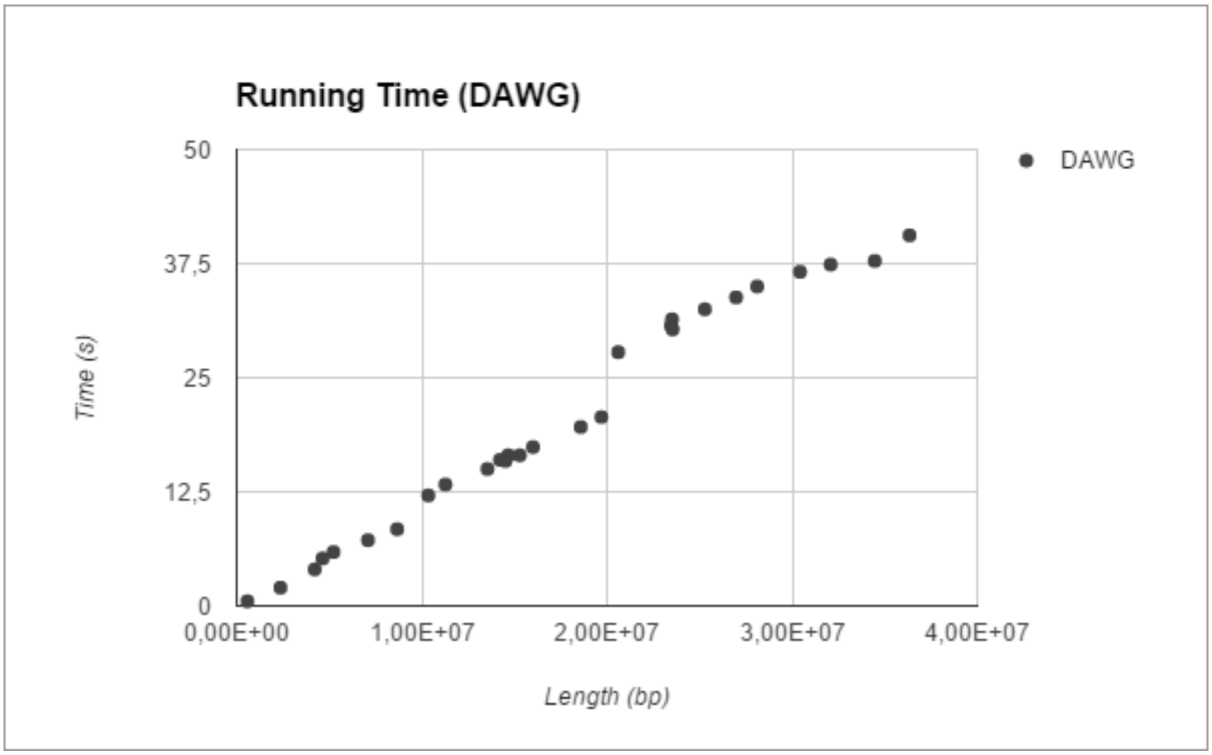


Figure 5.1: Running time of building DAWG.

5.3 Finding MFW

In this section the theoretical running time will be compared to the practical running time of the algorithm implemented in this thesis for finding minimal forbidden words. The theoretical running time of the algorithm is linear $O(n)$.

Figure 5.2 shows the running time for finding the minimal forbidden words on the input sequences from Table 4.1. The algorithm has a theoretically linear running time, and therefore a linear line would be expected.

The graph shows as expected a linear line. It seems like one of the largest data is a little bit slower than expected. Nevertheless this is a very good indication that the actual running time of the algorithm is indeed linear.

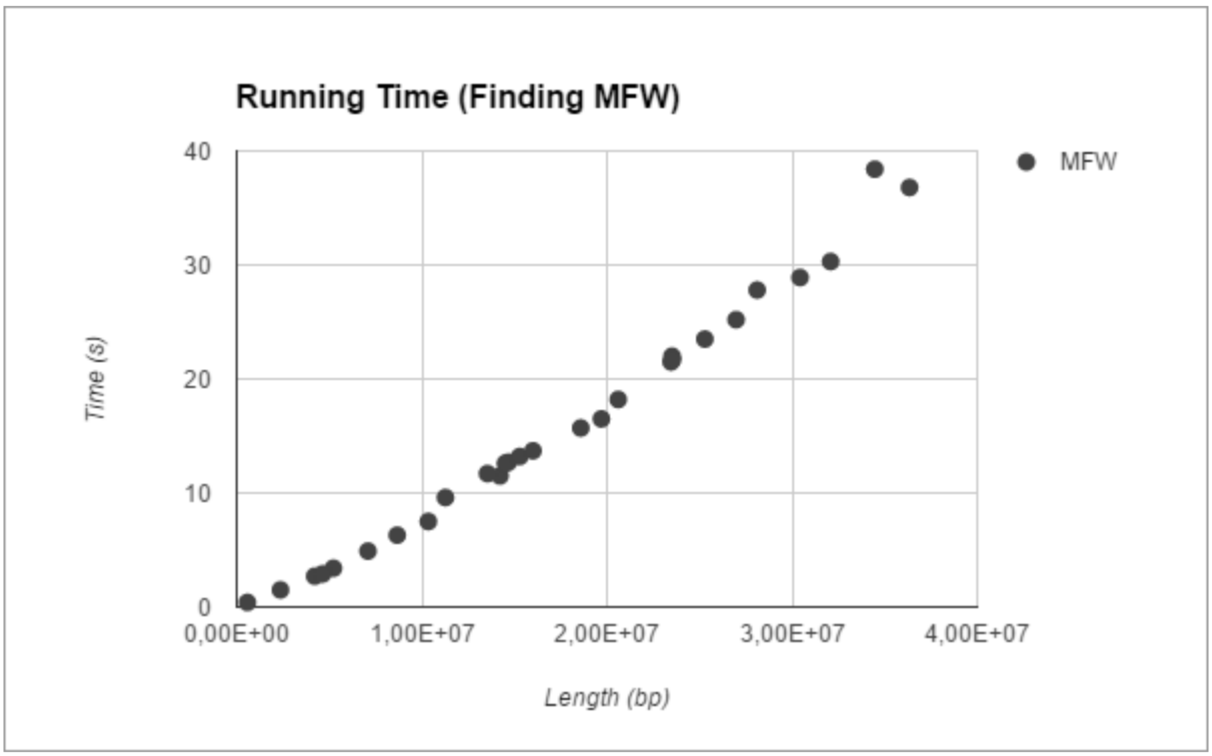


Figure 5.2: Running time of finding MFW.

5.4 MFW vs MAW - Time

In this section the running time of **MFW** and **MAW** will be compared in order to see how well they compete against each other. Figure 5.3 shows the running time of the complete **MFW** and **MAW**. As expected the graph shows that both algorithms run in linear time. It is clear to see that **MAW** is running a factor of 2-3 faster than **MFW**. Even thought both algorithms has an expected linear running time, **MAW** has a lower constant than **MFW**.

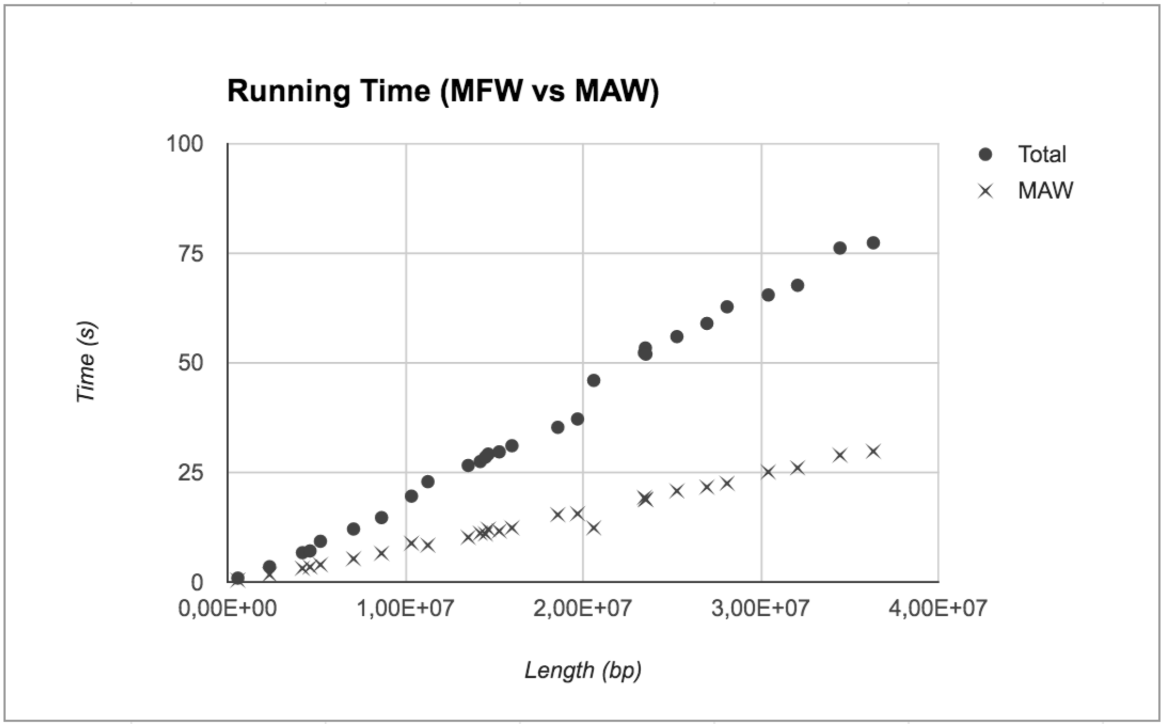


Figure 5.3: Running time of **MFW** and **MAW**.

Another experiment shows the result of taking the running time for building the factor automaton out of the equation. Figure 5.4 shows the running time of **MAW** and **MFW**, if the DAWG was build beforehand. It is clear to see that **MAW** is still slightly better than **MFW** alone.

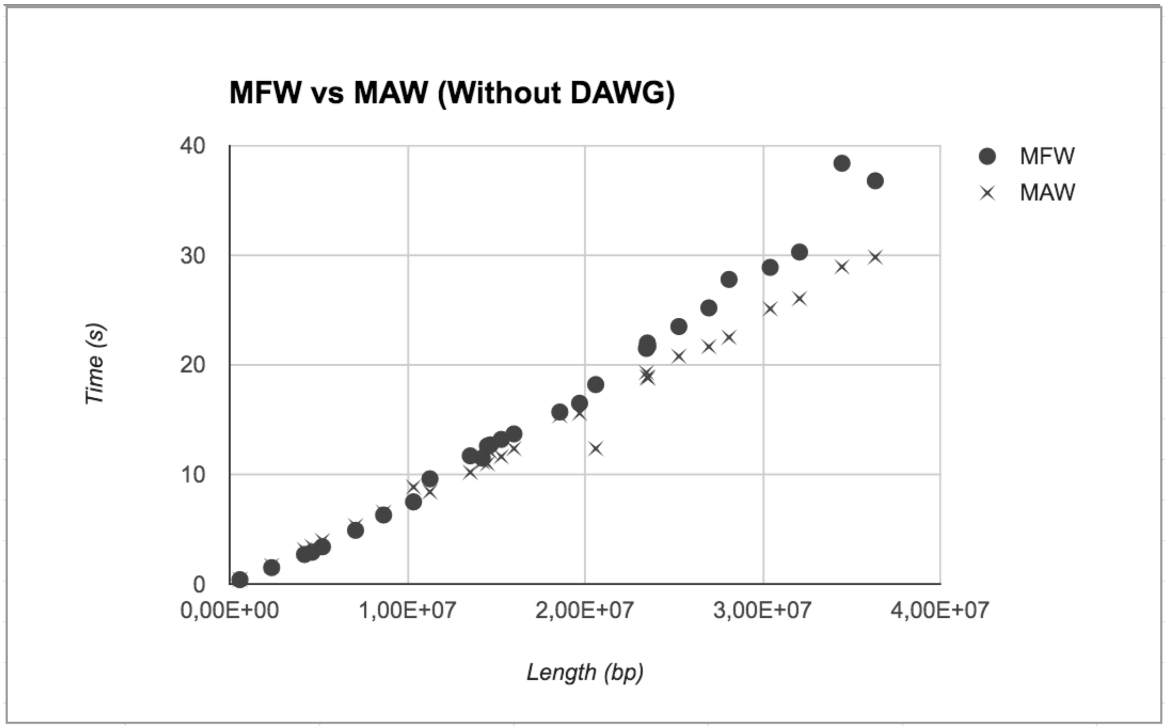


Figure 5.4: Running time of **MFW** without DAWG creation and **MAW**.

5.5 MFW - Spikes

In this section the sudden increase of running time on Figure 5.1 will be investigated. The main suspicion is due to space consumption/memory usages, since the implementation is dealing with a huge amount of data. Therefore a good place to start is to make a space consumption profile. This is done using **Valgrind** [2]. Valgrind is a tool for program analysis like detecting cache misses, branch misprediction, memory leaks etc. Here it is used to detect the space consumption of the program on the input data, to measure how much memory is used. Figure 5.5 shows the space consumption on the differently sized inputs.

It is easy to see that when the input size gets over about 20 million, it gains a massive increase of memory usage. It leaps from around 3 GB to 6 GB, which is a doubling of memory usage. This can very well be the explanation of why the running time for building the factor automaton on Figure 5.1 is increasing around the input size of 20 million.

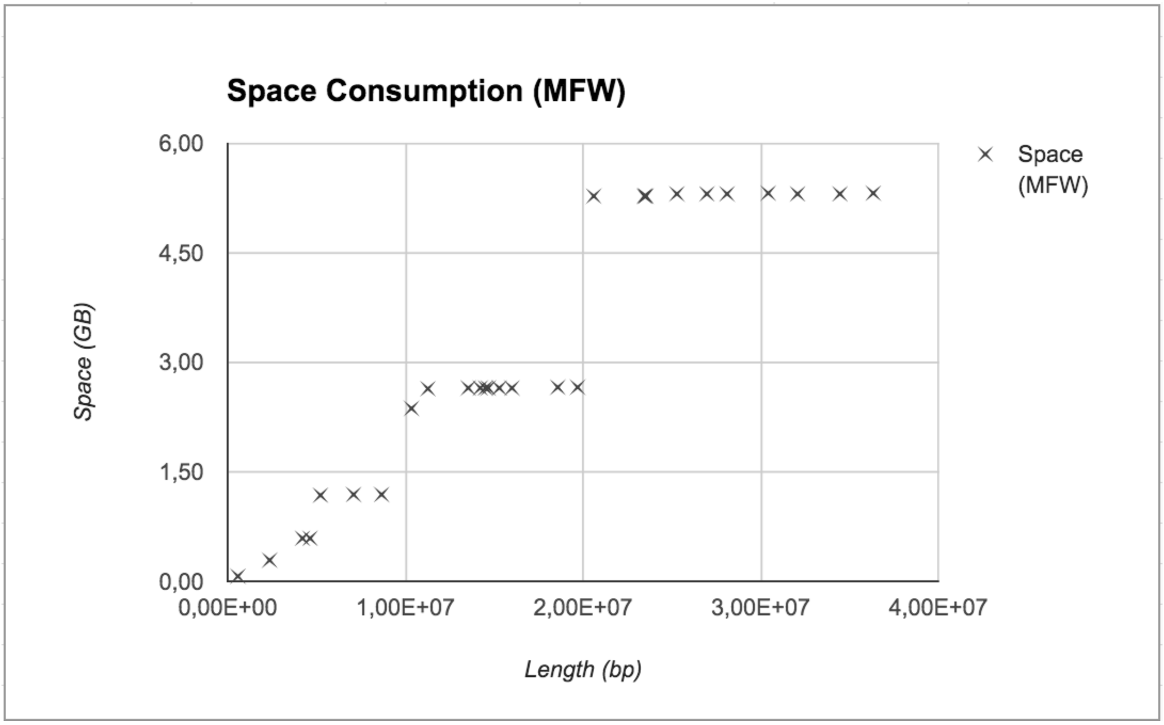


Figure 5.5: Space consumption of **MFW**.

The reason why the size increases drastically is that the implementation of the factor automaton is using **vectors** to represent the automaton. Then, because the size of the resulting factor automaton is unknown, the **push_back** method is used to add a new state. This means that the vector dynamically increases the memory size, when it is about to run out of the previously allocated memory. In most cases, when a vector is going to reallocate memory size, it is done by doubling the memory size, similar to what is seen in Figure 5.5. This means that when the input size becomes bigger than 20 million, it needs to reallocate a lot of memory in order to store the factor automaton, hence the increase in running time.

So in order not to use a lot of time reallocating memory etc., it will be better to allocate the memory at the beginning of the implementation. At the beginning, the algorithm does not know how big the factor automaton will be. However it can be at max $2n \times |\Sigma|$, because the algorithm at most increases the automaton with 2 states for each letter.

In the earlier version of the implementation for finding the minimal forbidden words, while doing the BFS, the BFS path was pushed around as a string. This is also inefficient. Instead the factor automaton is extended to contain an index i for each state, which represents the prefix $s_0 s_1 .. s_i$. This way the path do not have to be pushed around during the BFS because the substring can be obtained using the index and length of the BFS path.

Figure 5.6 shows the running time of the new improved version of **MFW** called **MFW_IMP** next to the old **MFW** and **MAW**. The new improved implementation is approximately 1/3 times faster than the old **MFW** and is now only about a factor 2 slower than **MAW**.

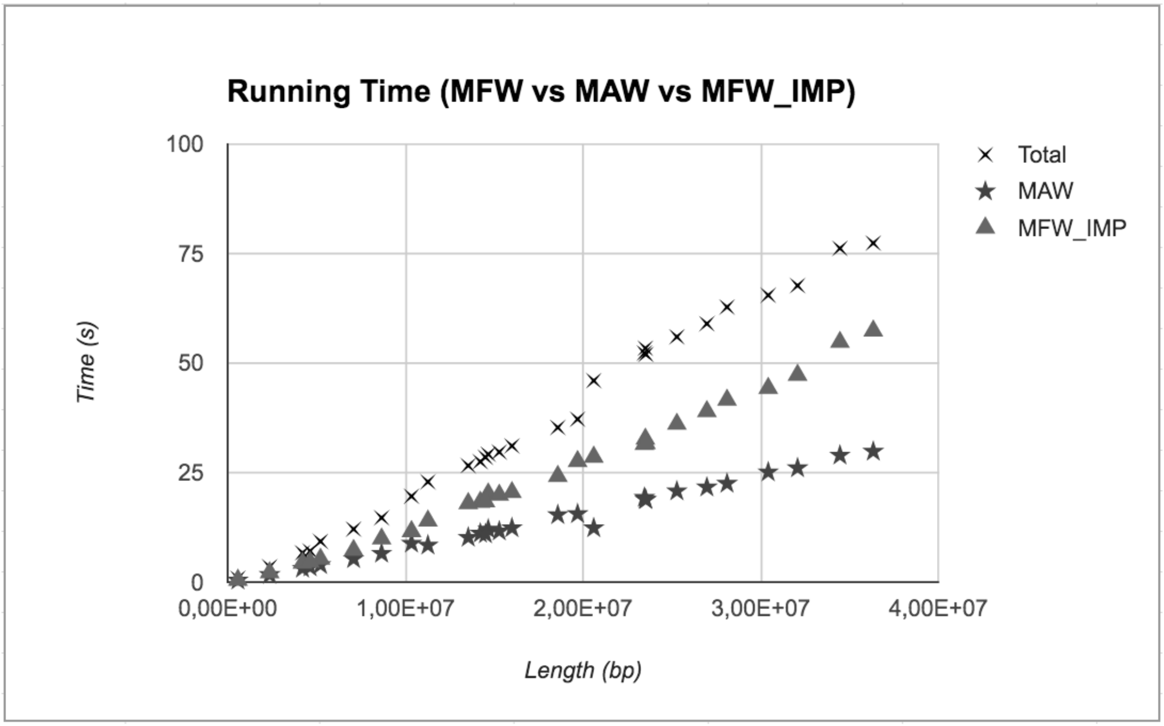


Figure 5.6: Running time of **MFW**, **MAW**, **MFW_IMP**.

5.6 Input/Output

In this section the input and output for the algorithm introduced in this thesis, will briefly be discussed. If Σ of the input only consists of one letter a and the input sequence consists of a sequence of a's, then the factor automaton will consist of $n + 1$ states and a transition between each state will be labeled a. The algorithm for finding minimal forbidden words will then iterate each state and check for minimal forbidden words. Since Σ only consists of a, there do not exist any minimal forbidden words. Therefore this input will be a best case for the algorithm.

Describing the worst input for the algorithm is a bit more difficult. The algorithm for finding the minimal forbidden words is bound to the number of states of the factor automaton. Therefore the algorithm for building the factor automaton needs to make a lot of splits. A split will add an extra state and increase the size of the automaton. An input that would produce a lot of splits has a long sequence of the same letters as a suffix. Let Σ consist of a, b. Then the input *abbbbbbbb* will produce a lot of splits and increase the size of the automaton. The longer the sequence of b's, the bigger the automaton gets. Another thing that will increase the running time of the algorithm, is to have a lot of updates of the suffix links and redirections of links while building the automaton. Table 5.2 shows a little experiment testing 5 different inputs. The first input A consists of an increasing sequence of C's which all starts with a single A. Next input B consists of only A's. Input C consist of a single C followed by A's. Input D is a random sequence of A's and C's. Last input E consists of increasing lengths of A's and C's. All inputs have a total length of 10,000,000 (10 million) letters. The experiment shows that the best case is input B, as expected, because it does not make any splits. Next the experiment shows that the worst is input C which produces the biggest automaton, however it is not the slowest. The slowest is the random input D. This means that it is most likely spending more time updating suffix links and redirecting.

Input	**Description**	**Time**(s)	**Size(Fac_auto)**(states)
A	$ACACCACCC$ etc.	6,7	19999997
B	Only $A's$	3,7	10000001
C	C followed by $A's$	6,7	19999999
D	Random of A and C	10,7	19999956
E	$ACAACCAAACCC$ etc.	6,7	19999996

Table 5.2: Running time and size of the automaton after running the algorithm for building the factor automaton.

The output consists of the minimal forbidden words. How many can be expected in the worst case? Since the algorithm is running in linear time, there can only be a linear amount of forbidden words. In the worst case it is possible to look at all the substrings in a string. This would yield $O(n^2)$ substrings. However, since the algorithms can find and report all minimal forbidden words in linear time, there can only exist a linear amount of minimal forbidden words.

Chapter 6

Conclusion

In this thesis the problem and theory for finding all minimal forbidden words of a string has been explained.

Chrochemore et al, 1998 [4] introduces an algorithm for finding all minimal forbidden words of a string. This algorithm has been implemented in this thesis and the practical running time has been compared to the theoretical running time. Barton et al, 2014 [3] also introduces an algorithm for the same problem. This algorithm has briefly been described. These two algorithms has also been compared against each other based on performance.

The experiments show that this thesis implementation of the algorithm by Chrochemore et al, 1998 [4] has a running time of $\mathcal{O}(n)$ on the input size, which agrees with the theoretical running time which is linear.

The experiment comparing the implementation of this thesis and the implementation by Barton et al, 2014 [3] shows that the algorithm implemented in this thesis, is running around a factor of 2 slower than the implementation of Barton et al, 2014 [3].

However the algorithm introduced in this thesis seems easier to understand and to implement. No heavy optimization has been done either. Therefore a trade-off of a factor 2 does not seem that bad, based on the input sizes tested in this thesis.

6.1 Future Work

In this thesis no heavy optimization has been done. It might be possible to reduce the running time by optimizing the code. Below is some suggestions that may or may not increase the performance of the algorithm.

Data-structure

The implementation used a lot of memory for storing the factor automaton etc. Therefore it might be possible to reduce the memory size by using a more efficient data-structure or a more efficient library than the c++ standard library.

Minimal factor automaton

As mentioned in Chapter 2 it is not guaranteed that the factor automaton that has been build is a minimal automaton. The algorithm for finding all minimal forbidden words is bound by the size of the factor automaton. Therefore by making the factor automaton a minimal automaton, the number of states will be reduced. However making it minimal will cost time. Therefore it cannot take more time to make the automaton minimal than the possible decrease in the total running time of the algorithm.

Bibliography

[1] Fasta format - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/FASTA_format`. (Accessed on 04/28/2016).

[2] Valgrind home. `http://valgrind.org/`. (Accessed on 04/24/2016).

[3] Carl Barton, Alice Heliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15(1):1–10, 2014.

[4] M. Crochemore, F. Mignosi, and A. Restivo. *Mathematical Foundations of Computer Science 1998: 23rd International Symposium, MFCS'98 Brno, Czech Republic, August 24–28, 1998 Proceedings*, chapter Minimal forbidden words and factor automata, pages 665–673. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[5] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1994.

[6] Armando J. Pinho, Paulo JSG Ferreira, Sara P. Garcia, and João MOS Rodrigues. On finding minimal absent words. *BMC Bioinformatics*, 10(1):1–11, 2009.