MAXIMAL COVER ALGORITHM IMPLEMENTATION

Efficient Implementation & Application of Maximal String Covering Algorithm

By HOLLY KOPONEN, B.SC.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Masters of Science

McMaster University © Copyright by Holly Koponen April 22, 2022

McMaster University
Masters of Science (2022)
Hamilton, Ontario (Department of Computing and Software)

TITLE: Efficient Implementation & Application of
Maximal String Covering Algorithm
AUTHOR: Holly Koponen (McMaster University)
SUPERVISOR: Dr. William F. Smyth & Dr. Neerja Mhaskar
NUMBER OF PAGES: xi, 60

Lay Abstract

This thesis deals with a simple yet essential data structure called a ***string***, a sequence of symbols drawn from an ***alphabet***. For example, a DNA sequence is a string comprised of four letters.

We describe a new software called MAXCOVER that identifies ***maximal covers*** of a given string $\boldsymbol{x}$ (a repeating substring that *'covers'* the most positions in $\boldsymbol{x}$). This software is based on the algorithms in [1]. We propose two new algorithms that perform faster in practice.

We also extended MAXCOVER for the closely related task of computing ***non-extendible repeats***. We compare this extension to the well-known MUMmer software (600+ citations). We find that MAXCOVER is many times faster than MUMmer with much lower space requirements and produces a more compact, exact and user-friendly output.

Abstract

This thesis describes the development and application of the new software MAXCOVER that computes ***maximal covers*** and ***non-extendible repeats*** (a.k.a. *"maximal repeats"*).

A ***string*** is a finite array $\boldsymbol{x}[1..n]$ of elements chosen from a set of totally ordered symbols called an ***alphabet***. A ***repeat*** is a substring that occurs at least twice in $\boldsymbol{x}$. A repeat is *left/right extendible* if every occurrence is preceded/followed by the same symbol; otherwise, it is *non-left/non-right extendible (NLE/NRE)*. A ***non-extendible*** (NE) repeat is both NLE and NRE. A repeat ***covers*** a position i if $\boldsymbol{x}[i]$ lies within the repeat. A ***maximal cover*** (a.k.a. *"optimal cover"*) is a repeat that covers the most positions in $\boldsymbol{x}$.

For simplicity, we first describe a quadratic $\mathcal{O}(n^2)$ implementation of MAXCOVER to compute all maximal covers of a given string based on the pseudocode given in [1]. Then, we consider the logarithmic $\mathcal{O}(n \log n)$ pseudocode in [1], in which we identify several errors. We leave a complete correction and implementation for future work. Instead, we propose two improved quadratic algorithms that, shown through experiments, will execute in linear time for the average case.

We perform a benchmark evaluation of MAXCOVER's performance and demonstrate its value to biologists in the protein context [2]. To do so, we develop an extension of MAXCOVER for the closely related task of computing NE repeats. Then, we compare MAXCOVER to the REPEAT-MATCH feature of the well-known MUMmer software [3] (600+ citations). We determine that MAXCOVER is an order-of-magnitude faster than MUMmer with much lower space requirements. We also show that MAXCOVER produces a more compact, exact, and user-friendly output that specifies the repeats.

Availability: Open source code, binaries, and test data are available on Github at `https://github.com/hollykoponen/MAXCOVER`. Currently runs on Linux, untested on other OS.

Acknowledgements

Thank you to my supervisors Dr. W.F. Smyth and Dr. Neerja Mhaskar for taking a leap of faith. Their guidance and has pushed me to produce the work behind this thesis, constantly driving me for better. Dr. Ridha Khedri who guided me in the beginning of my Masters studies. Dr. Brian Golding who provided insight from the biological perspective.

I would like express my gratitude towards my friends and family for supporting me on this journey. My parents gave my the foundation to study critically and thoroughly, while dreaming big. Thank you to my partner, Vitaliy, for keeping me sane during this process. I couldn't have done it without you. Thank you to my dear friend, Sam, for reading over my thesis and providing corrections and edits in the finale.

Contents

Lay Abstract iii

Abstract iv

Acknowledgements v

Acronyms xi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Contributions . . . 3
- 1.3 Overview . . . 4

2 Preliminaries 5
- 2.1 Definitions . . . 5
- 2.2 Arrays . . . 8
  - 2.2.1 Suffix Array $\mathcal{SA}$ . . . 8
  - 2.2.2 Longest Common Prefix $\mathcal{LCP}$ Array . . . 8
  - 2.2.3 Substring v_i . . . 8
  - 2.2.4 Border Array . . . 9
  - 2.2.5 Repeating Substring Frequency $\mathcal{RSF}$ Array . . . 9
  - 2.2.6 Overlapping Positions $\mathcal{OLP}$ Array . . . 10
  - 2.2.7 Repeating Substring Positions Covered $\mathcal{RSPC}$ Array . . . 10
- 2.3 Maximal Covers . . . 11
- 2.4 Data Analytics . . . 13

3 Applications to Data Analytics 14
- 3.1 Bioinformatics . . . 14
- 3.2 Information Security . . . 17
  - 3.2.1 Repeats in Ciphertext . . . 17
  - 3.2.2 Intrusion Detection System: Signature-based . . . 18
- 3.3 Image Analysis & Video Processing . . . 19
  - 3.3.1 Image Analysis . . . 19
  - 3.3.2 Video Recognition . . . 19

4 MAXCOVER 21
- 4.1 Software Architecture . . . 21

4.2 Data Structures . . . 24
4.2.1 R_1 & R_m . . . 24
4.2.2 Runs . . . 25
4.2.3 Exrun function . . . 26
4.3 Improved Quadratic Algorithm . . . 27
4.3.1 Algorithm 1 . . . 27
4.3.2 Algorithm 2 . . . 29
4.3.3 Runtime Comparison . . . 31

5 Experiments **36**
5.1 Machine Specifications . . . 36
5.2 Performance . . . 37
5.2.1 MUMmer . . . 37
5.2.2 Extension of MAXCOVER . . . 38
5.2.3 Protein Dataset . . . 39
5.2.4 MAXCOVER vs. MUMmer's REPEAT-MATCH . . . 40

6 Conclusion and Future Work **41**
6.1 Discussion . . . 41
6.2 Future Work . . . 42

A Tables **43**

B Pseudocode **50**

List of Figures

2.1 All runs $\boldsymbol{r}_z$ of $\boldsymbol{x} = kokokokkoko$. . . . 7
2.2 $\mathcal{SA}$, $\mathcal{LCP}$, and $\mathcal{RSF}$ Array of $\boldsymbol{x} = tartarus$. . . . 9
2.3 Border Array of $\boldsymbol{x} = tartarus$. . . . 9
2.4 $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, $\mathcal{OLP}$, and $\mathcal{RSPC}$ arrays of $\boldsymbol{x} = kokokokko$. . . . 10
2.5 An eligible run $\boldsymbol{r}$ of $\boldsymbol{u}$ where $\boldsymbol{u} = \boldsymbol{u}'\boldsymbol{u}'' = \boldsymbol{b}''\boldsymbol{b}'''\boldsymbol{b}'\boldsymbol{b}''$. . . . 12

3.1 BWM for $\boldsymbol{x} = abaaba\$$, where $BWT_{abaaba\$} = abba\$aa$ is the last column in bold. . . . 16

4.1 Simplified diagram for the software architecture of the MAXCOVER software to compute maximal covers and NE repeats. . . . 23
4.2 R_1 & R_m Arrays of $\boldsymbol{x} = kokokokko$. . . . 24
4.3 Plot of **Original** $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$. . . . 32
4.4 Plot of **Original** $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$. . . . 33
4.5 Plot of Improved $\mathcal{O}(n^2)$ Algorithm 1 using **While** loop Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$. . . . 33
4.6 Plot of Improved $\mathcal{O}(n^2)$ Algorithm 1 using **While** loop Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$. . . . 34
4.7 Plot of Improved $\mathcal{O}(n^2)$ Algorithm 2 using **Stack** Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$. . . . 34
4.8 Plot of Improved $\mathcal{O}(n^2)$ Algorithm 2 using **Stack** Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$. . . . 35

5.1 Output from MUMmer's REPEAT-MATCH of NE repeats for the string $\boldsymbol{x} = ADAQADADAQADAQADA$. . . . 38
5.2 MAXCOVER output of NE repeats for $\boldsymbol{x} = ADAQADADAQADAQADA$. 38

B.1 `Compute_Frequency()` computes the frequency $f_{r,\boldsymbol{u}}$. Correction of Line 2-4: Changed from -1 to $+1$. Correction of Line 13: Removed an extra round bracket. . . . 50
B.2 Pseudocode to compute the R_1 & R_m arrays. . . . 51

B.3 Pseudocode for EXRUN() procedure to compute eligible run. 51
B.4 RMQ() procedure to compute the Range Minimum Query of $\mathcal{LCP}$ or $\mathcal{LCS}$ using RMQ_SUCCINCT procedure in [4] 52
B.5 COMPUTE_RUNS() procedure to determine all runs of a given string $\boldsymbol{x}$. Based on algorithm DETECTRUNS(u) by Crochemore et al. in [5] 52
B.6 MAXBORDER() procedure to compute the Border Array of a given text, and return the length of the longest border. Code based on Algorithm 1.3.1 in [6]. 53
B.7 PROCESSSTACK() procedure to process the pairs $(index, r_1)$ on the stack. A near replica of Algorithm 1. 54

List of Tables

5.1 Comparison between MUMmer's `repeat-match` and MAXCOVER software to compute NE substrings on a random sample of 40,000 protein sequences. 40

A.1 Examples of maximal covers for *C. elegans* proteins within a particular range of maximal cover lengths computed using MAXCOVER. 43
A.2 Examples of maximal covers for *D. melanogaster* proteins within a particular range of maximal cover lengths computed using MAXCOVER. . . 44
A.3 Examples of maximal covers for Arabidopsis proteins within a particular range of maximal cover lengths computed using MAXCOVER. 45
A.4 Examples of maximal covers for human proteins within a particular range of maximal cover lengths computed using MAXCOVER. 46
A.5 Original $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds 47
A.6 Improved $\mathcal{O}(n^2)$ Algorithm using While loop Runtime in Seconds 48
A.7 Improved $\mathcal{O}(n^2)$ Algorithm using Stack Runtime in Seconds 49

Acronyms

LCP Longest Common Prefix

NE repeat non-extendible repeat

NLE non-left extendible

NRE non-right extendible

OLP Overlapping Positions

RSF Repeating Substring Frequency

RSPC Repeating Substring Positions Covered

SA suffix array

Chapter 1

Introduction

A ***string*** is a finite array $\boldsymbol{x}[1..n]$ of elements chosen from a set of totally ordered symbols Σ, called an ***alphabet***. A ***substring*** is a string $\boldsymbol{u} = \boldsymbol{x}[i..j]$, where $1 \leq i \leq j \leq n$. A ***repeating substring*** $\boldsymbol{u}$ is one that occurs more than once in $\boldsymbol{x}$.

Algorithms that process these primary data structures are widely applicable to various fields, such as bioinformatics, information security, and image analysis [7]. Thus, it is imperative to identify specific types of strings and improve the runtime and space usage required to compute them. *(See Section 1.1 for more on the motivation.)*

This thesis refers to two specific types of repeating substrings called *maximal covers* and *NE repeats*. A ***maximal cover*** is a repeating substring $\boldsymbol{\mu}$ that '*covers*' a maximum number of positions in $\boldsymbol{x}$. For example, $\boldsymbol{x} = abgzabg$ has a maximal cover $\boldsymbol{\mu} = abg$, which occurs at $\boldsymbol{x}[0..2]$ and $\boldsymbol{x}[4..6]$. Therefore $\boldsymbol{\mu}$ covers 6 positions, which is the maximum number covered by any repeating substring $\boldsymbol{u}$ in $\boldsymbol{x}$.

A similar type of repeating substring that we refer to in this thesis is a *non-extendible repeat (NE repeat)*. A repeating substring $\boldsymbol{u}$ is ***left/right extendible*** if every instance of its occurrence is preceded/followed by the same symbol; otherwise, it is ***non-left/non-right extendible, NLE (NRE)***. A repeating substring is ***non-extendible (NE)*** if and only if it is both NLE and NRE. For example, in $\boldsymbol{x} = abgcabg$, the repeating substring $\boldsymbol{u}_1 = b$ is both left and right extendible because every occurrence is both preceded and followed by 'a' and 'g,' respectively. If we extend it, we find that $\boldsymbol{u}_2 = abg$ is NE, which also happens to be a maximal cover. *(See Section 2.1 for Definitions.)*

This thesis describes the development and application of the new software MAX-COVER which computes maximal covers for the first time, and computes NE repeats. This software was implemented to determine the usefulness and applicability of maximal covers and any significant features of these particular repeating substrings. *(See Section 5 on Experiments.)* During the experiments, we identified errors in the pseudocode that the software was based on in [1] to compute maximal covers and subsequently provided improved algorithms. *(See Section 4.3.1 and 4.3.2.)*

1.1 Motivation

Studying strings and the algorithms that process them is widely applicable to various fields of research. Hence, it is essential to identify string types and their uses as well as optimize the algorithms that compute them.

In 1990, Apostolico and Ehrenfeucht first introduced the concept of a *cover* using the term ***quasiperiodicity***. A string $\boldsymbol{x}$ is *quasiperiodic* if there exists a substring $\boldsymbol{u} \neq \boldsymbol{x}$ such that the occurrences of $\boldsymbol{u}$ *cover* $\boldsymbol{x}$ entirely (i.e. every position of $\boldsymbol{x}$ falls within some occurrence of $\boldsymbol{u}$) [8]. A generalized definition of *cover* does not need to cover all positions of $\boldsymbol{x}$, but covers only some positions, thus a ***partial-cover*** [9]. For example, $\boldsymbol{x} =$ '*ababa*' is entirely covered by $\boldsymbol{u} =$ '*aba*', but only partially covered by $\boldsymbol{u} =$ '*ba*'. Several types of covers have been defined, such as *k-cover* [10], *enhanced cover* [11], *α-partial cover* [9], *frequency cover* [12], and most recently *maximal cover* [1].

Our motivation for this thesis was to test the theoretical and practical usefulness of the latest concept: *maximal cover*. Covers give a large amount of information about the strings. One such benefit is that covers provide information about the structure of repeating patterns compactly. Thus, we were motivated to explore the usefulness of maximal covers in data compression.

In addition to testing the theoretical usefulness, we also wanted to test the practical applicability of maximal covers. A cover applies to any field where *regular* strings occur. This feature allowed us to test the applicability of maximal covers to study the patterns of different sequences (i.e. regular strings), including biological sequences.

1.2 Contributions

To study the usefulness of *maximal covers*, we implemented the first-ever software to compute them: MAXCOVER. Moreover, we based the software on pseudocode provided in [1] rather than [9]. This is because [1] uses suffix arrays instead of the slower and more space-consuming suffix trees in [9].

The pseudocode in [1] provides both a quadratic $\mathcal{O}(n^2)$ algorithm and a logarithmic $\mathcal{O}(n \log n)$ algorithm. Due to the algorithm's simplicity, we first implemented the quadratic version. Thus, we implemented data structures not provided in the pseudocode, namely: computing R1 and RM arrays, COMPUTE_RUNS() to compute a hash table of runs, and EXRUN() to compute all eligible runs *(See Section 4.2)*.

This incremental process allowed us to implement the more complex $\mathcal{O}(n \log n)$ version by reusing common data structures and testing the accuracy compared to the quadratic version. Through testing, we determined several errors in this pseudocode such that not all possible cases are covered. Thereby, the proof of this pseudocode in [1] is incomplete. We leave this problem to future work.

Therefore, we proposed two new $\mathcal{O}(n^2)$ algorithms to compute maximal covers of a string *(See Sections 4.3.1 and 4.3.2)*. In addition, we demonstrated that the two algorithms execute in linear time on average.

To benchmark the MAXCOVER software, we must compare it to existing software. However, since this is the first implementation to compute maximal covers, we decided to compare based on NE repeats, which are very similar to maximal covers. We identified a well-known and often cited software MUMmer, with the REPEAT-MATCH feature that computes NE repeats. Thus, we modified the MAXCOVER software to compute NE repeats. For this computation, we determined that MAXCOVER is an order-of-magnitude faster than MUMmer with much lower space requirements. We also show that MAXCOVER produces a more compact, exact, and user-friendly output that specifies the repeats.

1.3 Overview

First, this thesis goes into the background knowledge required to understand this thesis's material in Chapter 2, starting with *Definitions (Section 2.1)*; then *Maximal Covers (Section 2.3)* and finally, *Data Analytics (Section 2.4)*.

Chapter 3 talks about the *Applications* of string processing algorithms in various fields of Data Analytics, in particular, *Bioinformatics (Section 3.1)*, *Information Security (Section 3.2)*, and *Image Analysis and Video Recognition (Section 3.3)*.

Next, this thesis describes the MAXCOVER software in Chapter 4, starting with the *Software Architecture (Section 4.1)*; then describing the *Data Structures* implemented *(Section 4.2)*; concluding with two new *Improved Quadratic Algorithms* and an average-case runtime comparison *(Section 4.3.1 and 4.3.2)*.

Chapter 5, *Experiments*, begins with *Machine Specifications (Section 5.1)*; followed by a discussion on the *Performance (Section 5.2)* of MAXCOVER's extension to compute NE repeats compared to MUMmer's REPEAT-MATCH.

Finally, the thesis is completed by the *Conclusion* in Chapter 6, which has a *Discussion (Section 6.1)* summarizing the results of this thesis and a description of *Future Work (Section 6.2)*.

Chapter 2

Preliminaries

2.1 Definitions

A ***string*** is a finite array $\boldsymbol{x}[1..n]$ of elements (***letters***) chosen from a set of totally ordered symbols Σ, called an ***alphabet***. A ***substring*** is a string $\boldsymbol{u} = \boldsymbol{x}[i..j]$, where $1 \leq i \leq j \leq n$, which ***occurs*** at position i and has ***length*** $|\boldsymbol{u}| = j - i + 1$. A string with a length of zero is called an ***empty string*** ϵ. For example, with $\boldsymbol{x} = tartarus$, then the substring $\boldsymbol{u} = \boldsymbol{x}[2..4] = art$ occurs at position 2 of length $|u| = 3$.

This thesis only concerns *regular* strings. A string is ***regular*** when each letter in $\boldsymbol{x}$ represents only a single symbol from the alphabet. In contrast, an ***indeterminate*** string allows letters to represent any symbol from a subset of the alphabet [13]. For example, an indeterminate string can be *'{b,c}at'* to mean both *'bat'* and *'cat'*, which are regular strings by themselves.

A ***prefix*** of $\boldsymbol{x}$ is the substring $\boldsymbol{x}[1..j]$. Similarly, a ***suffix*** of $\boldsymbol{x}$ is the substring $\boldsymbol{x}[i..n]$. If the prefix/suffix is not the entire string $\boldsymbol{x}$ itself (i.e. $j \neq n$ and $i \neq 1$ respectively), then it is a ***proper prefix/suffix***. For example, $\boldsymbol{x} = tartarus$ has a proper prefix $\boldsymbol{u}_p = tart$ and the proper suffix $\boldsymbol{u}_s = us$.

A ***border*** of $\boldsymbol{x}$ is a substring that is both a proper prefix and a proper suffix of $\boldsymbol{x}$. For example, $\boldsymbol{x} = kokko$ has the substring $\boldsymbol{x}[1..2] = \boldsymbol{x}[4..5] = ko$, which is both a proper prefix and proper suffix for $\boldsymbol{x}$.

A ***repeating substring*** or ***repeat*** is a substring $\boldsymbol{u}$ where $\boldsymbol{u}$ occurs more than once in $\boldsymbol{x}$, i.e. the ***frequency*** $f_{x,u} > 1$, which is the number of times $\boldsymbol{u}$ occurs in $\boldsymbol{x}$. For example, with $\boldsymbol{x} = tartarus$, since $\boldsymbol{u} = tar$ occurs twice ($f_{x,u} = 2$) at $\boldsymbol{x}[1..3]$ and $\boldsymbol{x}[4..6]$, we say $\boldsymbol{u}$ is a repeating substring (a *repeat*) in $\boldsymbol{x}$.

We say a substring ***extends*** left/right if the pattern can continue in either direction in $\boldsymbol{x}$. A repeat $\boldsymbol{u}$ is left/right ***extendible*** if every instance of its occurrence is preceded/ followed by the same symbol; otherwise, it is ***non-left/non-right extendible (NLE/NRE)***. A repeating substring is ***non-extendible (NE)*** if and only if it is both NLE and NRE. For example, in $\boldsymbol{x} = tartarus$, the repeating substring *'a'* is both left

and right extendible because every occurrence is preceded and followed by *'t'* and *'r,'* respectively. If we extend it, we find that *'tar'* is NE. [1]

Instances of repeats can ***overlap*** with each other, i.e. $\boldsymbol{u} = x[g..h] = [i..j]$ where $i \leq h$. The total overlapped positions of $\boldsymbol{u}$ in $\boldsymbol{x}$ is the total positions shared by each instance of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, i.e. $h - i + |u|$. For example, with $\boldsymbol{x} = kokokokko$, then $\boldsymbol{u} = koko$ overlaps with itself at $\boldsymbol{u}_1 = \boldsymbol{x}[1..4]$ and $\boldsymbol{u}_2 = \boldsymbol{x}[3..6]$ to create $\boldsymbol{x}[1..6] = kokoko$. This means there is an overlap at $\boldsymbol{x}[3..4] = ko$ of 2 overlapping positions.

Repeats can occur in ***tandem*** (adjacent to each other), i.e. $\boldsymbol{x}[i..j] = \boldsymbol{uuu} = \boldsymbol{u}^e$, where ***exponent*** e refers to the number of instances of $\boldsymbol{u}$ that occurs in tandem. We refer to ***period*** p as the length of the repeat $\boldsymbol{u}$ that occurs in tandem in $\boldsymbol{x}$, i.e. $p = |u|$. If $\boldsymbol{x}$ is of the form $\boldsymbol{u}^e$ and $e > 1$, then $\boldsymbol{x}$ is ***periodic***. More specifically,[2] if $e \geq 2$ then $\boldsymbol{x}$ is ***repetitive***. Otherwise, if $e = 1$ then $\boldsymbol{x}$ is ***primitive***. For example, the periodic string $\boldsymbol{x} = koko$ is repetitive since we have the form $\boldsymbol{x} = u^2 = (ko)^2$ where $e = 2$, whereas $\boldsymbol{u} = ko$ is primitive.

A ***run*** $\boldsymbol{r} = (i, j, p)$ in $\boldsymbol{x}$ is a *repetitive* substring $\boldsymbol{x}[i..j]$ of ***period*** $p = |\boldsymbol{b}|$, where the ***base*** $\boldsymbol{b}$ is a primitive ***generator*** of the *tandem repeat*. In particular:

- $\boldsymbol{r}$ is of the form $\boldsymbol{b}^e\boldsymbol{b}'$ where $\boldsymbol{b}'$ is a (possibly empty) proper prefix of $\boldsymbol{b}$;
- *base* $\boldsymbol{b}$ is primitive;
- *period* $p = |\boldsymbol{b}|$;
- *exponent* $e = \lfloor \frac{j-i+1}{p} \rfloor \geq 2$;
- and i, j are the furthest positions the run can extend left/right.
  (i.e. largest possible e and longest possible $\boldsymbol{b}'$)

For example, $\boldsymbol{x} = kokokokkkoko$ has the run $\boldsymbol{r}_1 = kokokok = (ko)^{3.5} = (ko)^3k = \boldsymbol{b}^3\boldsymbol{b}'$ where $\boldsymbol{b}'$ is the proper prefix *'k'*. This run is represented as $\boldsymbol{r} = (1, 7, 2)$, where the run occurs at $\boldsymbol{x}[1..7]$ and has a period of 2. Notice that position 7 is included, which extends the run to the right to have the longest possible $\boldsymbol{b}'$.

An ***eligible run*** of $\boldsymbol{u}$ in $\boldsymbol{x}$ is a run $\boldsymbol{r} = \boldsymbol{x}[i..j] = (i, j, p)$ where $\boldsymbol{u}$ occurs at least twice inside $\boldsymbol{r}$ (i.e. $f_{r,u} \geq 2$) and the base $\boldsymbol{b}$ is shorter than $\boldsymbol{u}$ (i.e. period $p < |\boldsymbol{u}|$). For example, all the runs of $\boldsymbol{x} = kokokokkkoko$ are: *(See Figure 2.1)*

- $\boldsymbol{r}_1 = (1, 7, 2) = (ko)^3k$;
- $\boldsymbol{r}_2 = (5, 10, 3) = (kok)^2$;
- $\boldsymbol{r}_3 = (7, 8, 1) = k^2$;
- and $\boldsymbol{r}_4 = (8, 11, 2) = (ko)^2$.

[1] The term *'NE repeat'* is also commonly known as ***'maximal repeat'*** in literature [14]. For the sake of clarity, we use *non-extendible repeat (NE repeat)*, where this term is used in [6].

[2] If $1 < e < 2$ then $\boldsymbol{x}$ is ***weakly periodic***.

index :	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{x}$:	k	o	k	o	k	o	k	k	o	k	o
$\boldsymbol{r}_1$:	k	o	k	o	k	o	k	–	–	–	–
$\boldsymbol{r}_2$:	–	–	–	–	k	o	k	k	o	k	–
$\boldsymbol{r}_3$:	–	–	–	–	–	–	k	k	–	–	–
$\boldsymbol{r}_4$:	–	–	–	–	–	–	–	k	o	k	o

FIGURE 2.1: All runs $\boldsymbol{r}_z$ of $\boldsymbol{x} = kokokokkoko$.

The eligible run for $\boldsymbol{u} = kok$ is only $\boldsymbol{r}_1 = \boldsymbol{x}[1..7] = (ko)^3k$ where $\boldsymbol{u}$ occurs thrice in $\boldsymbol{r}_1$ at $\boldsymbol{x}[1..3]$, $\boldsymbol{x}[3..5]$, and $\boldsymbol{x}[5..7]$. Notice that $\boldsymbol{r}_2$ is not an eligible run for $\boldsymbol{u}$ since the base is not shorter than $\boldsymbol{u}$ (i.e. $p_{r_2} \nleq |\boldsymbol{u}| = 3$).

We can count the number of positions a substring ***covers*** *(verb)* in $\boldsymbol{x}$. A string $\boldsymbol{x}$ is ***quasiperiodic*** if it is entirely *covered* by a repeat. Moreover, a ***cover*** *(noun)* of $\boldsymbol{x}$ is a proper substring $\boldsymbol{u}$ such that $\boldsymbol{x}$ can be entirely constructed from (possibly overlapping) instances of $\boldsymbol{u}$. For example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$.

In 1990, Apostolico and Ehrenfeucht [8] introduced the concept of *covers* though the term ***'quasiperiodicity'*** *(See Section 2.1 Definitions)*. Since exact covers are uncommon in practice, most recent research focuses on generalized definitions of covers *(See [13]*. One such generalization relaxes the requirement for the *entire* string $\boldsymbol{x}$ to be covered by $\boldsymbol{u}$, to only a (pre-determined) minimum number of positions, called a *'partial cover'*. In this thesis we are interested in *maximal covers*.

2.2 Arrays

2.2.1 Suffix Array $\mathcal{SA}$

The ***suffix array*** ($\mathcal{SA}[1..n]$) of $\boldsymbol{x}$ is an integer array, where $\mathcal{SA}[i]$ is the starting position of the ith ***lexicographical*** (*alphabetical*) suffix in $\boldsymbol{x}$, i.e. $\boldsymbol{x}[SA[i]..n]$.

For example with $\boldsymbol{x} = tartarus$, we sort all suffixes, in lexicographical order:

(0) ϵ[3] (1) '*artarus*', (2) '*arus*', (3) '*s*', (4) '*tartarus*', (5) '*tarus*', and (6) '*us*'.

Then, we construct an array where the index i represents the starting position in $\boldsymbol{x}$ of the suffix we are referring to, i.e. $\mathcal{SA}[i]$ refers to $\boldsymbol{x}[i..n]$, and the value represents the ith rank in the lexicographical ordering. *(See Figure 2.2)*.

2.2.2 Longest Common Prefix $\mathcal{LCP}$ Array

Alongside introducing the suffix array, in 1993, Manber and Myers [15] also introduced the *longest common prefix array* ($\mathcal{LCP}$).

The ***Longest Common Prefix*** ($\mathcal{LCP}[1..n]$) array of $\boldsymbol{x}$ is an integer array, starting with $\mathcal{LCP}[1] = 0$, where $\forall i \in [2..n]$, $\mathcal{LCP}[i]$ is the length for the longest common prefix between each pair of lexigraphically ordered suffixes referred to by $\mathcal{SA}[i-1]$ and $\mathcal{SA}[i]$ (i.e. $\boldsymbol{x}[\mathcal{SA}[i-1]..n]$ and $\boldsymbol{x}[\mathcal{SA}[i]..n]$).

For example, with $\boldsymbol{x} = tartarus$, we compare each suffix in lexicographical order for their common prefixes and store the length of the longest one. Therefore, between $\boldsymbol{x}[\mathcal{SA}[6]..8] = \boldsymbol{x}[1..8] = tartarus$ and $\boldsymbol{x}[\mathcal{SA}[7]..8] = \boldsymbol{x}[4..8] = tarus$ the longest common prefix is just *'tar'*, so the length stored is $|'tar'| = 3$, i.e. $\mathcal{LCP}[7] = 3$. *(See Figure 2.2)*

2.2.3 Substring v_i

In this thesis, we often refer to the particular substring referenced by $\mathcal{SA}[i]$ and $\mathcal{LCP}[i]$. For the sake of brevity, we shall refer to this specific substring as:

$$\boldsymbol{v}_i = \boldsymbol{x}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$$

For example, in $\boldsymbol{x} = tartarus$, then:

$$\begin{aligned}\boldsymbol{v}_2 &= \boldsymbol{x}[\mathcal{SA}[2]..\mathcal{SA}[2] + \mathcal{LCP}[2] - 1] \\ &= \boldsymbol{x}[5..5 + 2 - 1] \\ &= \boldsymbol{x}[5..6] \\ &= ar\end{aligned}$$

[3]The empty string ϵ is not necessary to the computation of $\mathcal{SA}$ since. Unlike suffix trees, which use a ***sentinel*** – end-of-string symbol, typically denoted as '$'.

index :	1	2	3	4	5	6	7	8
$\boldsymbol{x}$:	t	a	r	t	a	r	u	s
$\mathcal{SA}$:	2	5	3	6	8	1	4	7
$\mathcal{LCP}$:	0	2	0	1	0	0	3	0
$\mathcal{RSF}$:	0	2	0	2	0	0	2	0

Figure 2.2: $\mathcal{SA}$, $\mathcal{LCP}$, and $\mathcal{RSF}$ Array of $\boldsymbol{x} = tartarus$.

index :	1	2	3	4	5	6	7	8
$\boldsymbol{x}$:	t	a	r	t	a	r	u	s
β :	0	0	0	1	2	3	0	0

Figure 2.3: Border Array of $\boldsymbol{x} = tartarus$.

2.2.4 Border Array

The ***Border Array*** $\beta[1..n]$ of $\boldsymbol{x}$ is an integer array where $\beta[i]$ is the length for the longest border of the prefix $\boldsymbol{x}[1..i]$. Also, since a border requires a proper prefix/suffix then $\boldsymbol{x}[1]$ cannot have a border, thus $\beta[1] = 0$. For example, with $\boldsymbol{x} = tartarus$, we see $\beta[4..6] = \{1, 2, 3\}$ increasing to reflect the borders in the prefixes: *'tart'*, *'tarta'*, and *'tartar'*. *(See Figure 2.3)*. An algorithm to compute $\beta[1..n]$ in $\Theta(n)$ time can be found in [6].

2.2.5 Repeating Substring Frequency $\mathcal{RSF}$ Array

The ***Repeating Substring Frequency*** ($\mathcal{RSF}[1..n]$) array of $\boldsymbol{x}$ [12] is an integer array that computes the frequencies of all NRE repeats, where $\mathcal{RSF}[i]$ is the frequency of the substring $\boldsymbol{v}_i$ referred to by $\mathcal{SA}[i]$ and $\mathcal{LCP}[i]$. Note when $\mathcal{LCP}[i] = 0$, which are empty strings, then $\mathcal{RSF}[i] = 0$. For example, in $\boldsymbol{x} = tartarus$, if we consider index $i = 2$, we see the substring $\boldsymbol{v}_2 = ar$ occurs twice ($f_{x,u} = 2$) at $\boldsymbol{x}[2..3]$ and $\boldsymbol{x}[5..6]$, so $\mathcal{RSF}[2] = 2$. *(See Figure 2.2)*

index :	1	2	3	4	5	6	7	8	9
$\boldsymbol{x}$:	k	o	k	o	k	o	k	k	o
$\mathcal{SA}$:	7	8	5	3	1	9	6	4	2
$\mathcal{LCP}$:	0	1	2	3	5	0	1	2	4
$\mathcal{RSF}$:	0	5	4	3	2	0	4	3	2
$\mathcal{OLP}$:	0	0	0	2	3	0	0	0	2
$\mathcal{RSPC}$:	0	5	8	7	7	0	4	6	6

Figure 2.4: $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, $\mathcal{OLP}$, and $\mathcal{RSPC}$ arrays of $\boldsymbol{x}$ = *kokokokko*.

2.2.6 Overlapping Positions $\mathcal{OLP}$ Array

The ***Overlapping Positions*** ($\mathcal{OLP}[1..n]$) array of $\boldsymbol{x}$ is an integer array where $\mathcal{OLP}[i]$ is the total number of overlapping positions between all instances of $\boldsymbol{v}_i$ in $\boldsymbol{x}$ referred to by $\mathcal{SA}[i]$ and $\mathcal{LCP}[i]$. Note when $\mathcal{LCP}[i] = \{0, 1\}$ (an empty string or a single letter) or $\mathcal{RSF}[i] = 0$ (v_i does not occur in $\boldsymbol{x}$) then $\mathcal{OLP}[i] = 0$.

For example, with $\boldsymbol{x} = kokokokko$: *(See Figure 2.4)*

- At index $i = 5$, $v_5 = kokok$ occurs twice ($\mathcal{RSF}[5] = 2$) at $\boldsymbol{u}_1 = \boldsymbol{x}[1..5]$ and $\boldsymbol{u}_2 = \boldsymbol{x}[3..7]$;
- Instances of v_5 overlap at $\boldsymbol{x}[3..5]$ with 3 overlapping positions ($\omega_{u_1,u_2} = 3$);
- Therefore $\mathcal{OLP}[5] = 3$.

Whereas at index $i = 3$, $v_3 = ko$ cannot overlap because *'ko'* does not have a *border*.

We can improve $\mathcal{OLP}$ to $\mathcal{OLP}^*$ by setting $\mathcal{OLP}^*[i] = 0$ when an immediate duplicate occurs ($\mathcal{OLP}[i] = \mathcal{OLP}[i+1]$) since index i is referring to the same $\boldsymbol{v}_i$.

2.2.7 Repeating Substring Positions Covered $\mathcal{RSPC}$ Array

The ***Repeating Substring Positions Covered*** ($\mathcal{RSPC}[1..n]$) array of $\boldsymbol{x}$ is an integer array where $\mathcal{RSPC}[i]$ is the number of positions covered by $\boldsymbol{v}_i$ referred to by $\mathcal{SA}[i]$ and $\mathcal{LCP}[i]$. The values are simply computed with the formula: $\mathcal{RSPC}[i] = \mathcal{RSF}^*[i] * \mathcal{LCP}[i] - \mathcal{OLP}^*[i]$. Traversing the array for the maximum value will provide the information needed to identify the maximal cover $\boldsymbol{\mu}$, which is $\boldsymbol{v}_i$.

For example, $\boldsymbol{x} = kokokokko$, the substring $\boldsymbol{v}_9 = \boldsymbol{x}[2..6] = okok$ has a length of 4 ($\mathcal{LCP}[9] = 4$), occurs twice ($\mathcal{RSF}[9] = 2$) and overlaps 2 positions ($\mathcal{OLP}[9] = 2$). So, the total positions covered by $\boldsymbol{v}_9$ is $\mathcal{RSPC}[9] = 2 * 4 - 2 = 6$. The maximum value is $\mathcal{RSPC}[3] = \mathcal{RSPC}[4] = 8$ (which is the same substring $\boldsymbol{v}_3 = \boldsymbol{v}_4$), so the maximal cover is *'ko'*. *(See Figure 2.4)*

2.3 Maximal Covers

A ***maximal cover***[4] of $\boldsymbol{x}$ is the shortest or longest repeat $\boldsymbol{\mu}$ that *covers* the maximum number of positions in $\boldsymbol{x}$. Note that you can have multiple maximal covers if they are all tied for the maximum number of positions covered in $\boldsymbol{x}$. For example, $\boldsymbol{x} = tartarus$ has the maximal cover $\boldsymbol{\mu} = tar$, which covers 6 positions in $\boldsymbol{x}$.

In 2015, Kociumaka et al. [9] introduced ***α-partial covers***, where at least α positions are covered. This paper presents the *'AllPartialCovers'* problem, described as follows:

- Let $\boldsymbol{u}$ be a cover in $\boldsymbol{x}[1..n]$;
- $\textsc{Covered}(\mathbf{u}, \mathbf{x})$ denotes the number of positions in $\boldsymbol{x}$ covered by $\boldsymbol{u}$;
- $\forall\alpha = \{1, ..., n\}$ return the shortest $\boldsymbol{u}$ such that $\textsc{Covered}(\mathbf{u}, \mathbf{x}) \geq \alpha$.

The AllPartialCovers problem solves computing the *shortest* maximal cover(s) $\boldsymbol{\mu}$ (rather than *longest*) that covers at least α positions, for the largest possible α (i.e. maximum number of positions).

The solution presented by [9] to *'AllPartialCovers'* uses *suffix trees*, which are not ideal for computing long strings. Mhaskar and Smyth [1] remedy this concern using *suffix arrays* instead.

In 2018, Mhaskar and Smyth [1] introduced the concept of *maximal covers* as ***optimal covers***, and two new algorithms to compute them (as described below). Through this research, multiple new data structures were presented.

Assuming $\mathcal{SA}$, $\mathcal{LCP}$, and $\mathcal{RSF}$ are pre-computed in linear-time, [1] presented algorithms that can compute $\mathcal{OLP}$ in quadratic $\mathcal{O}(n^2)$ time and logarithmic $\mathcal{O}(n \log n)$ time. To improve the time to compute, they proposed $\mathcal{RSF}^*$ and $\mathcal{OLP}^*$, where the tandem duplicate values refer to the same $\boldsymbol{v}_i$ (determined through $\mathcal{SA}$ and $\mathcal{LCP}$) are set to zero.

The algorithms to compute the $\mathcal{OLP}^*$ array depend on the following concept:

> "Suppose $\boldsymbol{u}$ is a substring of $\boldsymbol{x}$, and $\boldsymbol{r}$ is an eligible run of $\boldsymbol{u}$.
> Then any two consecutive occurrences of $\boldsymbol{u}$ in $\boldsymbol{r}$ must overlap." [1]

We can see this demonstrated in Figure 2.5, where $\boldsymbol{u}$ occurs at least twice in $\boldsymbol{r}$ and $|u| > p = |b|$, thus $\boldsymbol{r}$ is an eligible run of $\boldsymbol{u}$. In the figure, we define the base $\boldsymbol{b} = \boldsymbol{b}'\boldsymbol{b}''\boldsymbol{b}'''$, and the substring $\boldsymbol{u} = \boldsymbol{u}'\boldsymbol{u}'' = \boldsymbol{b}''\boldsymbol{b}'''\boldsymbol{b}'\boldsymbol{b}''$. In general, $\boldsymbol{u}$ must share a common proper prefix and proper suffix (i.e. a *border*) $\boldsymbol{b}''$. *(See [1] for the proof)*

[4]This term was introduced in [1] as *'optimal cover'*, but updated to *maximal cover* in [7] for the sake of clarity.

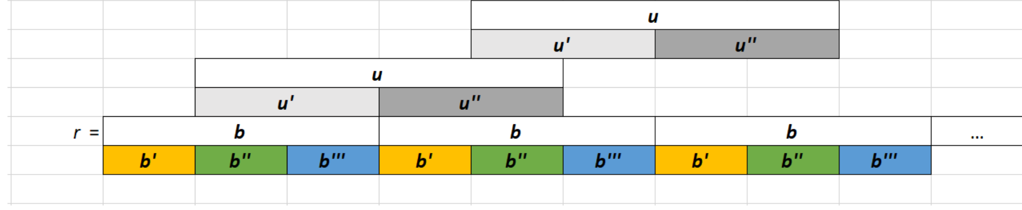

FIGURE 2.5: An eligible run $\boldsymbol{r}$ of $\boldsymbol{u}$ where $\boldsymbol{u} = \boldsymbol{u}'\boldsymbol{u}'' = \boldsymbol{b}''\boldsymbol{b}'''\boldsymbol{b}'\boldsymbol{b}''$.

We can determine the eligible run $\boldsymbol{r}$ of $\boldsymbol{u}$ by using the EXRUN(i', j')[5] function by providing $\boldsymbol{x}[i'..j']$ where $\boldsymbol{u}$ appears at least twice *(See Section 4.2)*, and ensuring $|u| > p$. We can determine the range (i', j') that $\boldsymbol{u}$ occurs consecutively using the $\mathcal{SA}$ values that refer to the same $\boldsymbol{v}_i$. *(See Section 4.2.1 on R_1 & R_m arrays.)*

We leave an explanation of the two algorithms to compute the $\mathcal{OLP}$ array to Section 4, *Implementation.*

In order to count the number of positions a substring covers, the *Repeating Substring Positions Covered* ($\mathcal{RSPC}$) array was created. With a pre-computed $\mathcal{OLP}^*$ array, [1] presented a linear-time algorithm to compute the $\mathcal{RSPC}$ array. By traversing the $\mathcal{RSPC}$ array for the maximal value, [1] presented a linear-time algorithm to compute the list of maximal covers $\boldsymbol{\mu}$ for a given $\boldsymbol{x}$.

[5]EXRUN(i', j') returns the run in $\boldsymbol{x}[i'..j']$ with $min(p)$.

2.4 Data Analytics

The content in this section contains information published in the research paper [7].

> "Data Analytics is defined as the application of computer systems to the analysis of large data sets for the support of decisions." [16]

A common form of "large datasets" in modern computer processing are strings, e.g.:

- binary integers $\{0, 1\}$;
- genomic nucleotides $\{a, c, g, t\}$;
- upper/lower case letters of the English alphabet.

Such strings can contain billions or trillions of symbols so efficient processing methods (i.e., data analytics) are of prime importance in extracting meaning from them.

The stages of data analytics have been defined in various ways, e.g. KDD (1996) [17], CRISP-DM (2000) [18], or ASUM (2016) [19]. Here, for clarity, we use Runkler's breakdown into four stages [16]:

(1) Preparation *(data is assessed and selected)*

(2) Preprocessing *(data is cleaned and filtered)*

(3) Analysis *(data is visualized and analyzed)*

(4) Post-processing *(data is interpreted and evaluated)*

Since string processing methods are particularly adapted to scanning, filtering and integrating massive data sets [20], the primary focus of this paper is on their application to the second and third of these stages: preprocessing and analysis.

During Preprocessing, to extract meaning from large disorganized raw data sets, the input is first cleaned and filtered by removing outliers and noise. Cleaning can be performed by identifying repeating keywords or phrases. The cleaned data can then be compressed using various data compression techniques, which is also a major aspect of stringology over the last 50 years. As a result, large datasets may be handled using significantly less space. Thus, they are more amenable to further processing using various string algorithms and corresponding data structures. Once cleaned and filtered, the data may be tabulated and formatted for further processing.

In the Analysis phase, the data is first compartmentalized or modelled using a variety of data mining algorithms. The resulting datasets may then be used for prediction, classification, or other forms of analysis. Many algorithms for pattern matching or computing repeats in strings may be suitable for this analysis, e.g., classification or clustering based on common patterns in the data.

Chapter 3

Applications to Data Analytics

The methods of data analytics, in particular data mining and clustering, have many application areas; for instance, bioinformatics, information security, image analysis, machine learning, event and time series analysis, or human-computer interaction.

The content in this chapter contains information published in the research paper [7]:

Title: *'An Overview of String Processing Applications to Data Analytics'*
Authors: Holly Koponen, Neerja Mhaskar and W. F. Smyth
Conference: 2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)
Publisher: IEEE
Year: 2021
Pages: 1-8
doi: 10.1109/RDAAPS48126.2021.9452004

3.1 Bioinformatics

Bioinformatics is an interdisciplinary area that analyzes biological data using the methods of computer science, mathematics and statistics. Usually, this data consists of very long biological sequences — genomes, DNA sequences, protein sequences — which may contain billions or trillions of characters. In order to process these strings efficiently, specialized and sophisticated algorithms have been developed to prepare, preprocess, and analyze the data. This data is not only stored as a sequence of characters but may also be filed in *Biological Data Bases (BDBs)* where the data is structured and interconnected using links that identify relationships among them. Therefore, many data analytics techniques such as data mining and clustering play a key role.

Analyzing these biological sequences is crucial in treating many deadly diseases such as AIDS and other forms of cancer. Perhaps even more important in the long run, this analysis yields insights into the genetic composition of living beings so that such diseases may be prevented entirely by the development of new vaccines.

As noted above, strings and their associated algorithms naturally relate to biological sequences. Thus it is not surprising that these methodologies arise in many of the

strategies adopted by data analytics to handle the big data sets that arise in bioinformatics. Here we outline some of these strategies for processing and analyzing biological sequences.

In [21], Gesú discusses clustering and data mining as strategies of data analytics for the analysis of bio-data. In [22] the authors present an overview of data analytics applications to the big data of bioinformatics. Yin et. al. in [23] specifically discuss various computing platforms for data analytics of big biological data and the challenges that these methods provide. In [24], Halzelhurt et. al. propose a new algorithm based on suffix array computation for clustering data, thus a direct application of stringology. All the problems discussed in these papers are stringological and handled by direct application of string algorithms.

In particular, repetitions in strings are a natural focus of bioinformatics. In [25], Haubold et. al. discuss the repetitive nature of genomes. Following up on this, [26] estimates that over 66% of the human genome consists of repetitive sequences. As discussed in [27], repetitive DNA results from random crossover events during cell division characterized by insertion, deletion or duplication in the chromosome. This kind of repetition is important because it is associated with several genetic diseases [27]. Thus biologists have a natural interest in repetitions in the genome and require efficient methods for recognizing and specifying them.

This focus on repetitive sequences is reflected in many other papers on bioinformatics. Repetitive DNA varies according to size, frequency, and genomic location of the repetition [28]. Based on these criteria, repetitive DNA sequences can be divided into two classes depending on whether or not the repeat is tandem [29].

In [30], Pickett et. al. propose a suffix array based algorithm for exhaustive and efficient searching of "simple sequence repeats" (SSR) in large genetic sequences.

Pattern matching algorithms (both exact and approximate) are common in sequence alignment [31, 32], identifying diseases [33] and genetic mutations and disorders [29]. The use of suffix arrays greatly reduces the space required for many biological applications originally based on suffix trees. However, these indexing data structures do not aid in compressing the string. For this, run-length encoding[1], an efficient lossless compression technique, works well for strings with many runs. On the other hand, for biological sequences with few runs but many repeats, one approach to their compression is to alter the repeats to form runs and represent them using run-length encoding. One drawback to this approach is that restoring the altered strings to their original form consumes excessive time and space.

Burrows-Wheeler Transform [34] ($BWT_{\boldsymbol{x}}$) is a permutation of $\boldsymbol{x}$ formed by the last column of the ***Burrows-Wheeler Matrix (BWM)***, which is constructed by writing down all the rotations of $\boldsymbol{x}\$$, where \$ is a special letter smaller than every letter in Σ, then arranging them in lexicographical order (see Figure 3.1). It turns out that

[1]Run-length encoding allows runs of data (sequences in which the same data value occurs in many consecutive data elements) to be stored as a single data value and count, rather than as the original run.

$BWT_{\boldsymbol{x}}$ can be efficiently computed in $\Theta(n)$ time, thus the computation of BWM can be bypassed. An advantage of BWT is that it contains many runs and is thus highly compressible. The following equation shows the interesting connection between BWT and $\mathcal{SA}$:

$$BWT(\boldsymbol{x}) = \begin{cases} \boldsymbol{x}[\mathcal{SA}[i]-1], & \text{if } \mathcal{SA}[i] > 1 \\ \$, & \text{if } SA[i] = 1 \end{cases}$$

$$\begin{array}{ccccccc}
\$ & a & b & a & a & b & \mathbf{a} \\
a & \$ & a & b & a & a & \mathbf{b} \\
a & a & b & a & \$ & a & \mathbf{b} \\
a & b & a & \$ & a & b & \mathbf{a} \\
a & b & a & a & b & a & \mathbf{\$} \\
b & a & \$ & a & b & a & \mathbf{a} \\
b & a & a & b & a & \$ & \mathbf{a}
\end{array}$$

FIGURE 3.1: BWM for $\boldsymbol{x} = abaaba\$$, where $BWT_{abaaba\$} = abba\$aa$ is the last column in bold.

The power of BWT has only in the last dozen years or so been recognized. This is primarily due to efficient data structures such as the FM-index [35], which further compresses the BWT for space efficiency. Also, BTW is easy to work with, thus significantly improving the efficiency of the applications using BWT. Furthermore, inverting the BWT to obtain the original string can efficiently be done using a backward-search technique. Hence, BWT has become the data structure of choice in many applications in bioinformatics [36, 37, 38]. The well-known mappers — Burrows-Wheeler-Aligner [36], Bowtie2 [39] and HISAT2 [40] — all use BWT for sequence alignment.

3.2 Information Security

According to the International Organization for Standardization (ISO/IEC 27002:2005 [41]), information security is the protection of information from any threat involving unauthorized collection of confidential data. Information security is achieved through policies, processes, and procedures that we survey below.

To develop these policies, current procedures must be assessed for vulnerabilities. This process may involve gathering data on current procedures and analyzing the results for any susceptibilities to breaches in the system. Here we focus on information security procedures that involve strings. These procedures include many forms of encryption including ciphertext and their cryptographic keys as well as intrusion detection systems based on digital signatures.

3.2.1 Repeats in Ciphertext

Cryptography seeks to provide secure communication in the presence of third parties by encoding information to make it unintelligible to them. Thus encryption transforms plaintext, the original message, into ciphertext.

Substitution ciphers are a type of encryption scheme in which each plaintext substring is replaced with an associated ciphertext substring. Ciphers that use substitution of plaintext substrings longer than a single character are called ***block ciphers***. Ciphers based on single-character substitution are susceptible to brute-force attacks, such as a ***dictionary attack***.

In a dictionary attack, an exhaustive list or *'dictionary'* containing all possible substitutions is systematically tried until a mapping that provides access to the information is found [42, 43]. A ***repetition pattern attack*** is a type of dictionary attack where testing focuses on the most frequently occurring substrings.

According to Bauer [44], the first *Invariance Theorem* states that a ***monoalphabetic***[2] substitution cipher with a repeat in the plaintext will also have a corresponding repeat in the ciphertext. ***Idiomorphs*** are strings with the same pattern in the location, length, and frequency of a repeat [44]. There is a theoretical maximum number of possible idiomorphs for a given length n of a string, represented by Bell's Numbers [44]. Thus, by using a dictionary of idiomorphs, an attacker can mount a repetition pattern attack based on a given repeat in the ciphertext to infer the associated plaintext character(s).

In terms of data analytics, given a ciphertext, preprocessing involves filtering the ciphertext for repeats. Nevertheless, of course, the same algorithms, for example, those of Gusfield [14], or Abouelhoda et al. [45], may also be used by an attacker.

[2]**monoalphabetic**: a substitution cipher in which each plaintext character has a single corresponding ciphertext equivalent.

Once repeats are found, analysis can be performed using Kasiski's Method [46] to determine the length of a keyword in a ***polyalphabetic***[3] substitution cipher. Using the length of a keyword with a repetition found through Kasiski's Method and Bell's Numbers, which determine the maximum possible idiomorphs for a given length, a systematic check is performed based on the dictionary of all possible idiomorphs. A shortened list of possible idiomorphs can now be provided based on context, grammar, and usage frequency.

A data analyst can now post-process using the shortened list of possible plaintext idiomorphs to suggest the most probable resulting plaintext.

Searching for repeats in the ciphertext is exemplified by the algorithm of Arnold et al. [47] that identifies patterns in encrypted data. Using data compression methods, this algorithm applies an invariance transformation to both the ciphertext and the chosen pattern(s). Then, standard string searching techniques are used to detect the pattern in the reduced forms and thus potentially, depending on the encryption method, corroborate matches.

3.2.2 Intrusion Detection System: Signature-based

An ***intrusion detection system*** (IDS) is an application used to monitor network traffic for malicious activity or policy violations. There are eleven detection methods for processing IDS input data [41]. The most common of these are signature-based and anomaly-based. An ***anomaly-based*** IDS builds a model of the appropriate behaviour from the protected system, then flags deviations from the system. More interesting perhaps is the ***signature-based*** IDS, which involves looking at strings found in network packets to identify signatures of known threats.

Network packets are units of data containing headers, metadata and the 'payload,' which is the actual intended message. Nearly all network traffic encryption protocols use a unique specific packet format with distinct packet payloads [48].

To perform analysis on signature-based IDS systems, a collection of strings within the packet payload is used as the dataset, e.g. the KDDCUP99 IDS dataset created from a network packet analyzer [49]. After preprocessing, string algorithms for regular expression matching can be used to identify these specific protocol patterns within the payload [48].

Packet content inspection and filtering depend on a fast multi-pattern matching algorithm since they constitute the central algorithmic step in a signature-based IDS. Suresh et al. [50] propose the All-Ready State traversal pattern matching algorithm, a hardware-based approach that provides efficient memory usage. The algorithm constructs a state traversal machine, which enables users to search and store large string patterns in a database by following a path vector.

[3] **polyalphabetic**: of a substitution cipher in which each plaintext character may use a different alphabet to produce a corresponding ciphertext equivalent.

3.3 Image Analysis & Video Processing

3.3.1 Image Analysis

Image analysis is a subfield of data analytics in which a collection of images forms the dataset from which meaningful information is extracted. For example, information such as colours, shapes, objects, people, places, or patterns may be extracted from an image using data analysis techniques.

Images may be converted into a string where each row of pixels is concatenated to the previous row and terminated by an end-of-file marker. Each symbol represents the colour and location of each pixel.

Preprocessing the image includes image enhancement and restoration, where removing variations such as noise or illumination can increase the processing speed for analysis. This process is essentially a pattern matching problem, particularly an application of BWT. For example, Gadde [51] implements this process to compress iris images for analysis. Similarly, Prandhan et al. [52] propose an image compression technique using BWT in order to achieve better compression efficiency than the traditional JPEG approach.

Once images are cleaned, classification of the subject in the image may be performed using string matching methods. This is shown by Chen and Gao in human facial recognition [53]. Attribute strings measure the similarity of shapes where edge pixels are considered "letters" grouped into straight lines with local structural information as "words". Collections of these words are then formed into "sentence" strings, which represent an edge curve on a shape when concatenated together. In human facial recognition, this shape edge is the curve on a face.

Chen and Gao go on to design an Ensemble String Matching scheme (ESM) to handle uncertainty problems relating to each string's direction and the measurement of the magnitude of dissimilarity between two "Stringfaces". Unlike many string matching methods that measure a single matching score, this ESM scheme is based on the aggregate of similarities among all matched pairs. Once the strings are scored based on their similarities, the strings can be categorized into clusters to determine whether or not the faces can be recognized.

3.3.2 Video Recognition

Videos are processed as a series of images, known as 'video stills,' in which the relationship between adjacent images can identify important features. For example, video recognition can identify the subject matter using a series of images as the data source. Moreover, video recognition can identify the type of motion involved for each subject, such as walking, running, jumping, or other actions.

These results are achieved by translating the sequence of images into a string of symbols to represent each event, where each event is the supposed action taken by the

subject. This method is implemented by Hsieh et al. [54] on human movement analysis videos such as surveillance systems where each pose is recognized using the skeleton of the subject and associated with a symbol. Then key postures are selected and concatenated into a series of poses represented by a string. Such strings are delineated by start- and end-of-sequence markers. For example, walking could be represented by *'swwwe'*, where s is the start, w is walking, and e is the end. Similarly, picking up an object could be represented by *'swwppwwe'*. or falling by *'swwwwffe'*, where p is picking up an object and f is falling.

Once a string of events is given, two movement sequences can be compared. The amount of dissimilarity between the sequences is measured by edit distance – a fundamental function of stringology. Depending on the edit distance, the string of events may be categorized into actions performed by the subject in that video segment.

Ghasemzadeh et al. [55] also implemented a similar method for structural action recognition in body sensor networks. In their work, a model template was generated to label each primitive with a unique symbol and a sequence of symbols representing a particular action. Then, again using edit distance, an action recognition computation was used as the basis for classifying the sequence of symbols.

Chapter 4

MAXCOVER

This chapter describes the MAXCOVER software developed to compute maximal covers. The extension to compute NE repeats is covered in Chapter 5, in Section 5.2.2.

Availability: Open source code, binaries, and test data are available on Github at https://github.com/hollykoponen/MAXCOVER. Currently runs on Linux, untested on other OS.

Note that the pseudocode and algorithms rely on counting $[1..n]$. However, during implementation we considered that arrays count $[0..n-1]$ and made adjustments to the algorithms accordingly. This is most significant when using $\mathcal{SA}$, since this array refers to start positions in $\boldsymbol{x}$, and must be decremented as a consequence.

4.1 Software Architecture

The software for MAXCOVER is based on the pseudocode provided in [1], written in C++ using standard libraries. Figure 4.1 shows a simplified diagram for the software architecture of MAXCOVER.

The MAIN() function receives an input of two flags that indicates whether to compute *maximal covers* or *NE repeats*. In addition, MAIN() also takes in two additional flags to indicate the following parameters:

(1) Whether or not to output the ten longest covers
(2) Minimum length for the NE repeat

The COMPUTE_MC() function begins a sequence of procedures that compute the following arrays: $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, R_1 & R_m, $\mathcal{OLP}$, $\mathcal{RSPC}$, and MC_List.

The first three arrays, $\mathcal{SA}$, $\mathcal{LCP}$, and $\mathcal{RSF}$, make use of code by:

- $\mathcal{SA}$: SA-IS algorithm [56] implemented by Yuta Mori[1]
- $\mathcal{LCP}$: Implementation by Simon J. Puglisi and Andrew Turpin [4]
- $\mathcal{RSF}$: Implementation by Neerja Mhaskar and W.F. Smyth [12]

[1] https://sites.google.com/site/yuta256/

We selected the above implementations since they are, to our knowledge, the most efficient linear time array constructions for each respective data structure. In addition, each of these implementations is modular C/C++ code and can be substituted for any potentially more efficient implementations. We generated a Makefile using Premake5 to facilitate this modularity.

In 2011, Nong et al. [56] proposed the SA-IS algorithm, an efficient algorithm for linear-time suffix array construction. Yuta Mori implemented and optimized the SA-IS algorithm, which we used in the MAXCOVER software. We used the implementation by Puglisi and Turpin to compute $\mathcal{LCP}$, which is computed in linear time and space, and the RMQ data structure in the MAXCOVER software. In 2018, Mhaskar and Smyth [12] presented an algorithm to compute the $\mathcal{RSF}$ array in linear time.

Then, we implemented the code to compute the $\mathcal{OLP}$ array based on the pseudocode in [1]. This code relied on data structures not provided in [1], namely: R_1 & R_m arrays, a hashtable of runs RUNSHT, and the EXRUN() function. *(See Section 4.2)*.

In addition to the above, computing RUNSHT requires the following data structures and procedures that we implemented: $\mathcal{LCS}$, $\mathcal{RANK}$[2], $\mathcal{REV_RANK}$, arrays and COMPUTE_RUNS() procedure. *(See Section 4.2.2)*

The code to compute the $\mathcal{RSPC}$ array and the list of maximal covers (MC_LIST) uses the pseudocode from [1]. For example, suppose the user indicated the output should include the top ten longest covers. In that case, MAXCOVER will scan the $\mathcal{LCP}$ array for the ten longest $\boldsymbol{v}_i$ and output them. Otherwise, the list of maximal covers is sent to the output.

Suppose the user indicated for MAXCOVER to compute NE repeats, calling the REPEAT_MATCHES() procedure. Then the software will perform a similar process to compute the following arrays: $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, R_1 & R_m, unlike COMPUTE_MC(). In addition to these arrays, MAXCOVER computes the *inverse arrays* for $\mathcal{ISA}^1$, $\mathcal{ILCP}$, and $\mathcal{IRSF}$. This is required because the previous code only computes NRE repeats. By taking the inverse, we can compute the NLE repeats as well, thus fulfilling the requirements for an NE repeat.

Note that the computation for NE repeats also takes into consideration the user defined “minimum repeat length”. *(See ‘α-partial covers’ in Section 2.3 and 5.2.2.)*

MAXCOVER then groups all similar NE Repeats $\boldsymbol{v}_i$ in the COMPUTE_REPORTED() procedure and outputs the length and a list of start positions that this repeat occurs at in the string. *(See Section 5.2.2 for details on this choice)*

[2]Note: $\mathcal{ISA}$ and $\mathcal{RANK}$ arrays are the exact same definition and only differ by name.

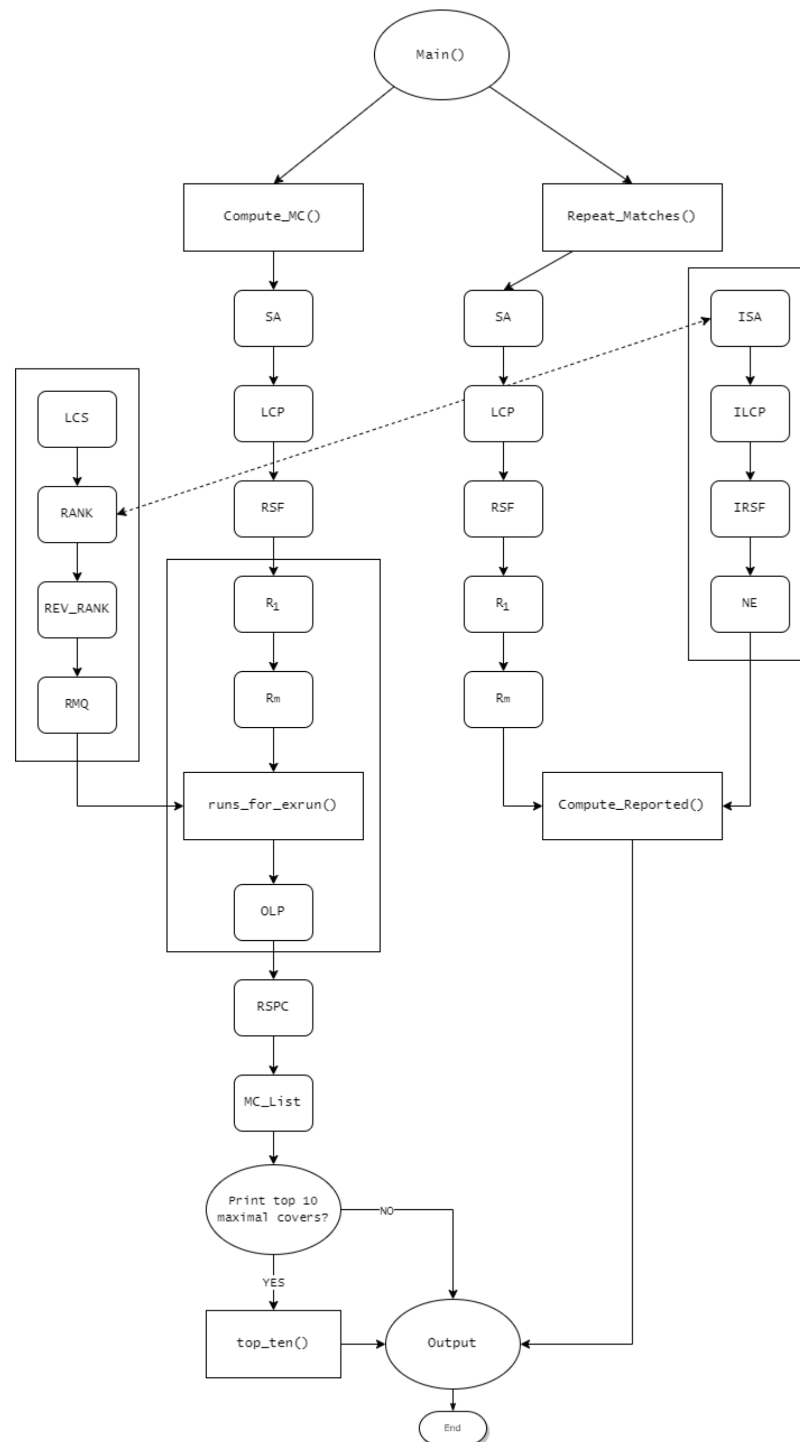

FIGURE 4.1: Simplified diagram for the software architecture of the MAXCOVER software to compute maximal covers and NE repeats.

4.2 Data Structures

Implementing the pseudocode consisted of more than transposing the algorithm, but also implementing new data structures that were not described in [1], namely:

- R_1 & R_m arrays;
- computing and storing all runs in a hashtable RUNSHT with the COMPUTE_RUNS() procedure;
- computing eligible runs with the EXRUN() function.

4.2.1 R_1 & R_m

The R_1 & R_m arrays indicate the range of $\mathcal{SA}$ values that all share the same prefix (or the substring itself) $\boldsymbol{v}_i$. In general, this means $\boldsymbol{v}_i$ occurs at starting positions $\mathcal{SA}[R_1[i]..R_m[i]]$.

For example, we return to $\boldsymbol{x} = kokokokko$ in Figure 4.2. By definition of $\mathcal{LCP}$ array, $\boldsymbol{v}_4 = kok$ is a common prefix between $\mathcal{SA}[3]$ and $\mathcal{SA}[4]$, so $R_1[4] = 3$. In addition, we see *'kok'* is also a prefix for $\boldsymbol{v}_5 = kokok$, so $R_m[4] = 5$. Therefore, *'kok'* occurs at start positions:

$$\mathcal{SA}[R_1[4]..R_m[4]] = \mathcal{SA}[3..5] = \{5, 3, 1\} \tag{4.1}$$

index :	1	2	3	4	5	6	7	8	9
$\boldsymbol{x}$:	k	o	k	o	k	o	k	k	o
$\mathcal{SA}$:	7	8	5	3	1	9	6	4	2
$\mathcal{LCP}$:	0	1	2	3	5	0	1	2	4
R_1 :	0	1	2	3	4	0	6	7	8
R_m :	0	5	5	5	5	0	9	9	9

FIGURE 4.2: R_1 & R_m Arrays of $\boldsymbol{x} = kokokokko$.

This was implemented by traversing the $\mathcal{LCP}$ array and keeping track of the rise and fall of values using a stack. *(See Appendix B.2 for the pseudocode)* In particular, if the $\mathcal{LCP}$ values increase ($\mathcal{LCP}[top] < \mathcal{LCP}[i]$), this is a new $\boldsymbol{v}_i$ to keep track of (so, we store the respective $R_1[i] = i - 1$) but shares a common prefix with the previous stack value. If the $\mathcal{LCP}$ values decrease ($\mathcal{LCP}[top] > \mathcal{LCP}[i]$), then we pop this value off the stack and store the respective R_m value. If they're equal, we do nothing. If $\mathcal{LCP}[i] = 0$, then we pop everything off the stack, storing the respective R_m values and storing a new R_1 value.

4.2.2 Runs

The r_1 and r_m values for $\boldsymbol{v}_i$ pertain to the range of $\mathcal{SA}$ values this repeat occurs at, i.e. $\mathcal{SA}[r1..rm]$. With them we can determine which occurrences of $\boldsymbol{v}_i$ occur consecutively by sorting $\mathcal{SA}[r1..rm]$ increasing order in a temporary array $\mathcal{SA}^*$. For example, $\boldsymbol{v}_4 = kok$ occurs at start positions: $\{5, 3, 1\}$. Therefore, $\mathcal{SA}^* = \{1, 3, 5\}$.

If the difference between a pair in $\mathcal{SA}^*$ is smaller than the length of the substring we're looking at (i.e. $|\mathcal{SA}^*[k]-\mathcal{SA}^*[k+1]| < \mathcal{LCP}[i] = |\boldsymbol{v}_i|$), then there exists an overlap.

Rather than simply computing $\mathcal{LCP}[i] - (\mathcal{SA}^*[k+1] - \mathcal{SA}^*[k])$ *(See Section 4.3.1 and 4.3.2)*, we take advantage of *eligible runs*. Recall that $\boldsymbol{r}$ is an ***'erun'*** of $\boldsymbol{u}$ when $f_{\boldsymbol{r},\boldsymbol{u}} \geq 2$ and $p < |u|$. This concept is useful because we traverse the same range of indices in $\mathcal{SA}$ multiple times, resulting in the quadratic $\mathcal{O}(n^2)$-time. Therefore, by using a pre-computed list of all runs in $\boldsymbol{x}$, we can reduce the runtime to logarithmic $\mathcal{O}(n \log n)$. *(See [1] for the proof)*

We compute all runs using the COMPUTE_RUNS() procedure. Then, we store all runs for $\boldsymbol{x}$ in a hashtable called RUNSHT. In C++, we use the standard library **multimap** which is an ordered hashtable so we can easily locate the list of runs with the same period. *(See EXRUN() below.)*

Based on the algorithm by Crochemore et al. [5] (DETECTRUNS(u)), we implemented the COMPUTE_RUNS() procedure to compute all runs. *(See Appendix B.5 for pseudocode.)* This procedure requires pre-computed $\mathcal{LCS}$ (*Longest Common Suffix*), $\mathcal{RANK}$, and $\mathcal{REV_RANK}$ arrays.

As the name implies, $\mathcal{LCS}$ is the longest common suffix between lexicographically ordered prefixes. This can be computed by reversing $\boldsymbol{x}$ and computing $\mathcal{SA}$ and $\mathcal{LCP}$ of $rev_{\boldsymbol{x}}$, since the opposite of a prefix for $\boldsymbol{x}$ is the suffix for $rev_{\boldsymbol{x}}$ (and vice versa).

$\mathcal{RANK}$ array ranks the suffixes in ascending order, i.e. $\mathcal{RANK}[\mathcal{SA}[i]] = i$. In fact, this is the exact same definition as the *inverse* of the $\mathcal{SA}$ array ($\mathcal{ISA}$). Thus, we say $\mathcal{REV_RANK}$ is the $\mathcal{RANK}$ array of the $rev_{\boldsymbol{x}}$.

We use $\mathcal{RANK}$ and $\mathcal{REV_RANK}$ in the LC() procedure [4], which computes the RMQ (range minimum query), to determine the minimum index for $\mathcal{LCP}$ and $\mathcal{LCS}$, respectively.

4.2.3 Exrun function

We use the EXRUN(i', j') function to compute the unique eligible run with $min(p)$ for $\boldsymbol{v}_i$ by providing the range of values where $\boldsymbol{v}_i$ appears at least twice. This range is indicated by the $R_1[i]$ and $R_m[i]$ values:

$$\boldsymbol{x}[\mathcal{SA}[r_k]..\mathcal{SA}[r_{k+1}] + \mathcal{LCP}[r_k] - 1] \tag{4.2}$$

We implemented the EXRUN(i', j') function in [1] to return a unique run with the smallest period within $\boldsymbol{x}[i'..j']$. *(See Appendix B.3.)* However, this implementation can be improved by using the algorithm proposed by Bannai et al. in [57] based on ***Lyndon words***. *(See Section 6.2 Future Work.)*

4.3 Improved Quadratic Algorithm

During the implementation of MAXCOVER, some errors were located in the pseudocode. However, the "gaps" problem is non-trivial to solve in the given $\mathcal{O}(n \log n)$ time, which we leave to Future Work *(See Section 6.2)*. Instead, we propose two new quadratic algorithms. Then, we tested the run-time of these algorithms on random strings and determined they take linear-time to process.
(See Section 4.3.3)

These algorithms rely on the following formula to compute the number of overlapped positions of consecutive occurences of v_i:

$$\mathcal{LCP}[i] - (\mathcal{SA}^*[k+1] - \mathcal{SA}^*[k]) \tag{4.3}$$

(See Section 4.2.2 on how this formula improves the original algorithm.)

4.3.1 Algorithm 1

(See Appendix B Algorithm 1.)

For $i \in [1..n]$, the steps taken are as follows:

1. Check if...
   (a) **Line 5:** ...$\boldsymbol{v_i}$ occurs more than once, i.e. $\mathcal{RSF}[i] > 1$;
   (b) **Line 6:** ...$\boldsymbol{v_i}$ is of the form: $\boldsymbol{bb}$, which means $\boldsymbol{v_i}$ has a border, or $\boldsymbol{v_i}$ is more than two letters;
   (c) **Line 7:** ...$\boldsymbol{v_i}$ has a border *(See MAXBORDER() below)*.
2. **Line 8-10:** Starting at the current index i, traverse backwards in the $\mathcal{LCP}$ array to find where $\boldsymbol{v_i}$ first occurs in $\mathcal{SA}$,
   i.e. determine r_1.
3. **Line 11-12:** Define the range (r_1,r_m) that v_i occurs in $\mathcal{SA}$.
4. **Line 13-14:** Sort the start position that v_i occurs in ascending order.
5. **Line 16:** For each $k \in (r_1, r_m)$:
   (a) **Line 17:** Determine the distance between each consecutive occurrence of v_i,
   i.e. $diff = (\mathcal{SA}^*[k+1] - \mathcal{SA}^*[k])$;
   (b) **Line 18:** If the distance $diff$ is smaller than the length of v_i...
   i.e. $diff < |v_i| = \mathcal{LCP}[i]$
   (c) **Line 19:** ...Then increment sum by the amount of overlap.
   i.e. $sum += |v_i| - diff$
6. **Line 20:** Set $\mathcal{OLP}[i]$ to the total number of overlapped positions for $\boldsymbol{v_i}$.
   i.e. $\mathcal{OLP}[i] = sum$

The excluded steps are simply the initialization of data structures.

We perform checks at Steps 1a-c to eliminate unnecessary computation where an overlap could not possibly occur, i.e. $\mathcal{OLP}[i] = 0$. To be exact:

- If $\boldsymbol{v}_i$ occurs only once then $\boldsymbol{v}_i$ is not a repeat, and thus cannot overlap with itself
- For a repeat $\boldsymbol{v}_i$ to overlap with itself, $\boldsymbol{v}_i$ must have a border *(See Figure 2.5 in Section 2.3)*

In fact, this leads to the next procedure we use: MAXBORDER(). *(See Appendix B.6.)* This function computes the border array $\beta[n]$, and returns the longest border length (which happens to be $\beta[n]$). Based on [6] (Algorithm 1.3.1), this procedure will traverse $\boldsymbol{v}_i$, computing the border for all prefixes of $\boldsymbol{v}_i$ until we compute the border for $\boldsymbol{v}_i$ itself.

Once we confirm $\boldsymbol{v}_i$ has a border, we must determine the start positions for all occurrences of $\boldsymbol{v}_i$ in $\boldsymbol{x}$ (Steps 2-4). Then we compare each consecutive occurrence to see if the distance is close enough that an overlap must occur (Step 5a-b). Suppose the distance is small enough, i.e. less than $|\boldsymbol{v}_i|$. In that case, we increase the total number of overlapped positions by $|\boldsymbol{v}_i|$ less the distance between each occurrence.

4.3.2 Algorithm 2

(See Appendix B Algorithm 2.)

This algorithm uses a stack to compute r_1 and r_m by checking the rise and fall of the $\mathcal{LCP}$ values.

We initialize the stack of pairs with the format: $(index, r_1)$.[3] We use this format to say: $\boldsymbol{v}_{index}$ starts at $\mathcal{SA}[r_1]$. We start the stack with $(1, 0)$ when counting index positions from 0, instead of 1. This means $\boldsymbol{v}_1$ has $r_1 = 0$, thus the first occurrence of $\boldsymbol{v}_1$ is at $\boldsymbol{x}[\mathcal{SA}[r_1]] = \boldsymbol{x}[\mathcal{SA}[0]]$.

We push onto the stack when we encounter a 'new' (or extended) $\boldsymbol{v}_i$. When the $\mathcal{LCP}$ value increases, i.e. $|\boldsymbol{v}_i| < |\boldsymbol{v}_{i+1}|$, we say we encountered a 'new' $\boldsymbol{v}_i$. This is 'new', or extended, because $\boldsymbol{v}_{i+1}$ is actually a longer prefix of $\boldsymbol{v}_i$, i.e. $\boldsymbol{v}_{i+1} = \boldsymbol{v}_i\boldsymbol{u}$.

We pop off the stack when we reach the r_m value for the current $\boldsymbol{v}_i$ and process the corresponding overlapped positions.

For $i \in [2..n]$, the steps taken are as follows:

1. **Line 7:** If the $\mathcal{LCP}$ value increases, then we push on the stack $r_1 = index - 1$.
2. **Line 8:** If the $\mathcal{LCP}$ values are the same, then we can disregard this computation since we are already computing the overlap for this $\boldsymbol{v}_i$.
3. **Line 9:** If the $\mathcal{LCP}$ value decreases, then process the values on the stack and pop them off:
   (a) **Line 10:** Continue to process the stack until we reach an index value where r_m for $\boldsymbol{v}_{top_i}$ continues past $index$, ...
   i.e. $\mathcal{LCP}[top_i] \leq \mathcal{LCP}[i]$
      i. **Line 11:** ...while keeping track of r_1 value for $\boldsymbol{v}_{top_i}$ as $prev_{r_1}$.
      ii. **Line 12-13:** We compute the overlapped positions of $\boldsymbol{v}_i$, then pop the respective $(index, r_1)$ off the stack.
      (See PROCESSSTACK() *below.)*
   (b) **Line 14-15:** If the stack is empty, we push back onto the stack $(index, prev_{r_1})$, implying we must continue to determine r_m for $\boldsymbol{v}_{top_i}$.
4. **Line 17-19:** Process the remaining values on the stack.
   (See PROCESSSTACK() *below.)*

The excluded steps are simply the initialization of data structures.

In Step 1, we set the r_1 value for $\boldsymbol{v}_{index}$ to be the previous $index$. This is because $\mathcal{LCP}[index]$, by definition, is the longest common prefix between the previous $(index-1)$ and the current $(index)$ lexicographical suffix.

[3]Note: In this section, we use $index$ and i interchangeably for clarity and brevity, respectively.

In Step 2, we explained that we are already computing this $\boldsymbol{v}_i$, so we can skip this computation to avoid duplicating work.

In Step 3, when the $\mathcal{LCP}$ value decreases, we have reached the r_m value of this $\boldsymbol{v}_i$. Now that we have both r_1 (from the top of the stack) and r_m values, we can process the overlap of all occurrences of $\boldsymbol{v}_i$ using the ProcessStack() procedure (Step3a *ii*).

In Step 3a, we repeat this process until we reach a pair (top_i, top_{r_1}) that implies the r_m value for $\boldsymbol{v}_{top_i}$ continues past the current *index*. That is:

$$\mathcal{LCP}[top_i] \leq \mathcal{LCP}[i] \tag{4.4}$$

$$\implies |\boldsymbol{v}_{top_i}| \leq |\boldsymbol{v}_i| \tag{4.5}$$

$$\implies \boldsymbol{v}_i = \boldsymbol{v}_{top_i}\boldsymbol{u} \tag{4.6}$$

We keep track of the previous r_1 value (top_{r_1}) using $prev_{r_1}$ in Step 3a i, so we can push it back onto the stack in Step 3b. This is because we have not yet found the r_m for $\boldsymbol{v}_{top_i}$, so we need to push it back onto the stack to compute in the future.

Since we push some values back onto the stack, when we reach the final *index*, we can assume any remaining $\boldsymbol{v}_i$ has $r_m = n$. Thus, in Step 4, we process any remaining values on the stack, using ProcessStack() again.

Now, we introduce the pseudocode for ProcessStack(). *(See Appendix B.7.)* This pseudocode is a near replica of Algorithm 1, less the while loop and slight adjustments to variable names. Therefore, this procedure also takes advantage of the maxborder() procedure.

This structure is intended to improve upon Algorithm 1's tendency to recompute r_1 by traversing backwards multiple times. Instead, Algorithm 2, inspired by the original $\mathcal{O}(n^2)$ algorithm, will keep track of r_1 as we traverse the $\mathcal{LCP}$ array from left to right.

4.3.3 Runtime Comparison

(See Tables A.5, A.6, and A.7.)

See Charts:

- 4.3 & 4.4 (Original $\mathcal{O}(n^2)$ Algorithm);
- 4.5 & 4.6 (Improved $\mathcal{O}(n^2)$ using a while loop);
- 4.7 & 4.8 (Improved $\mathcal{O}(n^2)$ using a Stack)

In order to compare the runtime for these three algorithms, we placed the start and end markers for a timer surrounding only the computation of the $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, $\mathcal{OLP}$, $\mathcal{RSPC}$ arrays and the list of maximal covers. The initial setup and output (i.e. print statements) were excluded. The time is measured by the number of clock ticks per second, then translated into seconds as the unit of measure for runtime comparison.

We created test data of 60 unique randomly generated strings. Although an average of multiple tests per string type would ideally be performed, only one test was performed per string because of the time it took to compute *(See analysis below)*.

This data varied based on twenty string lengths: half in the thousands ($|\boldsymbol{x}| \in \{1000, .., 9000\}$) and the other half in the millions ($|\boldsymbol{x}| \in \{1M, .., 9M\}$). These datasets also varied in alphabet size ($|\Sigma| = \{2, 3, 4\}$) to simulate binary strings, triples, and DNA strings.

Through our experiments, we realized that the larger the alphabet size, the faster the algorithms perform to compute maximal covers. This is because small alphabet sizes are more likely to result in highly periodic strings, which will require more processing to compute overlaps.

The runtimes for the Original $\mathcal{O}(n^2)$ algorithm took significantly longer to compute than the improved algorithms. For $|\boldsymbol{x}| \in \{1000, .., 9000\}$, the Original $\mathcal{O}(n^2)$ algorithm took up to 5 seconds to compute. In contrast, the improved algorithms took no more than 0.03 seconds. For $|\boldsymbol{x}| \in \{1M, .., 9M\}$, the Original $\mathcal{O}(n^2)$ has missing data points. This is because the algorithm takes superficially too long to complete computation, especially considering the 9M computation took $\sim$ 55 hours $\approx$ 2.3 days. In contrast, the improved algorithms took under a minute (maximum time taken was $\sim$ 45.7 seconds).

Although still quadratic, the overall overhead is reduced due to the simplified algorithm with fewer data structures to construct. Thus, performing linear-time in practice. This is most clearly seen when comparing the charts with binary alphabets ($|\Sigma| = 2$) between the original versus the improved algorithms.

The assumption would be that Algorithm 2 would perform better in practice than Algorithm 1 due to the improvement of using the stack to reduce the duplication of work to compute r_1. However, this is not the case. Algorithm 1 performed marginally better – most clearly seen when the string lengths are 9M. This is most likely due to the overhead required to set up the stack and the cost for push and pop which takes more

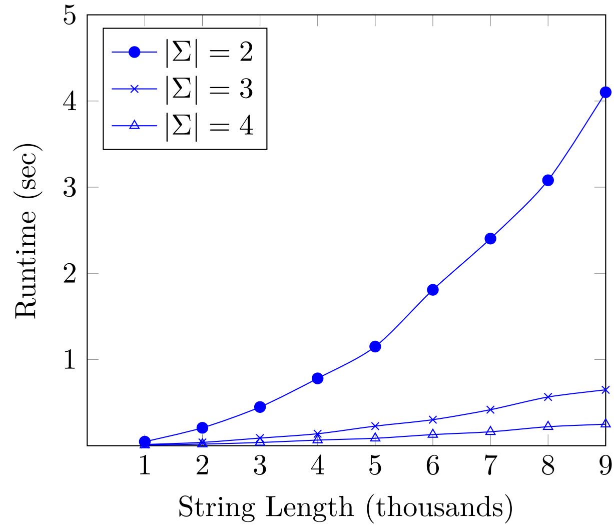

FIGURE 4.3: Plot of **Original** $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$.

time than a simple traversal repeatedly, even at millions of strings. Therefore, simplicity in algorithms is far more optimal in the practical case.

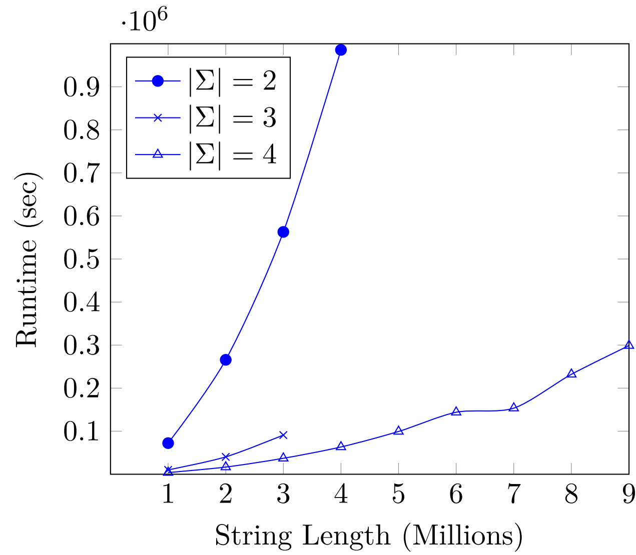

FIGURE 4.4: Plot of **Original** $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$.

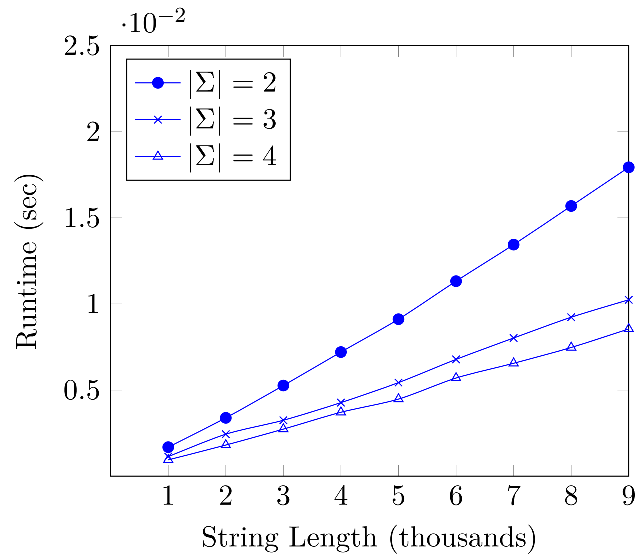

FIGURE 4.5: Plot of Improved $\mathcal{O}(n^2)$ Algorithm 1 using **While** loop Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$.

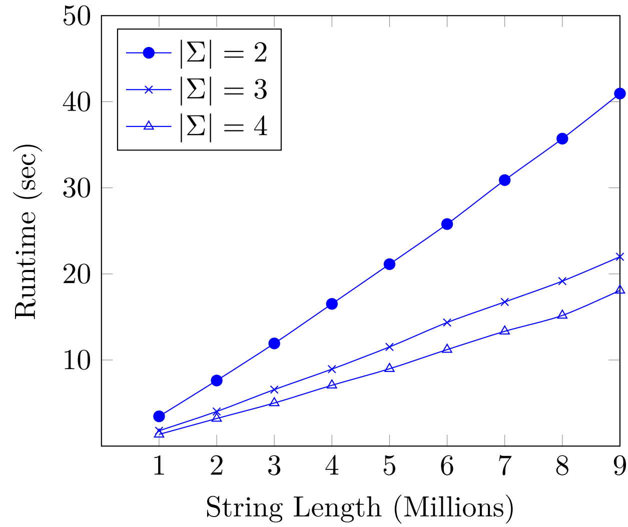

FIGURE 4.6: Plot of Improved $\mathcal{O}(n^2)$ Algorithm 1 using **While** loop Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$.

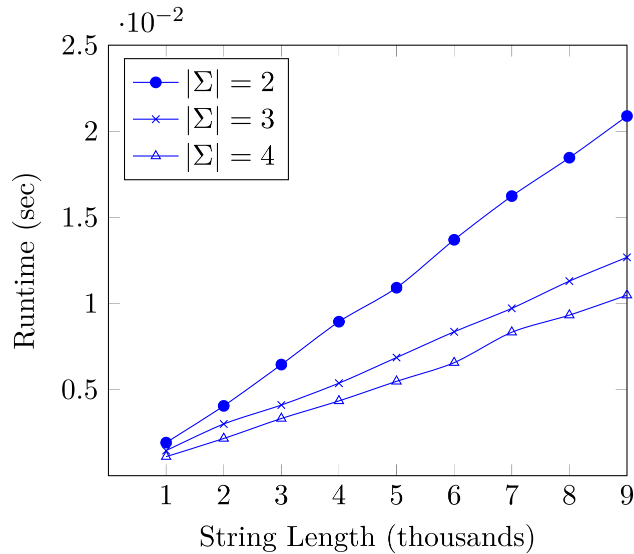

FIGURE 4.7: Plot of Improved $\mathcal{O}(n^2)$ Algorithm 2 using **Stack** Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1000, .., 9000\}$.

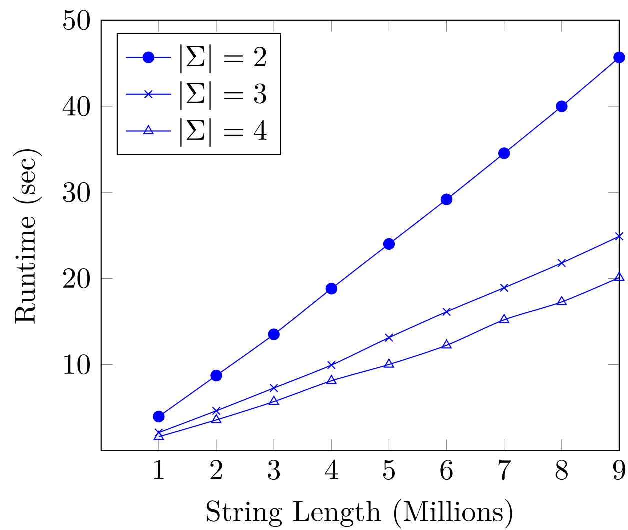

FIGURE 4.8: Plot of Improved $\mathcal{O}(n^2)$ Algorithm 2 using **Stack** Runtime in Seconds of Alphabet sizes $|\Sigma| \in \{2, 3, 4\}$ of Random Strings of Lengths: $|\boldsymbol{x}| = \{1M, .., 9M\}$.

Chapter 5

Experiments

The content in this chapter contains information published in the research paper [2]

> **Title:** *'Computing Maximal Covering Subsequences of Protein Sequences'*
> **Authors**[1]**:** G. Brian Golding, Holly Koponen, Neerja Mhaskar, W. F. Smyth
> **Conference:** Mathematical Foundations in Bioinformatics 2021 (MatBio)
> **Journal:** Journal of Computational Biology
> **Status:** Submitted with revisions

5.1 Machine Specifications

The MAXCOVER software was developed in C++ and run on a Microsoft Windows 10 Pro (10.0.19042 Build 19042) machine with Intel(R) Core(TM) i9-10980XE CPU @3.00GHz (3000 Mhz, 18 Cores, 36 Logical Processors) and CORSAIR Vengeance RGB PRO 128GB (4x32GB) DDR4 3600 (PC4-28800) RAM. Testing was performed on an Ubuntu Virtual Machine (Oracle VM VirtualBox Manager) with 81804MB Base Memory and 18 CPUs. The implementation used for $\mathcal{SA}$ array is by Yuta Mori[2] [56], $\mathcal{LCP}$ array by Simon Puglisi [4], and $\mathcal{RSF}$ array by Neerja Mhaskar [1]. The software was made available on Github[3] as an open-source software.

[1] Authors in alphabetical order by surname.
[2] `https://sites.google.com/site/yuta256/`
[3] `https://github.com/hollykoponen/MAXCOVER`

5.2 Performance

MAXCOVER is currently the only software available for the computation of maximal covers. For this reason we needed to find another basis for evaluating its efficiency. To do this, we used the fact that NE repeats and maximal covers are both computed using identical data structures: we made minor extensions to MAXCOVER to also compute NE repeats.

We were thus able to test MAXCOVER against an existing alignment tool, MUMmer [3], which computes NE repeats to identify a minimum match length for a given genome. Although the primary software functions are different — MAXCOVER computes maximal covers. In contrast, MUMmer is an alignment tool — we can assess MAXCOVER's performance by comparing its computation of NE repeats to that of MUMmer.

5.2.1 MUMmer

The source code for MUMmer describes the software as follows: "MUMmer is a system for rapidly aligning DNA and protein sequences." We use MUMmer 4.x [3], which currently has to date: 658 citations. This is a well-known and commonly used software for DNA and protein sequence alignment in the biology community. One particular feature of MUMmer is repeat-match, which computes NE repeats.

repeat-match

The source code for repeat-match[4] describes the feature as follows:

> **Description:** Finds exact repeats within a single sequence.
> **Input:** Single sequence in `FastA` format to search for repeats.

The program repeat-match from MUMmer uses suffix-trees to identify maximal exact repeat regions of a given minimum match length for a given genome. (See Figure 5.1.) The output of repeat-match has three columns: Start1 and Start2 specify the starting positions of exact matching region pairs, while the last column gives the length of the region. Note that if the given minimum match length was '2', only lines 1-9 would be returned.

We discovered that repeat-match does not report redundancies consistently. In Figure 5.1 lines 3-4 report the start positions {1, 7, 11} of length 7, while in line 5, positions {1, 7} are redundantly repeated. However, when we look at lines 6-9, the pair (5, 15) is not redundantly reported. Similarly, we know that at $\boldsymbol{x}[11..17]$ = ADAQADA, so we know $\boldsymbol{x}[11..13]$ = ADA, but position 11 is not reported between lines 6-9. Therefore, repeat-match does not consistently report redundancies and/or all repeat-matches.

[4] https://github.com/mummer4/mummer/blob/6c0da4101a55a0f89ecd00a5e569501a9a9f9ecd/README.md

	Long Exact Matches:		
1			
2	Start1	Start2	Length
3	7	11	7
4	1	11	7
5	1	7	7
6	7	15	3
7	1	15	3
8	5	7	3
9	1	5	3
10	15	17	1
11	5	17	1
12	11	17	1
13	1	17	1
14	13	15	1
15	5	13	1
16	11	13	1
17	1	13	1
18	9	15	1
19	5	9	1
20	9	11	1
21	1	9	1
22	3	15	1
23	3	5	1
24	3	11	1
25	1	3	1

Figure 5.1: Output from MUMmer's repeat-match of NE repeats for the string $\boldsymbol{x} = ADAQADADADAQADAQADA$.

1	Substring Repeat	Length	Position
2	1	7	11, 1, 7
3	2	3	15, 5, 11, 1, 7
4	3	1	17, 15, 5, 11, 1, 7, 13, 3, 9

Figure 5.2: MAXCOVER output of NE repeats for $\boldsymbol{x} = ADAQADADADAQADAQADA$.

5.2.2 Extension of MAXCOVER

Figure 5.2 shows the corresponding output from MAXCOVER. It has a more condensed format: 'NE Substring Repeat #' is a label for the non-extendible substring region to differentiate between matching regions of the same length; 'Length' is the length of the substring region; and 'Position' is a list of the start positions of the substring regions in $\boldsymbol{x}$. This format consistently reports all occurrences and uses a condensed tabular format.

Titin is the longest protein known. It is composed of 363 exons joined together to form the complete protein. [58] It is a structural protein used in the formation of muscles. We applied MAXCOVER to identify the top ten longest non-extendable repeats in this protein. Despite being the longest protein and a highly repetitive protein, there are only a maximum of four repeats of length 37 amino acids within the top ten longest. There are more numerous repeats in much smaller proteins within *C. elegans (See Table A.1.)*. Given the highly repetitive nature of titin, the small number of repeats might indicate that sequence divergence has occurred among the repeats. This suggests that a future variant of the current software that includes approximate matches would be useful. The software might then identify many more long repeats within this protein.

5.2.3 Protein Dataset

We acquired protein data from the NCBI (National Center for Biotechnology Information) databases. We chose proteins from four taxonomically distinct model organisms and collected the entire complement of proteins from these organisms. We used FASTA files of the protein sequences for four species: *Arabidopsis thaliana* (48,265 protein sequences), *Caenorhabditis elegans* (28,350 protein sequences), *Drosophila melanogaster* (30,717 protein sequences), and *Homo sapiens* (116,263 protein sequences). To ensure good quality sequences, proteins that contained unknown amino acids (X's) were removed from the dataset. The proteins processed ranged in size from 20 to 100,000 amino acids.

To provide a good representative sample and avoid excessive run times, we sampled 10,000 protein sequences per species, thus altogether 40,000. This ensured that the time required for testing was less than thirty minutes. The sequences were chosen randomly using the Python RANDOM.SAMPLE function[5].

Experimental Results

Representative examples of maximal covers for each of the four taxa are shown in Tables A.1 – A.4 *(See Appendix A.)*, which use ***amino acids*** (*aa*) as the unit of measure for the length of the strings.

As expected *C. elegans* had the least extensive repetitive proteome while *Homo sapiens* had the most extensive repetitive proteome. Nevertheless, a maximal cover of length 33 is a highly repeated substring within the *C. elegans* proteome, being repeated 83 times. Within *C. elegans* the maximum number of positions covered occurs at an intermediate cover length (30-39aa), and the maximum percent coverage is also at an intermediate length (30-39aa). Again, with Arabidopsis, the maximum number of positions covered occurs at an intermediate cover length (20-29aa) and the maximum percent coverage at 70-79aa. For *D. melanogaster* the maximum number of positions covered is for a cover of 50-59aa, and the maximum percent coverage is at 400-699. For humans, both of these occur with large covers of lengths 700-799aa and 600-699aa.

A possible reason for this discrepancy is that with the large population size of *C. elegans* only smaller protein repeats can exist before mutations destroy their similarity. However, large repeats can survive within the historically much smaller human population due to higher amounts of random genetic drift fixing the repeats before mutations can disrupt them. This hypothesis could be tested by more extensive applications of MAXCOVER and carefully chosen data. For instance, a comparative data sampling could be chosen where the population genetics is better understood.

[5] https://docs.python.org/3/library/random.html

5.2.4 MAXCOVER vs. MUMmer's repeat-match

We compare the output of MUMmer's REPEAT-MATCH and of MAXCOVER in Table 5.1.

REPEAT-MATCH took 29.5 minutes to compute the NE substring regions. In contrast, MAXCOVER required only 1.25 minutes – faster by a factor of more than 20. This difference in speed is no doubt largely due to the use of suffix trees by REPEAT-MATCH, whereas MAXCOVER uses suffix arrays.

In terms of accuracy, REPEAT-MATCH does not report every pair of start positions for matching regions. In contrast, MAXCOVER reports all start positions for matching regions. As we have seen, the output of REPEAT-MATCH is lengthy. It makes it difficult to recognize useful information. In contrast, MAXCOVER is condensed, and it is clear which positions refer to the same region.

	MUMmer `repeat-match`	**MAXCOVER**
Data Structure	Suffix Trees	Suffix Arrays
Time	Slower (29 min. 28 sec.)	Faster (1 min. 15 sec.)
Accuracy	Does not report every pair of start positions for matching regions	Reports ALL start positions for matching regions
Output Format	Lengthy; Hard to decipher useful information	Condensed; Clear which set of positions refer to the same region

TABLE 5.1: Comparison between MUMmer's `repeat-match` and MAXCOVER software to compute NE substrings on a random sample of 40,000 protein sequences.

Chapter 6

Conclusion and Future Work

6.1 Discussion

Although the AllPartialCovers problem solves computing the *shortest* maximal covers, the solution presented by [9] uses *suffix trees*, which are not ideal for computing long strings. Mhaskar and Smyth [1] remedy this concern using *suffix arrays* instead.

We implemented the first ever software (MAXCOVER) to compute maximal covers in $\mathcal{O}(n^2)$-time based on pseudocode in [1]. In doing so, we implemented data structures not described in [1], namely:

- R_1 & R_m arrays;
- computing all runs of a string $\boldsymbol{x}$ using the COMPUTE_RUNS() procedure and storing them in a hashtable RUNSHT;
- computing eligible runs of a substring $\boldsymbol{u}$ using the EXRUN() function.

During the implementation of the MAXCOVER software, errors were located in the pseudocode. The "gaps" problem is non-trivial to solve in the given $\mathcal{O}(n \log n)$ time. *(See Future work below.)* Instead, we proposed two new quadratic algorithms, which performed in average-case linear $\mathcal{O}(n)$ time when tested on random strings. We showed that despite being theoretically the same $\mathcal{O}(n^2)$ runtime, simplicity in algorithms performs far more optimally in practice.

We showed how string processing algorithms apply to multiple fields in data analytics. In particular, we tested the practical viability of MAXCOVER on protein sequences against a well-known software MUMmer. However, since MAXCOVER is the only software to compute maximal covers, we extended MAXCOVER to also compute NE repeats.

In comparing the performance of MAXCOVER versus MUMmer's REPEAT-MATCH feature, we determined MAXCOVER is an order-of-magnitude faster than MUMmer with much lower space requirements; while producing a more exact, compact, and user-friendly specification of the repeats.

6.2 Future Work

Future directions include improving the current algorithms used in MAXCOVER, performing more tests with the software, and further extending the software to compute more types of strings.

Firstly, improving the current algorithms used in MAXCOVER includes:

- The current EXRUN() function can be improved by using the algorithm proposed by Bannai et al. in [57] based on Lyndon words.
- Through implementation of the $\mathcal{O}(n \log n)$ algorithm presented by [1], several errors were located. In particular, the algorithm does not handle all possible cases in the SORT() and COMPUTE_SORTED_RANGE() procedures, hence the "gaps" problem. Future work will involve a detailed analysis of these errors and explore possible solutions to the "gaps" problem, such as interval trees.
- Furthermore, investigate optimizing the algorithm to a linear time algorithm – potentially using compressed data structures to do so.

Secondly, testing the uses of MAXCOVER and maximal covers includes:

- Testing the hypothesis that only smaller protein repeats can exist in large population sizes before mutations destroy their similarity.
  - This can be tested by more extensive applications of MAXCOVER and carefully chosen data. For instance, a comparative data sampling could be chosen where the population genetics is better understood.
- Explore the usefulness of maximal covers in data compression through tests on various string types.

Finally, investigating further extending MAXCOVER:

- MAXCOVER currently only computes on regular strings. However, we can extend the software to include an analysis of indeterminate strings.
- With an extension to compute indeterminate strings, we may extend the software to perform multiple sequence alignment of biological sequences, similar to MUMmer.

Much work remains to be explored about maximal covers, especially through using the MAXCOVER software we developed.

Appendix A

Tables

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001254049.1_3963	117	1	14	11 %	0.311 ms
2	NP_001024113.1_1693	73	2	16	21 %	0.244 ms
3	NP_001255514.1_4822	43	3	6	13 %	0.267 ms
4	NP_001368117.1_12245	76	4	8	10 %	0.26 ms
5	NP_001309530.1_8612	98	5	10	10 %	0.292 ms
6	NP_496365.2_17882	98	6	18	18 %	0.362 ms
7	NP_001334228.1_8988	351	7	114	32 %	0.595 ms
8	NP_494447.1_16558	604	8	256	42 %	1.561 ms
9	NP_001337314.1_9082	80	9	18	22 %	0.283 ms
10 - 19	NP_506718.2_24803	74	13	26	35 %	0.381 ms
20 - 29	NP_001257068.1_6103	223	24	96	43 %	0.454 ms
30 - 39	NP_001350976.1_9553	3317	33	2739	82 %	16.846 ms
40 - 49	NP_001368148.1_12271	826	41	160	19 %	1.46 ms
50 - 59	NP_503527.1_22569	274	57	145	52 %	0.624 ms
60 - 69	NP_493059.2_15693	468	64	74	15 %	0.74 ms
70 - 79	NP_001033466.2_2263	257	75	95	36 %	0.511 ms
80 - 89	-	-	-	-	-	-
90 - 99	-	-	-	-	-	-
100 - 199	NP_494641.2_16683	677	105	168	24 %	1.341 ms
200 - 299	NP_494430.2_16543	626	219	339	54 %	1.299 ms
300 - 399	NP_493601.2_16025	4647	378	1134	24 %	12.724 ms
400 - 499	-	-	-	-	-	-
500 - 599	-	-	-	-	-	-
600 - 699	-	-	-	-	-	-
700 - 799	NP_001366805.1_11196	13083	768	1536	11 %	54.72 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table A.1: Examples of maximal covers for *C. elegans* proteins within a particular range of maximal cover lengths computed using MAXCOVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001260924.1_8063	369	1	38	10 %	0.687 ms
2	NP_001285973.1_11292	43	2	6	13 %	0.252 ms
3	NP_001261881.2_8903	51	3	6	11 %	0.245 ms
4	NP_524256.1_15218	62	4	28	45 %	0.351 ms
5	NP_001285816.1_11135	66	5	10	15 %	0.25 ms
6	NP_001262537.1_9508	126	6	60	47 %	0.424 ms
7	NP_523815.2_14812	1334	7	172	12 %	2.876 ms
8	NP_609716.2_18399	78	8	24	30 %	0.361 ms
9	NP_572939.1_16848	185	9	59	31 %	0.488 ms
10 - 19	NP_729001.2_26993	241	14	92	38 %	0.458 ms
20 - 29	NP_476941.2_13802	111	25	33	29 %	0.421 ms
30 - 39	NP_650373.1_22567	173	33	66	38 %	0.478 ms
40 - 49	NP_572186.1_16259	407	44	176	43 %	0.75 ms
50 - 59	NP_610937.4_19403	4011	52	1404	35 %	11.068 ms
60 - 69	-	-	-	-	-	-
70 - 79	NP_729000.2_26992	414	79	158	38 %	0.742 ms
80 - 89	NP_647938.1_20661	431	88	221	51 %	0.75 ms
90 - 99	NP_572306.1_16355	300	97	173	57 %	0.592 ms
100 - 199	NP_996410.1_30113	1583	184	318	20 %	3.744 ms
200 - 299	-	-	-	-	-	-
300 - 399	-	-	-	-	-	-
400 - 499	NP_727078.1_26214	533	456	532	99 %	0.821 ms
500 - 599	-	-	-	-	-	-
600 - 699	NP_995994.1_29765	762	684	760	99 %	1.579 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

TABLE A.2: Examples of maximal covers for *D. melanogaster* proteins within a particular range of maximal cover lengths computed using MAX-COVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001031361.2_685	653	1	67	10 %	1.182 ms
2	NP_001332341.1_20487	26	2	4	15 %	0.232 ms
3	NP_850977.1_46441	58	3	6	10 %	0.325 ms
4	NP_001031813.1_1106	35	4	7	20 %	0.279 ms
5	NP_001329516.1_17663	66	5	10	15 %	0.334 ms
6	NP_172240.1_21498	44	6	12	27 %	0.304 ms
7	NP_189459.1_30049	111	7	14	12 %	0.377 ms
8	NP_001322312.1_10459	124	8	24	19 %	0.31 ms
9	NP_001321372.1_9520	68	9	15	22 %	0.353 ms
10 - 19	NP_568552.1_43794	321	19	57	17 %	0.543 ms
20 - 29	NP_001323842.1_11989	564	24	360	63 %	1.64 ms
30 - 39	NP_174445.1_23044	463	31	53	11 %	0.801 ms
40 - 49	NP_195887.2_34639	371	48	96	25 %	0.536 ms
50 - 59	NP_001154390.2_4083	744	57	114	15 %	1.278 ms
60 - 69	NP_176714.4_24533	318	61	122	38 %	0.72 ms
70 - 79	NP_001328999.1_17146	152	76	152	100 %	0.408 ms
80 - 89	NP_001325858.1_14005	411	89	159	38 %	0.695 ms
90 - 99	-	-	-	-	-	-
100 - 199	NP_849291.1_44918	228	152	228	100 %	0.425 ms
200 - 299	NP_851029.1_46486	305	228	304	99 %	0.643 ms
300 - 399	NP_173553.1_22430	430	304	332	77 %	1.026 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table A.3: Examples of maximal covers for Arabidopsis proteins within a particular range of maximal cover lengths computed using MAX-COVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	XP_006716126.1_69540	707	1	79	11 %	1.304
2	NP_001341932.1_30631	70	2	22	31 %	0.326
3	NP_006266.2_48075	343	3	77	22 %	0.512
4	NP_001135954.1_6750	67	4	8	11 %	0.318
5	XP_011524892.1_80014	474	5	50	10 %	0.744
6	NP_009048.1_48830	241	6	36	14 %	0.427
7	XP_016882728.1_107042	389	7	42	10 %	0.633
8	NP_001340727.1_29802	403	8	64	15 %	0.718
9	XP_016882449.1_106829	602	9	63	10 %	0.936
10 - 19	NP_001334947.1_25817	515	18	108	20 %	1.047
20 - 29	XP_005255792.1_64442	395	27	54	13 %	0.55
30 - 39	NP_001159711.1_8644	317	33	50	15 %	0.541
40 - 49	NP_001123991.1_5873	605	44	88	14 %	1.179
50 - 59	XP_011521003.1_78175	893	55	110	12 %	1.671
60 - 69	NP_031372.2_49017	676	64	124	18 %	1.041
70 - 79	XP_016883383.1_107510	557	78	139	24 %	1.118
80 - 89	XP_016878127.1_103859	533	85	170	31 %	0.787
90 - 99	NP_001375298.1_41921	1110	94	188	16 %	1.81
100 - 199	XP_011529785.1_82360	528	116	232	43 %	0.841
200 - 299	XP_011514536.1_75210	597	292	365	61 %	0.958
300 - 399	XP_011537716.1_86000	1782	327	456	25 %	4.41
400 - 499	XP_011509528.1_72895	7795	485	970	12 %	22.539
500 - 599	-	-	-	-	-	-
600 - 699	NP_066289.3_53510	684	608	684	100 %	1.46
700 - 799	NP_001289300.1_18913	3794	724	1692	44 %	13.823
800 - 899	NP_056198.2_50743	2818	893	1137	40 %	8.226

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table A.4: Examples of maximal covers for human proteins within a particular range of maximal cover lengths computed using MAXCOVER.

Original $O(n^2)$ Algorithm

	Alphabet size		
String Length	**2**	**3**	**4**
1k	0.046832	0.014094	0.007055
2k	0.207753	0.038506	0.018617
3k	0.448886	0.088368	0.037204
4k	0.781034	0.138716	0.065224
5k	1.15024	0.227379	0.086347
6k	1.80882	0.301818	0.12895
7k	2.40338	0.417859	0.161339
8k	3.08018	0.565143	0.220406
9k	4.10172	0.648165	0.249238
1M	72265	10130.6	3879.28
2M	265793	40313.6	16694.3
3M	562898	90967	37068.5
4M	985790	-	63357.8
5M	-	-	99369.1
6M	-	-	144026
7M	-	-	153545
8M	-	-	232124
9M	-	-	298793

Table A.5: Original $\mathcal{O}(n^2)$ Algorithm Runtime in Seconds

Improved $\mathcal{O}(n^2)$ Algorithm using While loop			
	Alphabet size		
String Length	**2**	**3**	**4**
1k	0.001683	0.001132	0.000946
2k	0.003385	0.002437	0.001806
3k	0.005263	0.003246	0.002731
4k	0.007206	0.004271	0.00371
5k	0.009115	0.005433	0.004474
6k	0.011323	0.006781	0.005698
7k	0.013446	0.008023	0.006555
8k	0.015689	0.009229	0.007472
9k	0.017936	0.010241	0.00855
1M	3.45577	1.78304	1.36231
2M	7.61682	4.01944	3.20243
3M	11.9251	6.5605	5.00429
4M	16.5183	8.94017	7.06271
5M	21.1296	11.5189	8.98169
6M	25.7895	14.3557	11.2073
7M	30.895	16.7438	13.3317
8M	35.7097	19.1625	15.174
9M	40.9496	21.994	18.0851

TABLE A.6: Improved $\mathcal{O}(n^2)$ Algorithm using While loop Runtime in Seconds

Improved $\mathcal{O}(n^2)$ Algorithm using Stack			
	Alphabet size		
String Length	**2**	**3**	**4**
1k	0.00192	0.001455	0.001105
2k	0.004054	0.003002	0.002159
3k	0.006452	0.004103	0.00332
4k	0.008943	0.005373	0.004342
5k	0.010915	0.006861	0.005478
6k	0.013699	0.008356	0.006563
7k	0.016236	0.009717	0.008331
8k	0.018467	0.011299	0.009324
9k	0.020886	0.01268	0.010483
1M	3.96427	2.09729	1.62154
2M	8.72573	4.62929	3.56961
3M	13.5106	7.27771	5.69653
4M	18.8211	9.94309	8.1138
5M	24.008	13.1321	10.0138
6M	29.1835	16.1299	12.241
7M	34.5443	18.9226	15.1982
8M	39.9884	21.7975	17.2768
9M	45.6818	24.8988	20.1043

TABLE A.7: Improved $\mathcal{O}(n^2)$ Algorithm using Stack Runtime in Seconds

Appendix B

Pseudocode

1: **procedure** COMPUTE_FREQUENCY($i,\ j,\ p,\ i',\ j'$)
2: $|\boldsymbol{r}| \leftarrow j - i \cancel{-1} +1$ $\triangleright\ \boldsymbol{r} = \boldsymbol{b}^e\boldsymbol{b}' = (i, j, p)$
3: $|\boldsymbol{u}| \leftarrow j' - i' \cancel{-1} +1$
4: $|\boldsymbol{b}'| \leftarrow (j - i \cancel{-1} +1) \mod p$
5: $e \leftarrow \lfloor \frac{|\boldsymbol{r}|}{p} \rfloor$
6: **if** $((i' - i) \mod p = 0)$ **then** $|\boldsymbol{x}| \leftarrow 0$
7: **else**
8: $|\boldsymbol{x}| \leftarrow p - (i' - i) \mod p$
9: $|\boldsymbol{y}| \leftarrow (|\boldsymbol{u}| - |\boldsymbol{x}|) \mod p$ $\triangleright\ \boldsymbol{u} = \boldsymbol{x}\boldsymbol{b}^{e'}\boldsymbol{y}$
10: $e' \leftarrow \frac{|\boldsymbol{u}|-(|\boldsymbol{x}|+|\boldsymbol{y}|)}{p}$
11: $\delta \leftarrow e - e'$
12: **if** $(|\boldsymbol{x}| = 0)$ **then**
13: **if** $(|\boldsymbol{y}| \leq |\boldsymbol{b}'|) \cancel{)}$ **then return** $\delta + 1$
14: **else**
15: **return** δ
16: **else**
17: **if** $(|\boldsymbol{y}| \leq |\boldsymbol{b}'|)$ **then return** δ
18: **else**
19: **return** $\delta - 1$

FIGURE B.1: `Compute_Frequency()` computes the frequency $f_{\boldsymbol{r},\boldsymbol{u}}$.
Correction of Line 2-4: Changed from -1 to $+1$.
Correction of Line 13: Removed an extra round bracket.

```
procedure COMPUTE_R1()
    Initialize R1 as an array of size n
    Initialize a stack
    lastr1 ← 0
    R1[0] ← −1
    stack.push(0, 0)                    ▷ Let us refer to stack.top as merely top
    for (i from 0 to n) do
        if ((LCP[i] ≠ 0) and (LCP[i] = LCP[top[0]])) then
            R1[i] ← top[1]
        else if (LCP[i] > LCP[top[0]]) then
            stack.push(i, i − 1)
            R1[i] ← top[1]
        else
            while (LCP[i] < LCP[top[0]]) do
                lastr1 ← top[1]
                stack.pop()
            if ((LCP[i] ≠ 0) and (LCP[i] = LCP[top[0]]) ) then
                R1[i] ← top[1]
            else if (LCP[i] = 0) then
                stack.push(0, −1)
                R1[i] ← −1
            else if (LCP[i] > LCP[top[0]]) then
                stack.push(i, lastr1)
                R1[i] ← lastr1

procedure COMPUTE_RM()
    Initialize an array called Rm of size n
    for i from 0 to n do
        if (RSF[i] ≠ 0) then
            Rm[i] ← R1[i] + RSF[i] − 1
```

FIGURE B.2: Pseudocode to compute the R_1 & R_m arrays.

```
procedure EXRUN(a, b)
    for per from 0 to (b−a+1)/2 do
        runs ← all runs r_k = (i′, j′, p′) in runsHT with period p′ = per
        if runs.size() ≠ empty then
            for each r_k in runs do
                if r_k.i′ ≤ a and b ≤ r_k.j′ then return r_k
```

FIGURE B.3: Pseudocode for EXRUN() procedure to compute eligible run.

```
procedure LC(lc_arr, RANK, a, b)
    return RMQ(
            lc_arr,
            min(RANK[a], RANK[b]) + 1,
            max(RANK[a], RANK[b])
    )
```

FIGURE B.4: RMQ() procedure to compute the Range Minimum Query of $\mathcal{LCP}$ or $\mathcal{LCS}$ using RMQ_SUCCINCT procedure in [4]

```
procedure COMPUTE_RUNS(LCP, LCS, RANK, REV_RANK)
    top ← 0
    for per from 1 to per ≤ ⌊input_size / 2⌋ do
        pos ← per − 1
        while (pos + per < n) do
                                        ▷ LC() gives the RMQ of LCP and LCS
            right ← LCP[lc(LCP, RANK, pos, pos + per)]
            left ← LCS[lc(LCS, REV_RANK, n − pos − 1, n − pos − per − 1)]
                                                                         ▷
            if (left + right > per) then
                run_pair ← [pos − left + 1, pos + per + right − 1]
                runsHT.insert( per, run_pair )
            pos ← pos + per
```

FIGURE B.5: COMPUTE_RUNS() procedure to determine all runs of a given string $\boldsymbol{x}$. Based on algorithm DETECTRUNS(u) by Crochemore et al. in [5]

Algorithm 1 Improved $O(n^2)$ to compute $\mathcal{OLP}$ array using While loop

```
1: procedure COMPUTE_OLP()
2:     Initialize arrays OLP and SA* of size n full of 0's.
3:     for i from 1 to n − 1 do
4:         v_i ← x[SA[i]..SA[i] + LCP[i]]
5:         if (RSF[i] > 1
6:                 and ((LCP[i] = 2 and v_i[0] = v_i[1]) or LCP[i] > 2)
7:                 and MAXBORDER(v_i, LCP[i]))  then
8:             i' ← i
9:             while (LCP[i' − 1] ≥ LCP[i]) do
10:                i' ← i' − 1
11:            r_1 ← i' − 1
12:            r_m ← r_1 + RSF[i] − 1
13:            Copy SA[r_1..r_m] to SA*[r_1..r_m]
14:            Sort SA*[r_1..r_m] in ascending order
15:            sum ← 0
16:            for k from 1 to SA*.size()-1 do          ▷ SA*.size() is RSF[i]
17:                diff ← SA*[r_1 + k] − SA*[r_1 + k − 1]
18:                if (diff < LCP[i]) then
19:                    sum ← sum + LCP[i] − diff
20:        OLP[i] ← sum
21:    return OLP
```

```
procedure MAXBORDER(text, n)
    if n = 0 or 1 then
        return 0                      ▷ If v_i is empty or a single letter, then return False
    else
        Initialize array β of size n full of 0's.
        for z from 1 to n − 1 do
            b ← β[i]
            while (b > 0 and text[z + 1] ≠ text[b]) do
                b ← β[b − 1]
            if (text[b] = text[z + 1]) then
                β[z + 1] ← b + 1
        return β[n − 1]               ▷ If β[n − 1] ≠ 0 then return True
```

FIGURE B.6: MAXBORDER() procedure to compute the Border Array of a given text, and return the length of the longest border. Code based on Algorithm 1.3.1 in [6].

Algorithm 2 Improved $O(n^2)$ to compute $\mathcal{OLP}$ array using Stack

1: **procedure** COMPUTE_$\mathcal{OLP}$()
2: Initialize arrays $\mathcal{OLP}$ and $\mathcal{SA}^*$ of size n full of 0's.
3: Initialize a stack of pairs $(index, r_1)$ ▷ We refer to *stack.top* as (top_i, top_{r1})
4: $push(1, 0)$
5: $prev_{r1} \leftarrow 0$
6: **for** i from 2 to n **do**
7: **if** $(\mathcal{LCP}[top_i] < \mathcal{LCP}[i])$ **then** $push(i, i-1)$
8: **else if** $(\mathcal{LCP}[top_i] = \mathcal{LCP}[i])$ **then continue**
9: **else**
10: **while** $(stack \neq empty$ **and** $\mathcal{LCP}[top_i] > \mathcal{LCP}[i])$ **do**
11: $prev_{r1} \leftarrow top_{r1}$
12: PROCESSSTACK()
13: $pop()$
14: **if** $(stack = empty$ **or** $LCP[top_i] < LCP[i])$ **then**
15: $push(i, prev_{r1})$
16: **else continue**
17: **while** $(stack \neq empty)$ **do**
18: PROCESSSTACK()
19: $pop()$
20: **return** $\mathcal{OLP}$

procedure PROCESSSTACK()
$\boldsymbol{v}_{top_i} \leftarrow input[\mathcal{SA}[top_i]..\mathcal{SA}[top_i] + \mathcal{LCP}[top_i]]$
if $(\mathcal{RSF}[top_i] > 1)$
and $(\mathcal{LCP}[top_i] > 2$ **or** $(\mathcal{LCP}[top_i] = 2$ **and** $\boldsymbol{v}_{top_i}[0] = \boldsymbol{v}_{top_i}[1]))$
and MAXBORDER$(\mathbf{v}_{top_i}, \mathcal{LCP}[top_i]))$ **then**
$r_1 \leftarrow top_{r1}$
$r_m \leftarrow r_1 + \mathcal{RSF}[top_i] - 1$
Copy $\mathcal{SA}[r_1..r_m]$ to $\mathcal{SA}^*[r_1..r_m]$
Sort $\mathcal{SA}^*[r_1..r_m]$ in ascending order
$sum \leftarrow 0$
for k from 1 to $\mathcal{RSF}[top_i]$ **do**
$diff \leftarrow \mathcal{SA}^*[r_1 + k] - \mathcal{SA}^*[r_1 + k - 1]$
if $(diff < \mathcal{LCP}[top_i])$ **then**
$sum \leftarrow sum + \mathcal{LCP}[top_i] - diff$
$\mathcal{OLP}[top_i] \leftarrow sum$

FIGURE B.7: PROCESSSTACK() procedure to process the pairs $(index, r_1)$ on the stack. A near replica of Algorithm 1.

Bibliography

[1] N. Mhaskar and W. F. Smyth, "String covering with optimal covers," *Journal of Discrete Algorithms*, vol. 51, pp. 26–38, Jul. 2018. [Online]. Available: https://doi.org/10.1016/j.jda.2018.09.003

[2] G. B. Golding, H. Koponen, N. Mhaskar, and W. F. Smyth, "Computing maximal covering subsequences of protein sequences," in *Mathematical Foundations in Bioinformatics 2021 (MatBio)*, 2022, authors listed in alphabetical order by surname. Submitted with revisions.

[3] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, "MUMmer4: A fast and versatile genome alignment system," *PLOS Computational Biology*, vol. 14, no. 1, p. e1005944, Jan. 2018. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1005944

[4] S. J. Puglisi and A. Turpin, "Space-time tradeoffs for longest-common-prefix array computation," in *Proceedings of the 19th International Symposium on Algorithms and Computation*, ser. Lecture Notes in Computer Science, S.-H. Hong, H. Nagamochi, and T. Fukunaga, Eds. Springer Berlin/Heidelberg, 2008, vol. 5369, pp. 124–135. [Online]. Available: https://doi.org/10.1007/978-3-540-92182-0_14

[5] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, K. Stencel, and T. Waleń, "New simple efficient algorithms computing powers and runs in strings," *Discrete Appl. Math.*, vol. 163, p. 258–267, jan 2014. [Online]. Available: https://doi.org/10.1016/j.dam.2013.05.009

[6] W. F. Smyth, *Computing Patterns in Strings*, ser. ACM Press Bks. Boston, MA: Pearson/Addison-Wesley, Apr. 2003.

[7] H. Koponen, N. Mhaskar, and W. F. Smyth, "An overview of string processing applications to data analytics," in *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. IEEE, May 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/rdaaps48126.2021.9452004

[8] A. Apostolico and A. Ehrenfeucht, "Efficient detection of quasi–periodicities in strings," Leonadro Fibonacci Inst., Trento, Italy, Tech. Rep. 90.5, 1990.

[9] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń, "Fast algorithm for partial covers in words," *Algorithmica*, vol. 73, no. 1, p. 217–233, sep 2015. [Online]. Available: https://doi.org/10.1007/s00453-014-9915-3

[10] R. Cole, C. S. Iliopoulos, M. Mohamed, W. F. Smyth, and L. Yang, "The complexity of the minimum k-cover problem," *Journal of Automata, Languages and Combinatorics*, vol. 10, no. 5–6, pp. 641–653, 2005. [Online]. Available: https://doi.org/10.25596/jalc-2005-641

[11] T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczyński, "Enhanced string covering," *Theoretical Computer Science*, vol. 506, pp. 102–114, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397513006105

[12] N. Mhaskar and W. F. Smyth, "Frequency covers for strings," *Fundamenta Informaticae*, vol. 163, no. 3, pp. 275–289, Nov. 2018. [Online]. Available: https://doi.org/10.3233/fi-2018-1744

[13] ——, "String covering: A survey," *ACM Computing Surveys*, 2021, submitted for publication.

[14] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. [Online]. Available: https://doi.org/10.1017/CBO9780511574931

[15] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, Oct. 1993. [Online]. Available: https://doi.org/10.1137/0222058

[16] T. A. Runkler, "Introduction," in *Data Analytics*. Vieweg+Teubner Verlag, 2012, ch. 1, pp. 1–3. [Online]. Available: https://doi.org/10.1007/978-3-8348-2589-6_1

[17] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "Knowledge discovery and data mining," in *KDD-96 Proceedings*, 1996.

[18] C. Shearer, "The CRISP-DM Model: The New Blueprint for Data Mining," *Journal of Data Warehousing*, vol. 5, no. 4, pp. 13–22, 2000.

[19] IBM Corporation, "Analytics Solutions Unified Method," *Analytics services Datasheet*, 2016.

[20] J. Lu, C. Lin, J. Wang, and C. Li, "Synergy of Database Techniques and Machine Learning Models for String Similarity Search and Join," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. New York, NY, USA: ACM, nov 2019, pp. 2975–2976. [Online]. Available: https://dl.acm.org/doi/10.1145/3357384.3360319

[21] V. D. Gesù, "Data analysis and bioinformatics," in *Pattern Recognition and Machine Intelligence*, A. Ghosh, R. K. De, and S. K. Pal, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 373–388. [Online]. Available: https://doi.org/10.1007/978-3-540-77046-6_47

[22] C. S. Greene, J. Tan, M. Ung, J. H. Moore, and C. Cheng, “Big data bioinformatics,” *Journal of Cellular Physiology*, vol. 229, no. 12, pp. 1896–1900, Aug. 2014. [Online]. Available: https://doi.org/10.1002/jcp.24662

[23] Z. Yin, H. Lan, G. Tan, M. Lu, A. V. Vasilakos, and W. Liu, “Computing platforms for big biological data analytics: Perspectives and challenges,” *Computational and Structural Biotechnology Journal*, vol. 15, pp. 403–411, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2001037017300351

[24] S. Hazelhurst and Z. Lipták, “KABOOM! A new suffix array based algorithm for clustering expression data,” *Bioinformatics*, vol. 27, no. 24, pp. 3348–3355, 10 2011. [Online]. Available: https://doi.org/10.1093/bioinformatics/btr560

[25] B. Haubold and T. Wiehe, “How repetitive are genomes?” *BMC Bioinformatics*, vol. 7, no. 1, Dec. 2006. [Online]. Available: https://doi.org/10.1186/1471-2105-7-541

[26] A. P. J. de Koning, W. Gu, T. A. Castoe, M. A. Batzer, and D. D. Pollock, “Repetitive elements may comprise over two-thirds of the human genome,” *PLoS Genetics*, vol. 7, no. 12, p. e1002384, Dec. 2011. [Online]. Available: https://doi.org/10.1371/journal.pgen.1002384

[27] J. W. Pelley, “Organization, synthesis, and repair of DNA,” in *Elsevier's Integrated Review Biochemistry*, 2nd ed. Elsevier, 2012, pp. 125–135. [Online]. Available: https://doi.org/10.1016/b978-0-323-07446-9.00015-5

[28] M. F. Antolin and W. C. Black, “Genes, description of,” in *Encyclopedia of Biodiversity*. Elsevier, 2001, pp. 654–661. [Online]. Available: https://doi.org/10.1016/b978-0-12-384719-5.00063-0

[29] R. J. Trent, “Genes to personalized medicine,” in *Molecular Medicine*, 4th ed. Elsevier, 2012, pp. 1–37. [Online]. Available: https://doi.org/10.1016/b978-0-12-381451-7.00001-3

[30] B. D. Pickett, S. M. Karlinsey, C. E. Penrod, M. J. Cormier, M. T. W. Ebbert, D. K. Shiozawa, C. J. Whipple, and P. G. Ridge, “SA-SSR: a suffix array-based algorithm for exhaustive and efficient SSR discovery in large genetic sequences: Table 1.” *Bioinformatics*, vol. 32, no. 17, pp. 2707–2709, May 2016. [Online]. Available: https://doi.org/10.1093/bioinformatics/btw298

[31] H. Mathkour and M. Ahmad, “A pattern matching technique for multiple sequences alignment with GAP consideration,” in *2009 International Conference on Signal Acquisition and Processing*. IEEE, Apr. 2009, pp. 123–127. [Online]. Available: https://doi.org/10.1109/icsap.2009.35

[32] A. D. Prjibelski, A. I. Korobeynikov, and A. L. Lapidus, “Sequence analysis,” in *Encyclopedia of Bioinformatics and Computational Biology*, S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, Eds. Oxford:

Academic Press, 2019, vol. 3, pp. 292–322. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128096338201064

[33] T. Cheng and X. Zhan, "Pattern recognition for predictive, preventive, and personalized medicine in cancer," *EPMA Journal*, vol. 8, no. 1, pp. 51–60, Mar. 2017. [Online]. Available: https://doi.org/10.1007/s13167-017-0083-9

[34] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.

[35] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 390–398.

[36] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, May 2009. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp324

[37] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, May 2010. [Online]. Available: https://doi.org/10.1093/bib/bbq015

[38] J. Zhang, H. Lin, P. Balaji, and W.-C. Feng, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 377–384. [Online]. Available: https://doi.org/10.1109/ccgrid.2013.67

[39] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, Mar. 2012. [Online]. Available: https://doi.org/10.1038/nmeth.1923

[40] D. Kim, B. Langmead, and S. L. Salzberg, "HISAT: a fast spliced aligner with low memory requirements," *Nature Methods*, vol. 12, no. 4, pp. 357–360, Mar. 2015. [Online]. Available: https://doi.org/10.1038/nmeth.3317

[41] ISO and IEC, *Iso/Iec 27002:2005(E)*. International Organization for Standardization, 2005. [Online]. Available: www.iso.org

[42] B. Pinkas and T. Sander, "Securing passwords against dictionary attacks," in *Proceedings of the 9th ACM conference on Computer and communications security - CCS '02*. New York, New York, USA: ACM Press, 2002, p. 161. [Online]. Available: https://doi.org/10.1145/586110.586133

[43] WSWAS., *Social networks as a platform for distributed dictionary attack*. [S. l.]: WSEAS, 2011.

[44] F. L. Bauer, "Anatomy of Language: Patterns," in *Decrypted Secrets*, 4th ed. Springer Berlin Heidelberg, 2007, ch. 13. [Online]. Available: https://doi.org/10.1007/978-3-540-48121-8

[45] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, Mar. 2004. [Online]. Available: https://doi.org/10.1016/s1570-8667(03)00065-0

[46] F. W. Kasiski, *Die Geheimschriften und die Dechiffrir-Kunst: Mit besonderer Berücksichtigung der deutschen und der französischen Sprache.* Mittler, 1863.

[47] W. C. Arnold, D. M. Chess, J. O. Kephart, G. B. Sorkin, and S. R. White, “Searching for patterns in encrypted data,” U.S. Patent 19, 1994. [Online]. Available: https://patents.google.com/patent/US5442699

[48] P. Velan, M. Čermák, P. Čeleda, and M. Drašar, “A survey of methods for encrypted traffic classification and analysis,” *International Journal of Network Management*, vol. 25, no. 5, pp. 355–374, Jul. 2015. [Online]. Available: https://doi.org/10.1002/nem.1901

[49] S. Rosset and A. Inger, “Kdd-cup 99: Knowledge discovery in a charitable organization’s donor database,” *SIGKDD Explorations Newsletter*, vol. 1, no. 2, p. 85–90, jan 2000. [Online]. Available: https://doi.org/10.1145/846183.846204

[50] P. Suresh, R. Sukumar, and S. Ayyasamy, “Efficient pattern matching algorithm for security and binary search tree (BST) based memory system in wireless intrusion detection system (WIDS),” *Computer Communications*, vol. 151, pp. 111–118, Feb. 2020. [Online]. Available: https://doi.org/10.1016/j.comcom.2019.11.035

[51] R. B. Gadde, D. Adjeroh, and A. Ross, “Indexing iris images using the burrows-wheeler transform,” in *2010 IEEE International Workshop on Information Forensics and Security.* IEEE, Dec. 2010, pp. 1–6. [Online]. Available: https://doi.org/10.1109/wifs.2010.5711467

[52] A. Pradhan, N. Pati, S. Rup, and A. S. Panda, “A modified framework for image compression using burrows-wheeler transform,” in *2016 2nd International Conference on Computational Intelligence and Networks (CINE).* IEEE, Jan. 2016. [Online]. Available: https://doi.org/10.1109/cine.2016.33

[53] W. Chen and Y. Gao, “Face recognition using ensemble string matching,” *IEEE Transactions on Image Processing*, vol. 22, no. 12, pp. 4798–4808, Dec. 2013. [Online]. Available: https://doi.org/10.1109/tip.2013.2277920

[54] J.-W. Hsieh, Y.-T. Hsu, H.-Y. Liao, and C.-C. Chen, “Video-based human movement analysis and its application to surveillance systems,” *IEEE Transactions on Multimedia*, vol. 10, no. 3, pp. 372–384, Apr. 2008. [Online]. Available: https://doi.org/10.1109/tmm.2008.917403

[55] H. Ghasemzadeh, V. Loseu, and R. Jafari, “Structural action recognition in body sensor networks: Distributed classification based on string matching,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 2, pp. 425–435, Mar. 2010. [Online]. Available: https://doi.org/10.1109/titb.2009.2036722

[56] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, Oct. 2011. [Online]. Available: https://doi.org/10.1109/tc.2010.188

[57] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta, "The "runs" theorem," *SIAM Journal on Computing*, vol. 46, no. 5, pp. 1501–1514, Jan. 2017. [Online]. Available: https://doi.org/10.1137/15m1011032

[58] M.-L. Bang, T. Centner, F. Fornoff, A. J. Geach, M. Gotthardt, M. McNabb, C. C. Witt, D. Labeit, C. C. Gregorio, H. Granzier, and S. Labeit, "The complete gene sequence of titin, expression of an unusual ≈700-kDa titin isoform, and its interaction with obscurin identify a novel z-line to i-band linking system," *Circulation Research*, vol. 89, no. 11, pp. 1065–1072, Nov. 2001. [Online]. Available: https://doi.org/10.1161/hh2301.100981