

A KMER-BASED PARALLEL ALGORITHM FOR PATTERN SEARCHING IN DNA SEQUENCES ON SHARED-MEMORY MODEL

BY

KANIWA FREESON

Reg. No: 14101181

BSc (Computer Science) (UZ), MSc (Computer Science) (UZ)

Department of Computer Science and Information Systems,
College of Sciences,

Botswana International University of Science and Technology

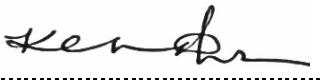
freeson.kaniwa@studentmail.biust.ac.bw

A Thesis Submitted to the College of Science in Partial Fulfilment of the Requirements for
the Award of the Degree of Doctor of Philosophy in Computer Science of BIUST

Supervisor: Dr. Venu. M. Kuthadi

Department of Computer Science & Info Sys,
College of Sciences, BIUST

kuthadiv@biust.ac.bw, (+267) 4931104

Signature: 

Date: 03/09/2018

Co-Supervisor: Dr. Othapile Dinakennyane

Department of Computer Science & Info Sys,
College of Sciences, BIUST

dinakenyaneo@biust.ac.bw, (+267) 4931108

Signature: 

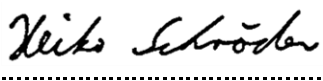
Date: 03/09/2018

Co-Supervisor: Prof. Heiko Schroeder

Berlin,

Germany,

heikoschro@gmail.com

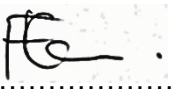
Signature: 

Date: 02/09/2018

August, 2018

DECLARATION

I, **FREESON KANIWA**, declare that this is my original work and that it has not been presented and will not be presented to any other university for a similar or any other degree award.

Signature: 

COPYRIGHT

This dissertation is copyright material protected under the Berne Convention, the Copyright Act of 1999 and other international and national enactments, in that behalf, on intellectual property. It must not be reproduced by any means, in full or in part, except for short extracts in fair dealing; for researcher private study, critical scholarly review or discourse with an acknowledgement, without the written permission of the office of the DVCAA, on behalf of both the author and the BIUST.

CERTIFICATION

The undersigned certifies that s(he) has read and hereby recommends for acceptance by the College of Science a thesis titled: **A KMER-BASED PARALLEL ALGORITHM FOR PATTERN SEARCHING IN DNA SEQUENCES ON SHARED-MEMORY MODEL** in fulfilment of the requirements for the degree of Doctor of Philosophy (in Computer Science) of the BIUST.



Dr. V. M. Kuthadi
(Supervisor)

Date: 03.09.2018



Dr. O. Dinakenyane
(Co-Supervisor)

Date: 03/09/2018



Prof. Heiko Schroeder
(Co-Supervisor)

Date: 02/09/2018

ACKNOWLEDGEMENTS

I am grateful to my supervisors, Dr. Venu Madhav Kuthadi, Dr. Otlhapile Dinakenyane and Professor Heiko Schroeder for their patience and expert guidance during the course of my studies. Their commitment, experience, work ethic played a major role in the successful completion of this thesis.

I would like to thank the Botswana International University of Science and Technology (BIUST) for sponsoring my doctoral studies. I would also like to thank my colleagues, with specific mention to Sethunya Roise Joseph for additional insight and support throughout my studies. I am also thankful to the IT Department of BIUST for providing the HPZ820 workstation for the test runs.

Lastly, it would not have been possible to accomplish the studies without the support from my family members and friends, whose constant encouragement gave strength to reach this stage. Thank you Lord for your grace.

DEDICATION

This thesis is dedicated to my parents and family; you have always been my inspiration.

ABSTRACT

The explosive growth of biological sequences over the last decade has consequently led to a lot of research on effective techniques for storing and analysing this big data. Generating Deoxyribonucleic Acid (DNA) sequences has become more reliable, faster, easier and cheaper due to latest sequencing technology leading to big data sequences. This poses a challenge to efficient analysis to such massive sequences. Advanced indexing algorithmic techniques are required to facilitate efficient analysis. A particular interesting indexing data structure for such analysis is the Suffix Tree. Search times for patterns have linear times, however longer construction times and high memory requirements make it prohibitive to use this data structure for very long sequences such as DNA sequences. To index these long sequences, several algorithms have been explored in recent years. These algorithms all target to improve running time and memory consumption for the Suffix Tree. That is, being able to index a long sequence by making it fit in the main memory and being able to look for patterns in reasonable times. DNA sequences carry basic information for each human being. Searching for patterns in DNA sequences has been shown to have biological relevance since patterns signify some underlying biological function such as disease detection, genetic variation, and regulation, etc. This thesis explores the use of truncating the generalised Suffix Tree basing on the details of the input, that is, the size of the repeat to be searched. This is aimed at improving the memory requirements. In this thesis, parallelisation of the Suffix Tree data structure is further explored to reduce the construction times. Parallelisation has become of more interest since parallel architectures are now ubiquitous in standard desktop computers. Based on the theoretical and experimental findings, we can conclude that specification of pattern size, prior to Suffix Tree construction by truncating, reduces the memory requirements of the generalised Suffix Tree. In conclusion, the novel alphabet dependant parallelisation algorithm achieves a speedup of 15x over the existing state of art sequential algorithm. Despite the success of our Parallel algorithm for Suffix Tree (PaST) algorithm, its main limitation is that of specification of pattern size prior to construction or searching. Other than that, comparatively our algorithm shows improved performance in terms of memory and time requirements.

Table of Contents

Chapter 1	1
Introduction	1
1.0 Background	1
1.1 Motivation	2
1.2 Statement of the problem and research objectives	3
1.2.1 Statement of the problem.....	3
1.2.2 General Objective	3
1.2.3 Specific Objectives	4
1.3 Hypothesis, research questions, rationale and contributions.....	4
1.3.1 Hypotheses.....	4
1.3.2 Research Questions.....	5
1.3.3 Contributions	6
1.3.4 Significance of the Study.....	7
1.4 Thesis structure	8
Chapter 2.....	10
Background of the Problem.....	10
2.0 Introduction	10
2.1 DNA Sequencing.....	10
2.2 Repeats, Patterns and Suffix Tree related data structures	11
2.2.1 Repeats.....	11
2.2.2 Patterns	11
2.2.3 Suffix Tree, Generalized Suffix Tree, and KMER-based Suffix Tree	12
2.2.4 Search and Construction Algorithms.....	13
2.3 Applications of Generalized Suffix Trees.....	13
2.3.1 Longest common substring.....	14
2.3.2 Genome projects	15
2.3.3 DNA contamination.....	15
2.3.4 Plagiarism detection	15
2.3.5 Code duplication.....	15
2.3.6 Information retrieval.....	16
2.3 Suffix Tree Implementation Issues	16
2.4.1 Alphabet Size.....	16
2.4.2 Storage Considerations	16
2.4 Computation Models.....	16

2.5.1 Turing Machine	17
2.5.2 Random Access Machine (RAM) Model	17
2.5.3 Parallel Random Access Machine Model (PRAM).....	18
2.6 Evaluation Methods.....	19
2.6.1 Theoretical Analysis	19
2.6.2 Experimental Analysis.....	23
2.7 Summary	23
Chapter 3.....	25
Literature Review.....	25
3.0 Introduction	25
3.1 Types of repeats	25
3.2 Data Structures	27
3.2.1 Suffix Trie.....	28
3.2.2 Suffix Tree.....	29
3.2.3 Generalized Suffix Tree.....	30
3.2.4 Suffix Array (SA)	31
3.2.5 Position Heaps	32
3.2.6 Longest Common Prefix (LCP) Array	33
3.2.7 Full –text index in Minute space (FM Index).....	34
3.2.8 Wavelet Tree.....	35
3.3 Techniques	36
3.3.1 Lempel - Ziv factorization (LZ)	36
3.3.2 Burrows-Wheeler Transform (BWT)	37
3.3.3 Huffman coding	38
3.3.4 Parallel Algorithms.....	38
3.4 Ukkonen’s Algorithm.....	40
3.5 Algorithms.....	43
3.5.1 Suffix Tree Algorithms.....	43
3.5.2 KMER-based Suffix Algorithms	46
3.5.3 Parallel programming	49
3.5.4 Parallel algorithms in Suffix Trees algorithms.....	50
3.6 Summary	54
Chapter 4.....	55
KMER algorithm.....	55
4.0 Introduction	55

4.1 Preliminaries	55
4.2 Sliding window design	56
4.3 KMER algorithm Design.....	58
4.4 Evaluation strategy	64
4.5 Summary.....	67
Chapter 5.....	68
PaST algorithm.....	68
5.0 Introduction	68
5.1 Preliminaries.....	68
5.2 PaST algorithm design.....	70
5.4 Evaluation Analysis.....	75
5.5 Summary.....	76
Chapter 6.....	77
Results and Discussion.....	77
6.0 Introduction	77
6.1 Theoretical Evaluation.....	77
6.2 Experimental Evaluation	81
6.3 Summary.....	92
Chapter 7.....	93
Conclusions and Recommendations.....	93
7.0 Introduction	93
7.1 Challenges and Limitations	93
7.2 Conclusions and Future research.....	93
References.....	96
Appendix A.....	101
KSA algorithm	101
Appendix B.....	103
PaST algorithm.....	103
Appendix C	106
Step by step construction using Ukkonen Algorithm.....	106
Appendix D.....	107
Sample Screenshot – KSA Algorithm.....	107
Appendix E	108
Sample Screenshot – PaST Algorithm	108
Appendix F.....	109

Sample Screenshot – Parallel Processing.....	109
--	-----

Table of Figures

Figure 1.1 Falling costs to sequence a genome in USD since 2001 [10]	2
Figure 2.1 Structure of a DNA [24]	10
Figure 2.2 Output form GST in polynomial time.....	14
Figure 2.3 Alan Turing Machine.....	17
Figure 2.4 The RAM	18
Figure 2.5 PRAM Architecture	18
Figure 2.6 Executing a loop whilst maintaining the loop invariant in a flow chart	21
Figure 3.1 Two repeats aligned, M is a Miss (mismatch) and H is a Hit (match)	27
Figure 3.2 Suffix Trie for S1	29
Figure 3.3 The Suffix Tree.....	30
Figure 3.4 Generalised Suffix Tree for string S1 and S2	31
Figure 3.5 Inserting suffix S into the position heap	33
Figure 3.6 Ascertaining maximal repeats.....	34
Figure 3.7 Wavelet Tree.....	36
Figure 3.8 Burrows-Wheeler Matrix.....	37
Figure 3.9 Suffix Tree with suffix links for string cacao	41
Figure 3.10 Ukkonen's Algorithm.....	42
Figure 3.11 Illustration of Skip-count trick process.....	43
Figure 3.12 (a) Suffix Tree (b) 3-TST of abaabaa# [17].....	47
Figure 3.13 Construction of the generalised kTST [18]	48
Figure 3.14 k-Factor tree extraction of substrings [82].....	49
Figure 3.15 MapReduce Operation [1]	50
Figure 3.16 Prefix Partitioning.....	52
Figure 3.17 Sub-tree for S-prefix T and sample contextual suffix link [58].....	52
Figure 3.18 A workflow of data dependencies of PCF [92]	53
Figure 4.1 Construction process for the Suffix Tree [70]	56
Figure 4.2 Explicit Suffix Tree of δ	56
Figure 4.3 Sliding window method	57
Figure 4.4 KMER Suffix Tree of S	59
Figure 4.5 Complete Suffix Tree.....	60
Figure 4.6 Sliding windows technique illustration as fixed substrings.....	60
Figure 4.7 Part a) Standard Suffix Tree vs. Part b) KMER-based Suffix Tree.....	62

Figure 4.8 The KSA Algorithm	63
Figure 4.9 Design summary	64
Figure 5.1 Shared Memory Model	70
Figure 5.2 Generating sets.....	71
Figure 5.3 Terminal Symbols are inserted in the final phase.....	71
Figure 5.4 Branch-wise Parallel Construction of Suffix Tree of S	72
Figure 5.5 Merged Complete Suffix Tree (Example 5.1)	72
Figure 5.6 Suffix Tree of {abaabc} with $ p = 3$	73
Figure 5.7 The PaST Algorithm.....	74
Figure 5.8 PaST Illustration	75
Figure 6.1 Space usage comparison on Rice sequences.....	84
Figure 6.2 Memory usage on Protein sequences.....	85
Figure 6.3 Construction time comparison on DNA sequences for pattern size of 5.....	87
Figure 6.4 Construction time comparison on DNA sequences for pattern size of 10.....	88
Figure 6.5 Construction time comparison on DNA sequences for pattern size of 15.....	88
Figure 6.6 Speedup relative to text size	92

List of Tables

Table 3.1 Suffix Array and LCP Array.....	32
Table 2. Indices for the S	33
Table 4.1 Characteristics of dataset	67
Table 6.1 Space comparison between ST-based algorithm [2] and KSA algorithm	82
Table 6.2 Space usage of for complete chromosomes and the whole genome.....	82
Table 6.3 Time cost for complete chromosomes and the whole genome.....	83
Table 6.4 kTST algorithm results [18]	86
Table 6.5 TST algorithm results [17].....	86
Table 6.6 Construction times of KSA vs. ST- Based algorithm.....	89
Table 6.7 PaST construction times for various genomic sequences.....	90
Table 6.8 Search times for various repeat sizes.....	90

List of Abbreviations

BWT	Burrows-Wheeler Transform
CRCW	Concurrent Read Concurrent Write
CREW	Concurrent Read Exclusive Write
CST	Compressed Suffix Tree
DNA	DeoxyriboNucleic Acid
ERCW	Exclusive Read Concurrent Write
EREW	Exclusive Read Exclusive Write
FM Index	Full-text index in Minute space
GST	Generalised Suffix Tree
KSA	KMER-based Suffix tree Algorithm
LCP	Longest Common Prefix
LCS	Longest Common Substring
LZ	Lempel-Ziv
PaST	Parallel Algorithm for Suffix Tree
PC	Personal Computer
PRAM	Parallel Random Access Machine
RAM	Random Access Machine
SA	Suffix Array
ST	Suffix Tree
TST	Truncated Suffix Tree

Chapter 1

Introduction

1.0 Background

In today's data-driven world where data sizes, such as DNA sequences, continue to increase daily, performance in data analysis has become increasingly more important than ever before. The amount of data generated is increasing daily and is close to the order of zeta bytes [1]. Improving running time of programs has been proved to increase workers' productivity, as well as user experience. In the past years, high performance computing solutions were only available to small communities since those solutions were more expensive and available in specialised computing environments. However, in the past decade, multicore architectures have become prevalent. Multicore technology has become widespread due to the physical limitations of hardware in transistor power density hence Moore's Law [2] no longer continues to hold. Therefore a shift in the technology of processors became inevitable and has been referred to as the "multicore revolution" [3]. Multicore technology is everywhere nowadays, and most personal computers (PCs) including cellular phones contain multiple cores. Therefore designing parallel algorithms that utilise all processors on a computer becomes important to gain scalable performance thereby allowing huge amounts of data to be processed efficiently [4].

DNA sequences carry basic information for each human being. A complete DNA sequence consists of many characters, approximately up to 3 billion nucleotides. A fast and accurate DNA analysis in these huge DNA sequences is one of the basic algorithm design problems. DNA sequencing involves figuring out the precise order of the sequence of the DNA. The order is based on four DNA bases. These DNA bases of the human genome are Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). Generating DNA sequences has become easy and cheaper due to current sequencing technology which has suddenly made DNA sequences easily available [5, 6]. Hence the missing link in this new exciting development is the absence of efficient analysis tools to such long sequences [7]. Figure 1.1 shows the trend since 2001 for the cost of sequencing genomes. Availability of genomes was further facilitated by the *1000 Genomes Project* which aimed at sequencing and cataloguing 1000 complete human genomes of various ethnic groups [8].

Furthermore also the recent *100,000 Genomes Project*, which targeted to sequence 100 000 complete genomes from individuals, thereby making this information publicly available for analysis [9].

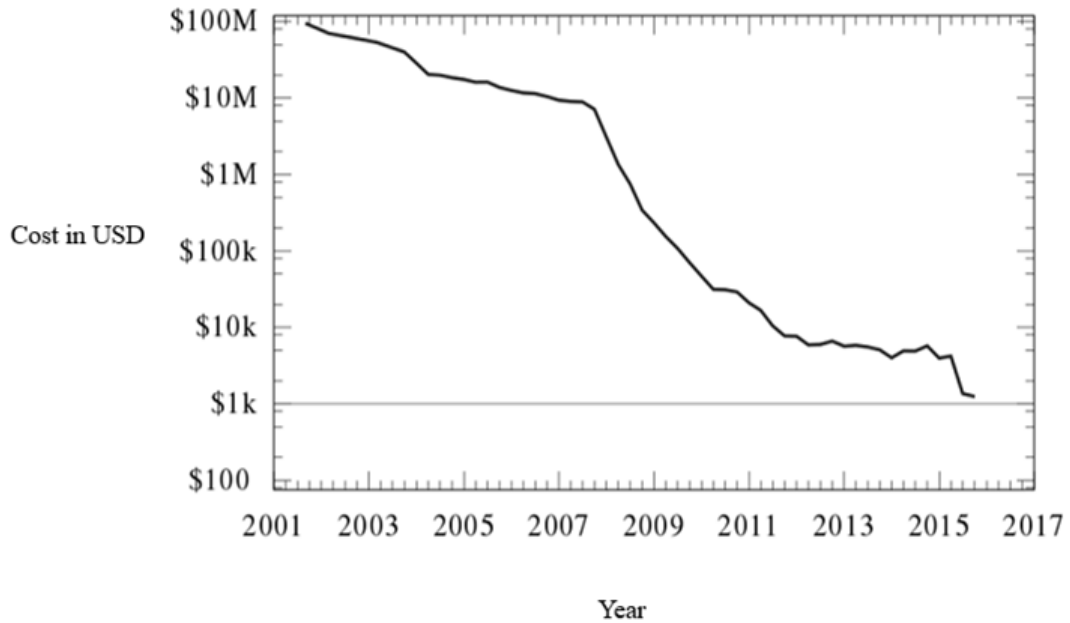


Figure 1.1 Falling costs to sequence a genome in USD since 2001 [10]

This research has the potential to facilitate early disease detection, mutation history and other various forensic applications like DNA typing in crime investigation. This can also allow us to better fight life threatening diseases such as cancer, diabetes and so forth [11]. An important shift from reactive to predictive medicine can then be possibly realised. The existing algorithms are prohibitive on their usability on huge DNA sequences because of their poor performance in analysis where some take several days to complete [7].

1.1 Motivation

DNA analysis is done in a number of ways, one of which is pattern searching. This thesis focuses on pattern searching in DNA sequences. Commonly, this process of pattern searching is being successfully conducted on super computers since it requires intensive computing power [12, 13]. Super computers are increasingly being used for other various DNA sequence analysis due to the computing intensive process of the DNA analysis [14, 15]. This study, therefore, presents an opportunity of this process to be carried out on standard workstation computers where the performance has been noted to be slow and

unreasonable, in some cases taking several days to complete the suffix tree construction before pattern searching even begins. The current algorithms are slow and have not yet gained acceptance to be adopted as mainstream tools for this DNA pattern searching process [7, 16].

Currently there is no Suffix Tree algorithm which can run the analysis within acceptable time unless on a powerful supercomputer. To the best of our knowledge, there is no parallel version of the idea of KMER generalised Suffix Tree construction [17, 18]. It remains a continuous challenge to design algorithms which are efficient in memory usage and processing time requirements when dealing with large amounts of information, therefore an efficient algorithm is required. Beller et al. [19] emphasises the need for new and better approaches to this sudden increase of DNA sequences. In general, storage, analysis and mining of the DNA sequences are considered to be serious bottlenecks to genome projects [20].

1.2 Statement of the problem and research objectives

The statement of the problem, research goal and objectives are discussed in the following section.

1.2.1 Statement of the problem

Designing construction algorithms which are fast and memory efficient when searching for patterns in the entire human genome has always been a challenge and research focus in computer science. The challenge remains, since most of the current algorithms have been proven to fall short on scaling, given the huge size of the entire human genomes [21]. This study, therefore, addresses the problem of an improved construction algorithm for searching of patterns in DNA sequences, which is both time and space-efficient.

1.2.2 General Objective

The general objective of this study is to design an improved algorithm that is time and space-efficient for pattern searching in DNA sequences based on the sliding window technique on KMER generalized Suffix Tree and the parallelization of the KMER generalized Suffix Tree.

1.2.3 Specific Objectives

In order to achieve the main goal of this research study, the following objectives were explored, namely, to:

1. Identify limitations of current Suffix Tree-based algorithms.
2. Design a space-efficient algorithm for the generalized Suffix Tree construction using the sliding window technique.
3. Design a parallel algorithm for KMER-based generalized Suffix Tree construction.
4. Evaluate the proposed algorithms using theoretical and experimental analysis.

1.3 Hypothesis, research questions, rationale and contributions

The hypothesis, research questions, justification and expected contributions are discussed below.

1.3.1 Hypotheses

The main research question of this thesis is therefore, *How can we improve performance of algorithms used in DNA analysis in a space and time-efficient way on a standard workstation computer?* This means we need to design an algorithm which can run in reasonable time with less space during construction of the generalized Suffix Tree data structure. Following from this question, we setup the hypothesis of this thesis on performance and parallelism.

Null hypothesis (H_0): The best performance of Suffix Tree construction time takes about four days, with linear time complexity excluding the repeat finding process on a 32 GB-memory machine for the entire human DNA [16]. ($H_0: \mu_t \geq 4$ days).

Alternate hypothesis (H_1): The performance of the generalised Suffix Tree construction can be less than four days if parallelisation is applied to the sequential generalised Suffix Tree construction. ($H_1: \mu_t < 4$ days).

This means the proposed algorithm can reject the null hypothesis when parallelisation is employed, and this can be also verified by Amdahl's law [22]. The alternative hypothesis is based on exploring the fact that, if we apply parallelisation to the construction of generalised Suffix Tree, then construction time requirements of the proposed algorithm

will be reduced. In this thesis, reduction of the space requirements of our proposed algorithm is explored by constructing the generalised Suffix Tree based on the truncated sliding window using KMER technique.

1.3.2 Research Questions

As a means of providing a solution to challenges faced by repeat finding algorithms in DNA sequences, the research attempted to answer or address the following questions:

1. How can Suffix Tree-based algorithm(s) in DNA sequences be improved in a space and time-efficient manner?
2. How can the ubiquitous multicore architecture be best utilised to improve the running time of pattern searching algorithm(s)?
3. To what extent can the memory requirements be improved by truncating the Suffix Tree using the KMER technique?

The first research question assists in investigating how Suffix Tree based algorithms can be improved in terms of space consumption and time requirements. This is because the Suffix Tree-based data structures have been shown to have a huge drawback on memory consumption during construction and longer running times, making it difficult to be used as mainstream tools. Once the construction is complete, the Suffix Tree algorithms are quicker to search or mine patterns yielding an $O(p)$ time where p represents the size of the pattern to be searched. The research objective number two is meant to address this particular research question by employing the sliding window technique to design a space-efficient algorithm.

The second research question explores the use of a multicore architecture to improve the execution time of the searching process in the pattern searching algorithms. The question is based on the idea that, the multicore architecture has become ubiquitous, hence the design of the algorithm should likewise take advantage of this, to improve the time requirements of the algorithm based on Amdahl's Law. This calls for a redesign of how serial algorithms of pattern searching are constructed to a new model, taking into consideration usage of multiple processors to reduce the running time of the algorithm. This particular research question is addressed by research objective three. The research objective answers this question by exploring the use of parallelization to reduce running

of the algorithm. This will entail finding parallelization techniques and opportunities for parallelism to the serial algorithms of pattern searching using the Suffix Tree-based data structure.

The last research question explores the idea of truncating the tree size to reduce the memory consumption of the Suffix Tree construction algorithms, yet being able to find the pattern to be searched. The research question further explores any limitations regarding this solution. The first and second research objectives solicit answers to the research question by use of the KMER algorithmic technique to reduce the space consumption of the algorithm.

1.3.3 Contributions

The following serve as contributions from this thesis to the academic and research community in algorithm design and analysis;

1. Current status of research trends in DNA analysis using various data structures and academia (string algorithms) through a research paper.
2. KMER-based Suffix Tree serial algorithm that can be used for LZ77 compression
3. A novel parallel algorithm for the KMER-based generalised Suffix Tree that:
 1. Runs faster with less space requirements on standard workstation computers.
 2. Can be used in compression algorithms.
 3. Sheds more insight in modern bioengineering especially in Bioinformatics.
4. Software tools (KSA and PaST implementations), which are ready for public use for DNA pattern identification on a standard workstation computer.

The area of designing efficient algorithms for pattern searching in DNA sequences is currently an intensive area of research, the survey paper covers and reviews the current research directions and techniques used in different data structures and algorithms in pattern searching and ultimately advice on the most promising data structure.

The parallel algorithm, as explained in Chapter 5, is intended to decrease the running time of the generalised Suffix Tree construction as well as using as less space as possible during its construction. The algorithm introduces new ways of thinking since it is a novel way of parallelising the KMER-based generalised Suffix Tree.

Lastly, a by-product of the algorithm, which is a software tool built upon the parallel algorithm, is ready to be used for its function of pattern identification.

1.3.3.1 Previous work by the author

This section shows the work related to this thesis, which was previously published. Most of the work outlined in this thesis has been previously published and is directly related to this thesis as follows:

1. F. Kaniwa, H. Schroeder, and O. Dinakenyane, "Repeat Finding Techniques, Data Structures and Algorithms in DNA sequences: A Survey," *Advances in Computer Science: an International Journal*, vol. 4, pp. 41-49, 2015.
2. F. Kaniwa, K.V. Madhav, O. Dinakenyane and H. Schroeder, "Space-efficient KMER algorithm for the Generalised Suffix Tree," *International Journal of Information Technology Convergence and Services*, Vol.7, No.1, 2017
3. F. Kaniwa, K.V. Madhav, O. Dinakenyane and H. Schroeder " Alphabet dependent Parallel Algorithm for Suffix Tree Construction for Pattern Searching, " *International Journal of Grid and Distributed Computing*, Vol. 10, No. 1, pp.9-20, 2017

1.3.4 Significance of the Study

Pattern searching is a very powerful and useful exercise in DNA sequences. The remarkable sudden increase in the cheap availability of DNA sequences has stimulated the study of many pattern searching algorithms and has consequently generated much research interest. The current research is necessary because of the potential immense benefits brought about by the results of DNA analysis such as early disease detection, and more insight into our evolutionary past. Multicore architectures have also become ubiquitous in today's standard personal computers, therefore parallel algorithms are the

most suitable approach to be explored towards improving efficiency in these algorithms [23]. The rapid and sudden growth in sequencing technologies has allowed us access to new DNA sequences, thereby giving us the opportunity to discover genetic material analysis rigorously.

Recent studies have shown that current algorithms are still relatively slower to be accepted as mainstream for DNA repeat analysis, hence this continued effort to improve the performance of those algorithms [7]. According to current knowledge, there are no parallel algorithms for the KMER-based generalised Suffix Tree, and can run on a standard workstation computer, hence the need for this design of the novel parallel algorithm.

1.4 Thesis structure

This thesis is presented in the following manner:

Chapter Two discusses the background of the study. It covers theoretical concepts used in the study. It introduces concepts of repeats, and patterns, to name some. Of particular importance is the discussion of the various variants of the Suffix Tree data structure and their differences. This includes the generalized Suffix Tree, the KMER-based tree and the standard Suffix Tree. This background can be carefully skipped by readers already conversant with patterns and Suffix Trees. It also serves as a useful starting point for those new to this type of study.

Chapter Three covers the literature review of the study. The chapter discusses different types of repeats from literature some in detail. Various data structures other than the Suffix Tree, which are being used in the area of indexing biological sequences are presented. The data structures include, but are not limited to Suffix Trie, Suffix Tree, Suffix Array, and Wavelet Tree. The next section covers techniques used in literature to mitigate the problem of time and memory requirements. These include the Burrows-Wheeler Transform, Huffman Coding and Parallel Algorithms. Lastly the chapter covers specific algorithms used for identification of repeats or patterns from DNA sequences.

Chapter Four presents the memory efficient KSA algorithm aimed at reducing the memory consumption of the KMER-based generalized Suffix Tree using a modified

sliding window technique. Before the presentation of the algorithm preliminaries specific to this algorithm are discussed. Finally, the evaluation strategy is provided.

Chapter Five provides the novel time-efficient PaST algorithm aimed at reducing the time requirements by parallelisation of the KSA algorithm. Lemma and its mathematical proof is provided on the observation of the parallelisation technique used.

Chapter Six presents and discusses the results. Theoretical and Experimental evaluation are conducted. Under theoretical evaluation, we provided formal proofs for correctness of the algorithms and complexity analysis and made comparisons with other algorithms in that area. Experimental results are also presented and discussed.

Chapter Seven concludes and recommends some research directions that could benefit from further research. Lastly, limitations to the study are provided.

Chapter 2

Background of the Problem

2.0 Introduction

This section briefly discusses the building blocks of concepts used in the area of DNA analysis. The focus of this research is to propose a variant of the generalised Suffix Tree to improve pattern identification in DNA sequences.

2.1 DNA Sequencing

DNA sequencing allows genes to be identified basing on their DNA sequences. The sequencing is useful for availing the DNA sequences used to determine relationships. For instance, the DNA of a human being can be compared to that of a chimpanzee to establish the relationship between them. DNA sequencing is also useful for other applications like crime forensics, disease diagnosis, and so on [24]. Biological researches and discovery have been accelerated by the advent of the DNA sequencing technologies. The sequencing of human genome has increasingly become cheaper and fast due to the rapid speed of modern DNA sequencing technology. Collaborations among research organisations and continents have made it possible to generate complete DNA of various animals, plants and so on [25]. The DNA structure is shown in Figure 2.2



Figure 2.1 Structure of a DNA [24]

DNA sequencing generates DNA sequences from a specific organism into a format that is workable by scientists and researchers thereby producing various ways of analysing it, in a ‘dry lab’ fashion. The cost of sequencing has reduced significantly and this is attributed to the advent of next generation sequencing technologies [26]. KMER is refers

to all possible subsequences or substrings of size k in a particular string or sequence generated from the DNA sequencing process.

2.2 Repeats, Patterns and Suffix Tree related data structures

This section covers various types of repeats and patterns. Selected related data structures are also discussed together with their specific differences.

2.2.1 Repeats

A repeat is a substring which occurs multiple times within the parent string, as shown in Example 2.1. The Substring ‘AAA’ is repeated three times in String S. Repeats are sometimes referred to as repeated patterns (More on patterns will be explained in Section 2.1.2). Repeats account for approximately half of the entire human genome sequence. Identification of these repeats is very useful since they have biological relevance to early disease diagnosis, hence the growing popularity of repeat finding algorithms [27].

Example 2.1

ACGTAAACGATTAGAAAGGAAATTTTTT
←—————|—————|—————|—————|—————→

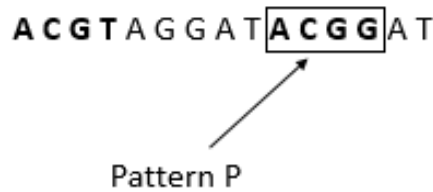
2.2.2 Patterns

A search pattern is a substring to be searched for its existence in a string. Its occurrence might be non-existent, or exists once or more than once. For example, given the following:

Example 2.2 *Search pattern, P:* ACGG
 String, S: ACGTAGGATACGGAT

The search pattern P occurs once in string S , as shown in Example 2.2 and this is achieved by a process called pattern searching which involves searching for a pattern in a given string using any pattern searching algorithm. A pattern can be called a repeated pattern if it occurs more than once, and can also be referred to simply as a repeat. Example 2.3 shows this scenario, that is, search pattern ‘ACGG’ is a repeated pattern or repeat in string S .

Example 2.3



The Example 2.3 shows pattern P as repeated pattern, that is, 'ACGG' in the given string. This shows example of how the repeated pattern is identified in any string.

2.2.3 Suffix Tree, Generalized Suffix Tree, and KMER-based Suffix Tree

A Suffix Tree is a data structure popularly used in string algorithms and it is used to index a string by representing the string with its suffixes. Once the Suffix Tree, is constructed it becomes very efficient to perform various operations like pattern searching, location of the repeats and so forth. A generalized Suffix Tree is more comparable to the Suffix Tree, however the Suffix Tree is built by using words in the string rather than characters as in the generalised Suffix Tree construction. KMER-based Suffix Tree is a variant of the generalised Suffix Tree which uses words of fixed sizes to construct the tree. Detailed explanation and examples of these three related data structures is in Chapter 3.

As the researcher attempts to address the issue of efficiency in repeat finding algorithms, several approaches have been explored, including dynamic programming, which is an algorithm design technique that works by breaking a complex problem into simpler sub problems. This approach was explored in an attempt to improve the performance on repeat finding algorithms. However, it was found to be limited both in terms of memory requirements and processing time [28]. The Burrows–Wheeler transformed works by including a special character called a terminal character, \$, to maintain the definition of a suffix which states that, no suffix should be a prefix of another suffix [29]. Hence, any sequence to be analysed will terminate with this character before generating its Burrows–Wheeler transformed sequence. This will be discussed in considerable detail in Chapter 3 (Literature Review section). Several repeat finding algorithms have been proposed, and though they all serve the same purpose of repeat finding, they differ in terms of the techniques and the data structures used. Consequently, they differ in terms of time and space requirements. Besides progressive improvements over time, they still fall short on [7].

There are several types of algorithms in DNA analysis, one of which is repeat finding algorithms. This is an algorithm which searches for repeats in a DNA sequence. Repeat finding algorithms can be largely categorised as *ab initio* and library-based algorithms. Library based are those that search for known repeats, whereas *ab initio* are those that search for novel repeats [30]. The present research therefore focuses on *ab initio* algorithms. Other types of algorithms used include construction algorithms. These focus on the efficient construction of the data structures so as to yield efficient pattern searching or repeat finding. This research also focused on improving the creation of the generalised Suffix Tree in terms of the memory used and time taken during the construction.

2.2.4 Search and Construction Algorithms

Once construction or indexing of the Suffix Tree is done, searching in the tree becomes simple. Suffix Trees are more challenging to engineer when it comes to efficient construction. However, once the construction is done the searching is fast. The searching of a pattern is done in constant time. The search process involves matching character by character from the root down until the search pattern is fully matched. The search pattern is matched starting from index 0 and onwards until the whole search pattern is matched or a match cannot be found [31]. Various methods are employed for indexing child nodes, by employing the array size $|\Sigma|$ at every node. It is possible to get a potential matching child in constant time, resulting in $O(m)$ complexity. The space complexity for this method is also $O(n \cdot |\Sigma|)$. Cole et. al. [32] explored an improved method with a space analysis of $O(n)$ space using an assumption of a varying alphabet size as a variable. The most optimal search achieved with demand of $O(n)$ space complexity is $O(n \cdot \log|\Sigma|)$. A typical example of such data structure is the Binary Search Tree. To make use of that search, children in a node should be sorted first; hence the construction complexity of a Suffix Tree using a binary search is bounded by the complexity of sorting.

2.3 Applications of Generalized Suffix Trees

In this sector we provide other applications of Suffix Trees to show their importance. As highlighted before, searching for a pattern in a Suffix Tree is done in $O(r)$ time using a sequence of length r . This impressive time property of the Suffix Tree becomes very useful in molecular biology when examining for short sequences in a lengthy DNA sequence. Comparing strings is also useful in homology tests and in particular, identification of longest common substring of two or more strings. Palindromes, inexact

matches, tandem repeats and so on can also be easily found utilizing the generalized Suffix Tree data structure.

2.3.1 Longest common substrings

Homology tests include finding the level of similarity among strings. Hence, finding the longest common substring (LCS) is a common feature in molecular biology. The LCS is easy to find in a generalized Suffix Tree. This is usually used when finding consensus among DNA sequences. Genes which construct proteins usually evolve with time, however functional gene regions remain the same to work correctly. Lessons can be drawn from what has been conserved over time by identifying the LCS of the same gene in different species. The LCS problem is a special case of edit distance, and this is more interesting since the LCS is usually faster way to calculate the edit distance [33].

Given n strings, $s_1, \dots, s_m$ the problem is to identify the LCS. The Suffix Tree is constructed first. The generalized Suffix Tree consists of internal nodes and each internal node should be labeled by $i_1, i_2, \dots, i_k$ for each internal node v . An internal node or point labeled with all strings symbolizes the common string with the length as the distance of the node. The longest depth of all the common substrings gives us the LCS. Figure 2.2 attempts to illustrate the kind of output we get from a generalized Suffix Tree which is in polynomial time.

INPUT

```
G C A A G T C T A A T A
C A A G G T T A T A T A
G C A A T T C T A T A A
C A A T T G A T A T A A
G C A A T C A T A T A T
```

OUTPUT



Figure 2.2 Output form GST in polynomial time

2.3.2 Genome projects

Genome projects mostly use Suffix Trees for mapping and the three popular projects are *Arabidopsis thaliana*, *Saccharomyces cerevisiae*/yeast and the *Borrelia burgdorferi* genomes. Under the *Arabidopsis thaliana* project, the Suffix Trees are applied in three ways [34], namely: searching for DNA sequence contaminations, identification of duplications in sequenced fragments and, lastly, searching for known biological patterns in sequences. In these *Saccharomyces cerevisiae* and *Borrelia burgdorferi* projects, the key data structure is the Suffix Tree. In this case it is used to resolve the problem of fragment assembly, that is, to find the complete sequence given sequence fragments.

2.3.3 DNA contamination

During DNA processing in the laboratory, usually foreign DNA sequences can contaminate original sequences and this poses a very serious challenge, since it can affect the whole experiment. However, normally the potential contaminating sequences are known and this issue can be sculpted as explained in [21]. The problem is modeled in the following manner. Given sequence S (original sequence) and a set of sequences which are possible contaminants C . The technique is to locate all substrings of $S \in C$ occurring in C and are longer than some specified length L . A generalised Suffix Tree is built of S and C , all internal nodes are marked with w which represents part of path of S . Then finally pick all the all path length of L marked w , this will show the parts, which are likely to be contaminated.

2.3.4 Plagiarism detection

Plagiarism detection was one of the initial applications of Suffix Trees [35]. Suffix Trees provide an easy and fast way of identifying sections of texts that have been plagiarized. Original text is combined with text that is likely to have been plagiarized, the Suffix Tree is then created and the nodes with long strings show the overlap. This is important since soft copy material such as documents and books have become increasingly available leading to an easy way of copying text and claiming to be one's own.

2.3.5 Code duplication

Code duplication is related to plagiarism detection. However, this makes it easier to maintain software large software systems [36]. Code duplication is tracked and dealt with. The type of Suffix Trees suitable for this is the parameterized Suffix Trees.

2.3.6 Information retrieval

Information retrieval has been the most popular application of Suffix Trees. Recently it has been used in Chinese documents [37]. This suits the application well given that Suffix Trees are most useful to words without boundaries such as in Chinese language, that is, a Suffix Tree can be made by setting every individual letter as a word and each letter in the string is completely represented contrary to other methods which require error-prone segmentation. Other uses include searching for maximal repeats, super-maximal repeats, circular string linearization and so forth.

2.4 Suffix Tree Implementation Issues

In the subsequent section we discuss briefly the two common issues involved in the implementation of Suffix Trees, which apply to our study.

2.4.1 Alphabet Size

In most practical applications of the Suffix Tree, the alphabet size is an issue when it's not constant. This adds another problem when a constant alphabet is not assumed, largely because all root edges represent the alphabet characters hence if this is not known beforehand, the memory estimates of the Suffix Tree will remain unknown too. Other solutions to this includes hash tables [38]. In this study we will assume a constant alphabet since DNA and protein sequences consist of a constant alphabet, which we are using as our datasets.

2.4.2 Storage Considerations

Usually a compact Suffix Tree takes $O(n)$ space. The ability to know the memory footprint of Suffix Trees beforehand can be tricky yet important to guarantee successful complete construction of the Suffix Tree without running out of memory during construction. However, for the purpose of this study we will be using KMER approach, which makes memory requirements a bit more precise since the size or depth of the Suffix Tree is known beforehand. This can be alternatively formally proven by [39] to approximate the space requirements of the Suffix Tree.

2.5 Computation Models

In this unit we explore the concept of 'computation'. This is fundamental to the study since the proposed algorithms will be based on these models of computation. Access

machines, also called mathematical machines process some input to produce some output [40]. The input can be a finite sequence. Mathematical machines are almost similar to real computers however they are theoretical, with omission of inessential features such as hardware bugs, input size etc. A discussion of three popular machines is provided, that is, Turing machine, Random Access Machine (RAM) and the Parallel Random Access Machine (PRAM).

2.5.1 Turing Machine

This is the first model of computation that was proposed by Alan Turing before programmable computers. The model consists of control unit and infinite storage of memory, hence the simplest machine over other machines [41]. They are very abstract and they differ from real computer in a number of ways. The major weakness of the Turing machine is the inaccessibility of its memory, that is, in order to read a distant memory cell, you have to traverse through other intermediate cells. However, this weakness is rectified by a model called Random Access machine, which we will discuss next. Figure 2.3 shows the Turing machine diagrammatically.

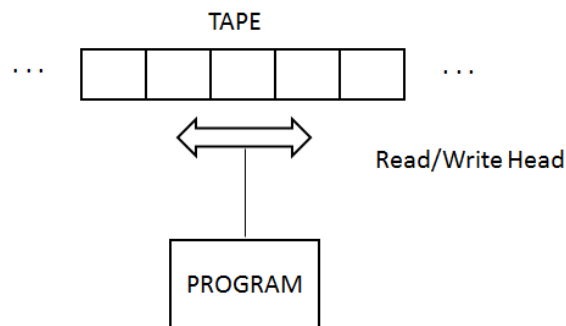


Figure 2.3 Alan Turing Machine

2.5.2 Random Access Machine (RAM) Model

The RAM can access any memory cell in one step. The RAM is programmable, that is, it can be programmed in some programming language. The RAM and Turing machines are similar in many ways. In fact, any operations, which can be done on Turing machine can also be done on RAM [42]. It is crucial to note that programs specified on RAM are only serial programs. **Beside its fast memory access**, it does come with a cost of addressing, that is, the address of the memory cells have to be stored somewhere for us to be able to read a memory register. An illustration of RAM is provided diagrammatically in Figure 2.4.

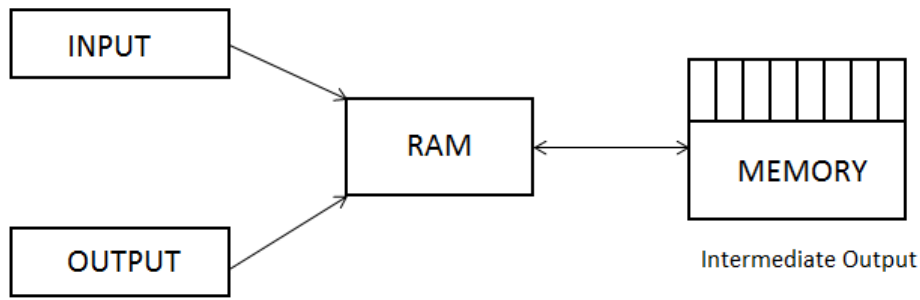


Figure 2.4 The RAM

In this study we used the RAM model for the serial algorithm, the RAM model is the standard model used for sequential algorithms in literature hence will follow the same direction [38] [42]. This will be used for time and space complexity algorithm analysis.

2.5.3 Parallel Random Access Machine Model (PRAM)

This is a machine model designed for parallel algorithms/programs. Several processors are attached to a single block of memory. A PRAM model consists of similar processor type and share a common memory unit. The architecture can be shown in Figure 2.5.

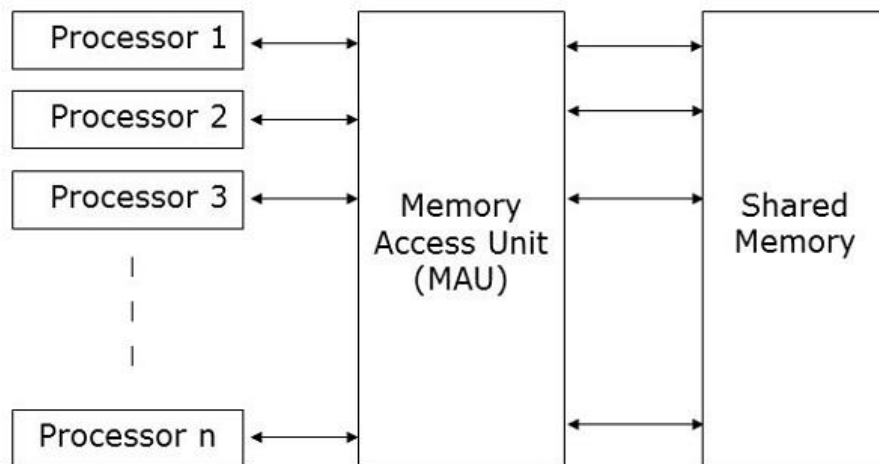


Figure 2.5 PRAM Architecture

Several processors can perform independent operations on the same data item at a certain time. This may result in simultaneous access of the same data item in the same memory location by different processors. To address this problem, the following rules have been introduced on the PRAM model:

- Exclusive Read Exclusive Write (EREW) – No 2 CPUs are permitted to read from or write to the same data item concurrently.

- Exclusive Read Concurrent Write (ERCW) – No 2 CPUs are allowed to read from the same data item at the same time, but are permitted to write to the same data item simultaneously.
- Concurrent Read Exclusive Write (CREW) – All CPUs are allowed to read from the same data item concurrently, but are not permitted to write to the same data item simultaneously.
- Concurrent Read Concurrent Write (CRCW) – All CPUs are permitted to read from or write to the same data item instantaneously.

In this study, we used the CREW PRAM model since it is best suited for the algorithms, that is, concurrent read of sequences which does not corrupt any data and exclusive write to avoid corruption of the nucleotides [43].

2.6 Evaluation Methods

This section discusses different ways of assessing algorithm performance in order to benchmark with other existing algorithms. The two ways of analysing algorithm performance are theoretical analysis and experimental analysis as discussed below. Before considering the evaluation methods, the algorithms need to be checked for correctness by using mathematical proofs [42]. Proving the correctness of an algorithm is done by following two basic steps, which will be shown later in this study [42]. It is also important to note that the formal proofs are also applied to confirm and validate the complexity of an algorithm.

2.6.1 Theoretical Analysis

The following sections provide explanations of complexity analysis and correctness proofs used in algorithm analysis.

2.6.1.1 Complexity Analysis

Algorithm complexity is used to approximate the behaviour of the algorithm in terms of how it scales as input size increases. It measures the asymptotic complexity of the algorithm, for instance given the growth rate function in equation 2.1:

$$f(n) = n^2 - n \log n + n - n^{\frac{2}{3}} + 1 \quad (2.1)$$

From equation 2.1, the asymptotic efficiency would be n^2 since it is the fastest growing element in the equation. The advantage of this analysis is that it is independent of the language, programming style, hardware, memory and processing capacity, which somehow gives a fair measure of the algorithms performance in terms of their comparison. There are basically two computation models in algorithm analysis which are RAM and PRAM. RAM is a computation model based on sequential algorithms whereas PRAM is based on parallel algorithms as already highlighted.

2.6.1.2 Correctness Proofs

Algorithm correctness is asserted with respect to its specification [44]. Correctness is defined as the adherence of an algorithm to its specifications. An algorithm is checked for correctness to verify if the algorithm correctly solves a given problem. The checking process is done through formal proofs of correctness. An algorithm and its specification is given formally for a mathematical proof. There are various techniques used to proof algorithms such as loop invariant, Hoare logic [45], and mathematical induction. Hoare logic is defined as a formal system for rigorous reasoning about computer algorithms. It makes use of axiomatic language to argue correctness of algorithms. For instance, consider the Hoare triple, given $\{x = 10\} y := x * 2 \{y > 0\}$. This is correct since x is 10 then y is always larger than 0 in this instance. However we can still assume a stronger post condition to this, that is $y > 10$ and $y < 30$. So this post condition is stronger and therefore more informative. Hoare logic includes what is called Hoare Triples which is formal reasoning about a program using the pre and post conditions [45]. The syntax is as follows:

$$\{P\} S \{Q\}$$

P and Q represent predicates and S is the program, for instance, $\{x = y\} x := x + 3 \{x = y + 3\}$. So if the program starts executing where P is true, then it will execute S , and S will terminate when the value of Q is true. Hoare logic is mostly applied to WHILE loops in an algorithm.

A loop invariant is a predicate used to define a loop [38]. A loop invariant is mostly applied to FOR-LOOP. The predicate should be satisfied before entering any single

iteration and should remain satisfied after the iteration [46]. For instance, a WHILE LOOP has the following structure:

While (B):
 $\{S\}$

B is the Boolean result of the condition of the loop and S is the body of the loop, that is, the statements to be executed once $B = \text{True}$. Hence as shown in the Figure 2.6 the invariant always holds with every iteration of the loop.

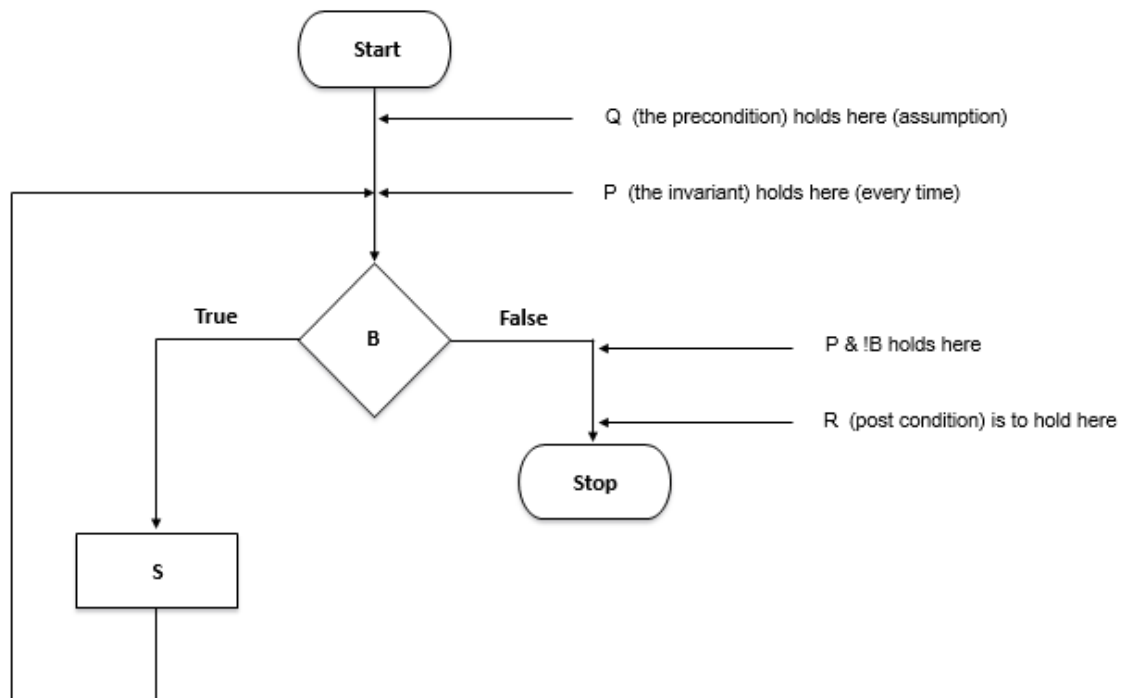


Figure 2.6 Executing a loop whilst maintaining the loop invariant in a flow chart

Mathematical induction is also a mathematical proof method used to prove logical statements especially ordered sets [47]. It is a direct proof normally accomplished in two steps, the first step is the base case. This step aims to prove the first element and then the second step is called inductive step which aims at proving that if the first element case is assumed true then it should be true for the next element too [48]. A basic example is given in Example 2.4 using proof of induction:

Example 2.4:

$4n - 1$ is a multiple of 3

A proof for the statement: if $4n - 1$ is a multiple of 3 is provided in the following steps:

1. First we show it is true for $n = 1$

$$4^1 - 1 = 4 - 1$$

Yes 3 is a multiple of 3. That was easy.

$$4^1 - 1 \text{ is true}$$

2. Assume it is true for $n = k$

$$4^1 - 1 \text{ is true}$$

Now, we prove that $4^{k+1} - 1$ is a multiple of 3

$$4^{k+1} \text{ is also } 4 * 4^k$$

And then split $4 \times$ into $3 \times$ and $1 \times$

Each of these are multiples of 3

This is because:

- $3 * 4^k$ is a multiple of 3 (we are multiplying by 3)
- $4^k - 1$ is true (we said that in the assumption above)

So:

$$4^{k+1} - 1 \text{ is true}$$

□

In this study we use mathematical induction and loop invariant proof techniques to prove the correctness of the algorithms.

2.6.2 Experimental Analysis

Experimental analysis involves implementing the algorithm and recording the actual running time and space used during program execution. The specifications of the underlying computer are given. This analysis affords us a method of testing the algorithm in real environment since it is possible for two algorithms to have the same algorithm complexity but have very different performance under the same environment when implemented, for this reason it is better to perform experimental analysis to complement theoretical analysis. However, implementation of an algorithm for experimental analysis has its limitations. These come from different factors such as programmer skill, programming language, and compiler.

2.6.2.1 The Speedup Performance Metric

When parallel programs are implemented, a common performance metric used is speedup. This is used to measure the performance gain achieved by applying parallelisation to a given application over a sequential implementation [49]. In other words, speedup is a performance measure that gives the relative benefit of solving a computational problem in parallel. For instance, when we decide to parallelise bubblesort, we assume the following:

Example 2.5

1. The sequential program of bubble sort with 10^5 records takes 150s
2. The serial program can sort the same list in 30s
3. The parallel program of bubble sort, for the same list takes 40s

The performance speedup is calculated as sequential time divided by parallel execution of P processors. The performance speedup in our Example 2.5 is therefore $30/40 = 0.75$, not $150/40 = 3.75$ because the performance speedup is computed with respect to the best serial algorithm. In this case, the best serial algorithm is serial quicksort algorithm. We will use this metric in our study to measure the performance of our parallel algorithm.

2.7 Summary

The chapter covered different types of repeats with respect to DNA. Various data structures were discussed in terms of their strengths, weaknesses and applicability. These included Suffix Trees, generalised Suffix Trees, and Suffix Arrays. Common techniques

applied in this area of DNA analysis were also presented, including parallel algorithms. The data structures discussed make use of algorithms during their construction, hence several algorithms were reviewed in their attempt to provide an efficient data structure.

Chapter 3

Literature Review

3.0 Introduction

Literature will be discussed in several sections. This is a crucial section to develop a body of knowledge in this study [50]. That is, types of repeats, data structures used in this area of pattern searching, common techniques and methods including their limitations will be reviewed. Lastly, a discussion of pattern searching algorithms follows, in terms of their performance.

3.1 Types of repeats

A DNA sequence, can be viewed as a string of alphabetic characters containing four characters, that is, A, C, G and T extracted from the molecules of DNA sequencing process ($\Sigma = \{A, C, G, T\}$). DNA sequencing in simple terms is a process of figuring the precise order of the base-pairs in a DNA strand. Using the following example of a sample DNA sequence:

Example 3.1

String $S = \{GGGGACGTACGTACGTAAGTACGT\}$

These characters stretch up to an approximate size of 3 billion character pairs making it very difficult to process, because of their high space and processing requirements [51]. ‘Period’ and ‘exponent’ are frequently used terms in repeat searching algorithms. Period refers to the length or size of a repeat, while exponent refers to the number of occurrences for a repeat in a sequence. The \$ symbol will be used to practical represents the end of the string to be searched or the sequence. A tandem repeat from Example 3.1 can be $\{ACGT\}$ and an interspersed repeat can be $\{TACG\}$ all of period 4. Therefore, a tandem repeat is a pattern, which occurs contiguously and an interspersed repeat is a pattern which occurs in a dispersed manner. Repeats can also be categorised according to their sizes. They are classified into 3 categories, which are, microsatellites, mini-satellites and satellites [27]. Microsatellites range from 2 to 5 inclusive, for instance, in Example 3.1, CGTA is a microsatellite repeat. Mini-satellites are of sizes 6 to 100 inclusive and satellites are any repeats larger than 100. Although these repeats slightly vary in other literature, the variance is not huge.

Another criterion used for classifying repeats is exact and approximate. From Example 3.1, an exact repeat would be $\{ACGT\}$ with an exponent of three and an approximate repeat would be $\{GTACGT\}$ with period of six and exponent two with only variation on the second position of the repeat. A maximal repeat can be formally defined as; let S be a sequence of length n , $S = S[1 \dots n] = s_1s_2 \dots s_n$, where every character s_i , where $1 \leq i \leq n - 1$, belong to a finite ordered alphabet Σ of size σ , while $s_n = \$$ is a special character which does not appear in Σ . Hence, a maximal repeat of sequence S is a substring of S that occurs in S at least double such that every extension of the substring occurs in S less times [28]. For instance, given the following example string:

Example 3.2

String $S = \{GATACGTAGATAGATGTACGTAG\}$

Then $\{GAT\}$ is a maximal repeat whilst $TACG$ is not, since $TACG$ can be extended to $\{TACGI\}$ and the extensions do not occur fewer times than the repeat. The initial problem in maximal repeats finding is identification of the candidate maximal repeats. There are two ways in addressing the problem which are filtration and the LCP array. Filtration involves identifying candidate repeats as seeds then extending to the left or right, to ensure if the repeat can still be matched. The LCP array makes use of the suffix array to construct a table or array listing for the longest lengths of the common prefixes in lexicographic order. The LCP array will be discussed further in Section 3.2.5. Typically, in genomes, repeats are not usually the same or exact and these are called approximate repeats and there are several ways to measure the differences between these repeats such as using compression based and distance based definitions. Compression based definitions define a repeat as a region which can be repeated plus a set of mutations that this repeat can be coded using less space than coding the same region directly. These compression based techniques have some usage limitations like the Star algorithm by Delgrange et al.[52], that is, it allows searching for repeats consisting of repetitions only of a given pattern. Approximate repeat finding involves the use of distance based definitions. Distance based definitions are based on two distance metrics which are Hamming distance, and Levenshtein distance. An approximate repeat is simply a repeat with some slight variations of some pattern. Alternatively, an approximate repeat can be

defined as a repeating unit where the repeating units are similar according to some distance metric as shown in Figure 3.1.

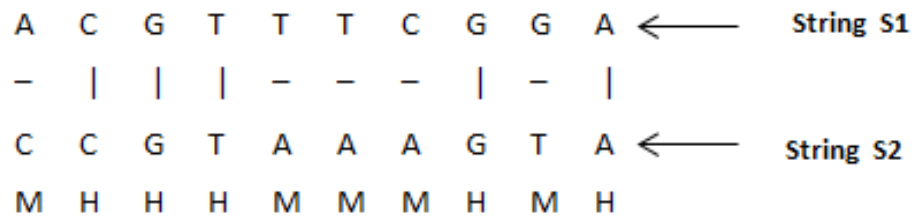


Figure 3.1 Two repeats aligned, M is a Miss (mismatch) and H is a Hit (match)

From Figure 3.1, String S_1 is an approximate repeat with Levenshtein distance of 4. Levenshtein distance is the least number of processes required to transform the first string into the second string and these operations are known as *indels*. In most cases the number of operations is bound to a certain given number. Hamming distance only holds when two strings in question are of the same size, that is, have the same lengths. The Hamming distance is the frequency of mismatches when two strings are aligned character by character. It is therefore possible to find repeats where each unit differs by k -mismatches.

3.2 Data Structures

Generally, a data structure specifies how the information is organised together with access methods. This section outlines different examples of data structures, which have been used in the area of DNA analysis specifically in repeat finding algorithms. These data structures are important as they are the basis of algorithms and contribute to performance of an algorithm with factors such as space consumption and traversal methods

The current study focuses on a tree-based data structure. Such data structures, in general, are used for text indexing. Indexing problems are usually solved by using tree traversal techniques. For instance, in computer linguistics a misspelled word can be automatically corrected using this technique [53]. A related application has been explored by [54] by calculating the edit distance. We discuss the following variants of tree-based data structures with various applications in Sub-sections 3.2 and we focus on the most researched data structures in computer science. Tree-based data structures are built with nodes and edges. There are three types of nodes and these include parent nodes, internal nodes and leaf nodes. The parent node is the first node in a tree data structure. The internal

node is a node with at least one child node in a tree data structure and lastly a leaf node is a node with no child nodes. The following sections will show diagrams, examples and further explanations.

3.2.1 Suffix Trie

Sometimes called the Prefix Tree or Radix Tree. The Suffix Trie was the first data structure to be discovered [55] which further led to developments of suffix which is better in terms of space consumption in comparison to the *suffix trie*. A *suffix trie* represents characters in a string as one node per every common prefix. The major advantage of the *suffix trie* is that, it's easy and fast to search a prefix of a word hence the name *prefix tree* and is constructed as follows:

Let $S = S[1, \dots, n]$ be a text of length n over a finite alphabet Σ . A *suffix trie* for S is a tree that fulfils the subsequent properties:

1. Every node is labelled with the character $c \in \Sigma$.
2. Every edge will be labelled with a non empty character of S

We provide Example 3.3 to illustrate how the suffix trie is constructed, for the strings S_1 and S_2 .

Example 3.3

Strings $S_1 = \{abab\}$ and $S_2 = \{baba\}$

The suffix trie for both strings in Example 3.3 is shown in Figure 3.2

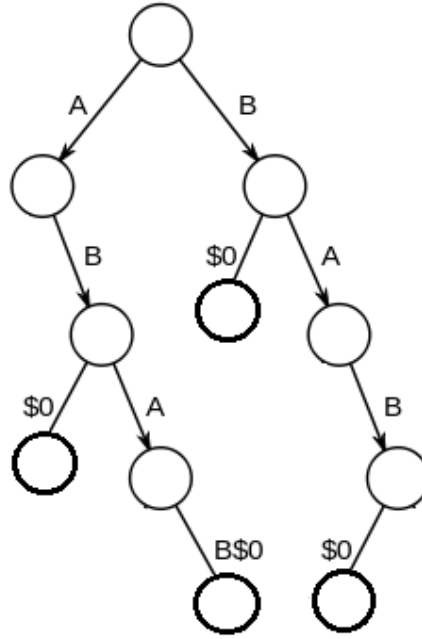


Figure 3.2 Suffix Trie for S1

3.2.2 Suffix Tree

A Suffix Tree can be defined as a compacted trie of all suffixes of a string which makes it easy and fast to carry out several string operations [56] . Such as pattern searching, matching statistics, longest common substring and so forth. Construction of a Suffix Tree is viewed as a pre-processing step to perform various string operations. The Suffix Tree has been first introduced by Weiner [57] with linear time construction of $O(n)$. The Suffix Tree is defined as follows:

Let $T = T[1..n]$ be a text of size n , over a finite alphabet . The Suffix Tree for T is a tree with n leaves that have the following properties:

1. Every internal node except the root has, at least, one child.
2. Every edge will be labelled with a non-empty substring of T
3. Every edge has a different initial letters
4. Combination of labels from the root to leaf gives the suffix of T .

We will demonstrate how the Suffix Tree is created using Example 3.3 for string S_1 . We first append the special terminal character, \$, to string S , so $S = \{abab\$ \}$, a Suffix Tree

of S is a compressed trie of all suffixes of $S = \{abab\}$ and therefore can be graphically represented as shown in Figure 3.3.

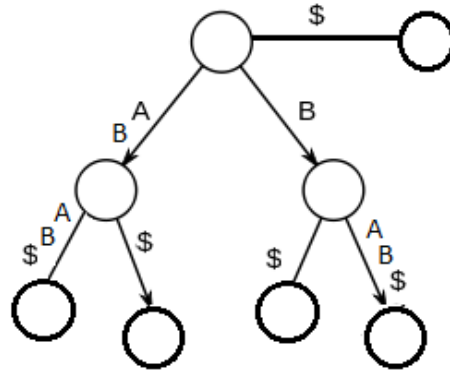


Figure 3.3 The Suffix Tree

The major limitation with the Suffix Tree is that, it consumes a lot of memory during its construction therefore, it requires a lot of memory [58]. This challenge becomes even more complicated when working with the entire human genome which is huge.

3.2.3 Generalized Suffix Tree

There are several methods of constructing a generalised Suffix Tree for a single text in linear time. By making use of these methods we can extend them to symbolize a set of suffixes of strings. When these suffixes are exemplified in a tree, the resulting tree is called the *generalised Suffix Tree* (GST). One simple way to construct a GST is by appending unique symbol to each of the strings in the set, and then finally concatenates all the strings together, to create one complete string. The disadvantage of creating a GST this way is that; the GST represents only the substrings of the concatenated string. Making use of Example 3.3, with the two strings S_1 and S_2 , we generate the GST in Figure 3.4., with unique terminal characters $\$0$ and $\$1$. We modify the tree to save space and improve construction time in this study, since this data structure suits the nature of our dataset (DNA and Protein Sequences).

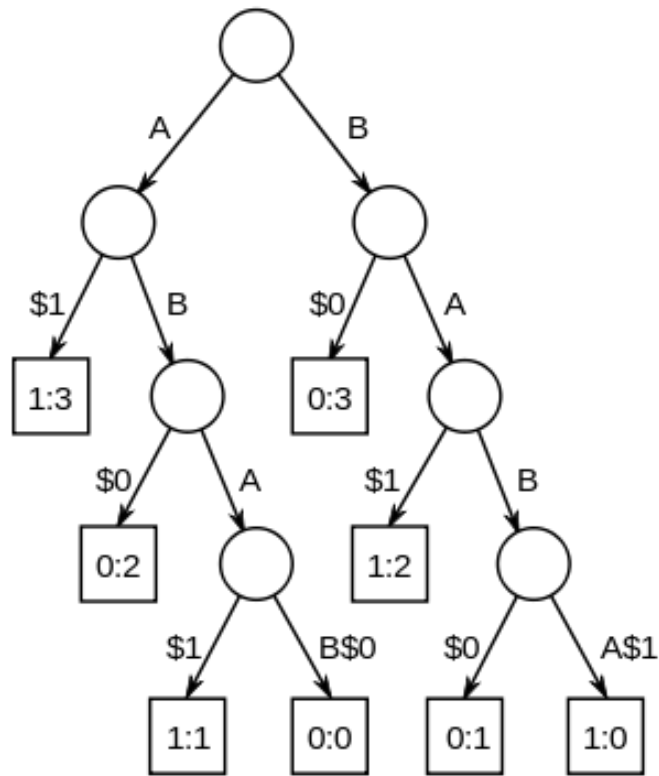


Figure 3.4 Generalised Suffix Tree for string S1 and S2

3.2.4 Suffix Array (SA)

A Suffix Array is a sorted array of all suffixes of a sequence. The data structure is memory efficient developed from a Suffix Tree [59]. A Suffix Array is defined by using an example as, given a text T , of length n the suffix array of T , is therefore an array of integers spanning from 0 to $n - 1$ in lexicographic ordering of the n suffixes of the string $T\$$. Table 3.1 displays the suffix array for $T = \text{mississippi\$}$, with $\Sigma = \{i, m, p, s\}$.

Table 3.1 Suffix Array and LCP Array

i	T_i	$T_{suf_array[i]}$	$Suf_array[i]$	LCP
0	mississippi\$	\$	12	0
1	ississippi\$	i\$	11	0
2	ssissippi\$	ippi\$	8	1
3	sissippi\$	issippi\$	5	0
4	issippi\$	ississippi\$	2	4
5	ssippi\$	mississippi\$	1	0
6	sippi\$	pi\$	10	0
7	ippi\$	ppi\$	9	1
8	ppi\$	sippi\$	7	0
9	pi\$	sissippi\$	4	1
10	i\$	ssippi\$	6	0
11	\$	ssissippi\$	3	3

Assuming that $\$ < \Sigma$, The LCP column is discussed in Section 3.2.6. The LCP array is an external data structure in an array structure used to store the sizes of lengthiest common prefixes among pairs of consecutive suffixes in a suffix array.

3.2.5 Position Heaps

Position heaps are another type of a tree-based data structure, which is also closely related to the Suffix Tree. This data structure is used for text indexing in order to perform pattern searching [60-62]. Position heaps were first proposed by Ehrenfeucht et al. [60]. A position heap for a string S is constructed by adding suffixes of S into a trie and observing the following two rules [63]:

1. A small suffix is inserted before a longer suffix
2. A shortest prefix for a suffix s that is not already in the tree is inserted into a position heap

For example, given $S = \text{bananababa}$, then the position heap for $S' = \text{ananababa}$ is displayed in Figure 3.5 and Table 2 displays indices for S .

Table 2. Indices for the S

10	9	8	7	6	5	4	3	2	1
b	a	n	a	n	a	b	a	b	a

We insert *ban* in the tree with label 10 since *ban* is the smallest prefix of suffix S that is not already in the position heap [63]. The main disadvantage of the position heap is that it makes use of pointers which require a lot of storage space hence we will not adopt this data structure since it's not in line with one of our objectives of coming up with a space-efficient data structure.

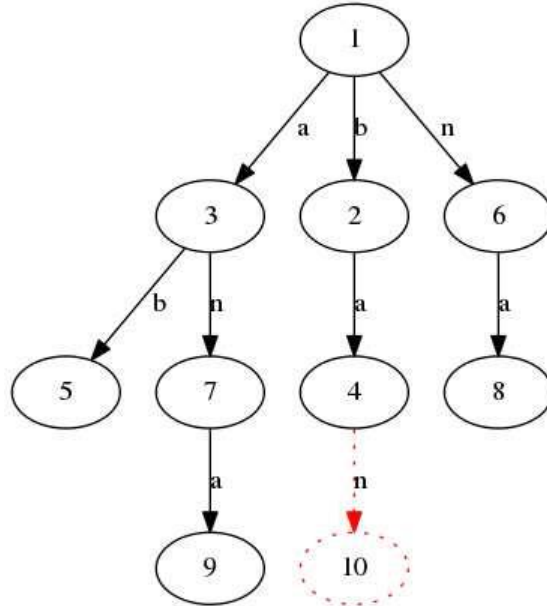


Figure 3.5 Inserting suffix S into the position heap

3.2.6 Longest Common Prefix (LCP) Array

The LCP array maintains the sizes of LCPs among pairs of successive suffixes in a Suffix Array. An LCP array is an auxiliary structure, which can also be generated from a Burrows-Wheeler Transform [59]. LCP array provides an initial point for candidate maximal repeats (as discussed in section 3.1). Hence, each LCP element is a candidate maximal repeat waiting to be confirmed by extending to both left and right preserving the repeat pattern. The LCP array is shown in Table 3.1, last column and the lengths of prefixes which are common are shown in $T_{\text{suffix_array}}[i]$ column. The process for extracting maximal repeats can be shown diagrammatically as in Figure 3.6. This is commonly implemented in the current techniques to improve the search process of

maximal repeats [64]. In our research we are not focussing on specific type of repeats but rather construction of the data structure to allow for searching of various types of repeats hence we will not employ this technique as it applies to improvement of search process.

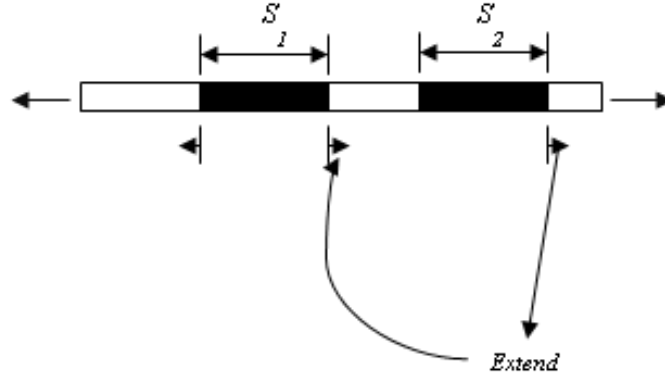


Figure 3.6 Ascertaining maximal repeats

From Figure 3.6, substrings S_1 and S_2 are initially repeats which are then re-compared by extending (character by character) to the left side to confirm if they can still maintain the maximal repeat definition. The same process is recursively repeated to the right side as well. Manzini and Ferragina [65] introduced a way of measuring the difficulty of suffix sorting by making use of the average LCP as in equation 3.1:

$$Average\ LCP = \left(\frac{1}{n-1}\right) \sum_{i=1}^{n-1} LCP(T[SA[i], n], T[SA[i+1], n]) \quad (3.1)$$

This means when the average LCP is big then many characters would have to be analysed in order to determine the suffix sorting.

3.2.7 Full –text index in Minute space (FM Index)

The FM Index is a form of data structure which combines the (BWT) Burrows-Wheeler transform, SA and the Wavelet tree (discussed in the next section). In this section, we discuss the Backward Search, which is at the core of FM index. Other data structures such as Burrows-Wheeler transform, Suffix Array are discussed separately in Sections 3.3.2 and 3.2.4 respectively. Backward Search is a search technique in which searching traverses from leaf nodes of the wavelet tree. This is sometimes called the rank query. Given a pattern P , in a sequence S , P is a *prefix* of a *suffix* and since all the suffixes are alphabetically ordered, then all instances of a search pattern P are found in a neighbouring section of the Suffix Array. The Backward search improves the query performance by using the Burrows-Wheeler Transform in a sequence of paired rank

queries that can be found in a Wavelet tree. Backward search provides p pairs of rank queries, where p is the size of the pattern P . Then the paired rank queries are, as in Eq 3.2:

$$s'e' = C[P[i]] + \text{rank}(s-1, P[i]) + 1 = C[P[i]] + \text{rank}(e, P[i]) \quad (3.2)$$

Where s represents beginning of the range and e represents end of range. First, $s = 1$ and also $e = N$. If $e < s$, then P is not found in S . C is a lookup data table consisting of the number of occurrences of all characters in the alphabet which sort alphabetically before $P[i]$. This particular data structure is complex as it requires a combination of other various data structures to construct therefore requires a lot of processing requirements. For this reason, we will not be considering this data structure in order for us to reach our objectives of space and time-efficient construction.

3.2.8 Wavelet Tree

A wavelet tree is a succinct data structure and is constructed like a balanced binary tree. The tree supports rank, select, access queries and backward search. To construct the wavelet tree, the string has to be first converted to a binary search tree consisting of bit vectors, where a 0 is used to replace half of the symbols, and a 1 is used to replace the other half. However, this introduces haziness at each stage. This alphabet gets cleaned and re-encoded, so the haziness reduces, until the uncertainty disappears completely.

Using string $S = \{a, b, r, a, c, a, d, a, b, r, a\}$, the alphabet is listed in an alphabetic order of the characters of S , which is, $\Sigma = \{a, b, c, d, r\}$, however our final set of the alphabet should include the special symbol, \$, as discussed before, hence $\Sigma = \{a, b, c, d, r\}$. Finally the alphabet $\{a, b, c, d, r, \$\}$ and is mapped to $\{0, 0, 0, 1, 1, 1\}$, that is, a will map to 0, d will map to 1 and so forth. The left sub-tree is formed by extracting just the 0 coded symbols $\{a, b, c\}$ and re-coding them, by dissecting the new alphabet: $\{0, 0, 0, 0, 1, 1, 1, 1\}$, take note, on initial phase, an e can be coded as 0, and now it is coded as 1 (it will become a 0 again at the leaf node). This process is defined recursively on each sub-tree and level. Querying the wavelet tree is done in three ways. This means that there are three types of queries supported by the wavelet tree, which are rank, access and select queries. A complete Wavelet Tree is shown in Figure 3.7.

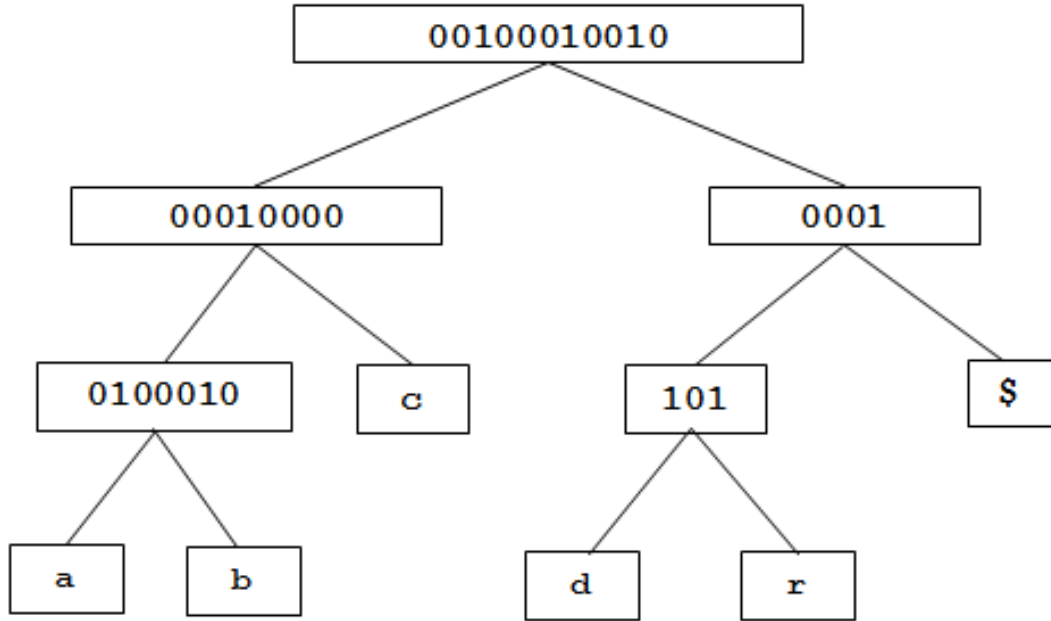


Figure 3.7 Wavelet Tree

3.3 Techniques

This section discusses different techniques used in pattern searching algorithms. These techniques perform the core operations of construction of data structures, space saving and searching for patterns in a string. These are therefore very important since they can greatly affect the processing time as well as memory requirements.

3.3.1 Lempel - Ziv factorization (LZ)

LZ factorisation is an algorithm usually applied in compression algorithms. Compression helps save space in disk-based algorithms. LZ algorithm is commonly applied to repetitive strings such as DNA sequences [66, 67]. These are algorithms which make use of main memory and additionally hard disk space to meet increasing space requirements unlike in-memory based algorithms. In-memory algorithms only require main memory without the aid of hard disk space. We adopt the approach of the in-memory algorithms to improve time requirements because disk-based tends to be slow due to time wasted in I/O requests. LZ parses a given string in a greedy manner into different phases [67]. In simple terms, given the string $\{AABABBBB\}$ The LZ takes the first character A which is the shortest phrase ever seen in the string, then AB , ABB and so on in that manner ($A|AB|ABB|BAB$).

3.3.2 Burrows-Wheeler Transform (BWT)

BWT is another compression algorithm that groups identical characters into similar runs making it easier to compress [68]. A strong advantage of the BWT lies in the fact that, it produces a compressible transformed string and it is reversible. This is important given the major problem of high memory requirements in DNA analysis. BWT can be outlined as follows, given a sequence S , compute all the cyclic permutations of S and arrange them in alphabetic order forming the Burrows-Wheeler Matrix (BWM). The first and last columns of the BWM are labelled as F and L respectively. When storing the result, only the L column and the index containing the full string are stored. An example of the Burrows Wheeler Transformed string $S = panama\$$ is given in Figure 3.8

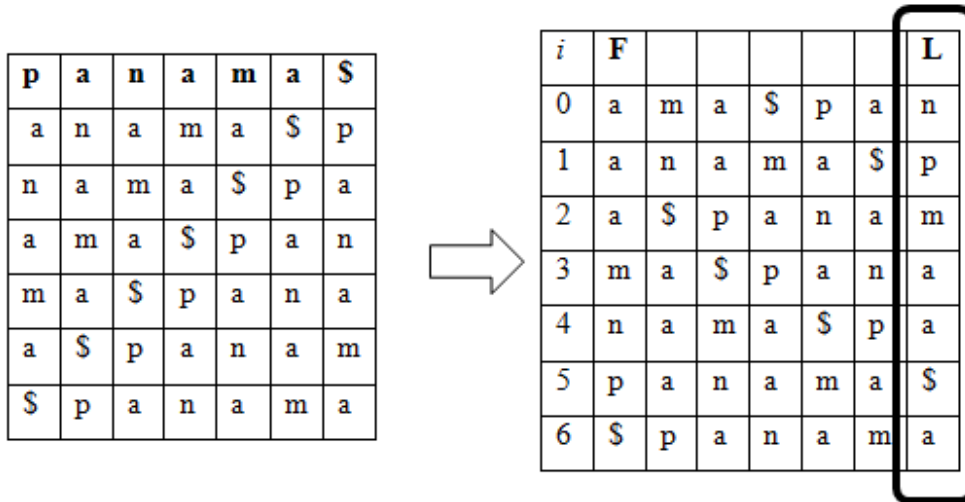


Figure 3.8 Burrows-Wheeler Matrix

This means that, the $BWT(S) = \{n, p, m, a, a, \$, a\}$ and index 5 are the only information to be stored. One last important step to be performed is the reconstruction of the original string. This is called the Burrows-Wheeler Backtransform which is the reverse of the process outlined. Huffman coding (discussed in Section 3.3.3) or any other coding method can then be used to compress the Burrows-Wheeler transformed string by utilising the BWT property which arranges characters in similar runs.

The data structures introduced earlier like the Suffix Trees and suffix arrays have high space requirements as mentioned earlier so this Burrows-Wheeler transformed string can then be used. However, we explore the strategy of truncating the tree since BWT is a complex process and therefore requires more processing time and space.

3.3.3 Huffman coding

Huffman code is a special kind of a prefix code which is normally applied to lossless data compression. Hence, Huffman coding becomes the process of searching and using such code. The algorithm was developed and named after David. A. Huffman [69]. Huffman can be used to save space in disk-based algorithms and since we are employing in-memory algorithms, this coding was not adopted in this study. The Huffman code takes in a character in a file or string and output a variable-length code table for encoding the original symbols. Just like in another entropy encoding symbols the commonest symbols are encoded with less bits than the common ones. Huffman coding works by first generating a binary tree and store in an array. The number of codes determines the size of the array. There are basically two kinds of nodes that is leaf nodes and symbol nodes.

The process starts with the leaf nodes consisting of the likelihoods of the symbols they represent. The next stage creates a new node with tow child nodes containing the smallest probability. The previous two nodes are combined into one node, and then the new node will then get considered. The procedure runs in repetition up to a point when one node remains resulting in the Huffman tree. Given that, the priority queue require time complexity of $O(\log n)$ per each insert operation, and the tree with n leaves will have $2n - 1$ nodes, this algorithm will run in $O(n \log n)$ time, where n is the number of the symbols. In most cases, algorithms' time complexity is not considered in the selection of algorithm to generate the binary tree, since n represents number of symbols in the given alphabet, which is usually a very small number (in comparison to the message size to be encoded); whereas algorithm complexity analysis focuses on the behaviour of the algorithm when n grows to be very large. It is usually very advantageous to reduce the difference of the length of the code word.

Therefore, this technique can then be applied to the Burrows-Wheeler Transformed sequence in order to reduce the space requirements.

3.3.4 Parallel Algorithms

Parallelisation is the process of distributing computational tasks across more than one processing unit to improve perfomance. Usually, the challenging part in parallel programming is identification of oppportunities for paralllellisation. It is important to note that, parallel algorithms are well suited for modern computers where the multi-core processors now come as a standard in every PC. A multiple cores processor is a single

computing component with two or more actual autonomous central processing units. Parallelisation has been used to improve the speed of carrying a computing task in most tasks which requires more processing power. Analysis of a parallel algorithm complexity is slightly different with that of a sequential algorithm. The parallel algorithm complexity analysis can be carried out in two ways which are, the Work Depth (WD) model and the PRAM model.

The problem in modelling the parallel computers is that there are many different configurations; therefore, making it difficult to choose one multi-processor model that is suitable for all machines as in RAM model. With reference to the WD model, the cost of a parallel algorithm is calculated by analysing the total number of actions that are performed and including its dependencies. An algorithm's work W is calculated as the overall number of actions that it performs and its depth D is calculated as the longest sequence of dependencies on its activities and can be expressed as $P = \frac{W}{D}$. The benefit of using a work-depth model is the fact that there are no machine dependant details to make the design and analysis of algorithms complex.

There are generally three ways when designing parallel algorithms, and these are:

1. Modifying an existing serial algorithm by taking advantage of the parts of the algorithm that are parallelizable.
2. Designing a new parallel algorithm that may have no natural sequential analog
3. Using brute force methods for parallel processing which use an existing sequential algorithm but each processor using differential initial conditions.

In this study we adopt Option 1. We make use of KSA algorithm and take advantage of the parts, which are parallelizable. As mentioned in the introduction of this section, the main challenge in parallel algorithms is to be able to identify parallelizable parts and on deciding how the communication is to be carried out for inter-process communication. It is important to note that there has been a paradigm shift in the processor technology since 2003 from single processor to multi-core processors and this has also impacted on how algorithms are now designed compared to how they were traditionally designed [70]. It has already been proven that performance can be greatly improved by adopting parallel programming approach [70]. Parallel computing is gradually getting popular as the

change of this computing paradigm continues because of its performance improving advantages according to [71].

Parallelisation is always feasible and applicable in a scenario where there are potential parallelisable parts in the serial task and the approximate expected performance increase is explained by Amdahl's Law, which discusses the relationship between the speedup and the portions of parallel part and serial part as shown in Eq. 3.3.

$$Speedup = 1/(s + \frac{p}{n}), \text{ where } s \text{ is } (1 - p) \quad (3.3)$$

Where s represents the percentage of serial parts and p is the percentage of parallel parts. This means, the maximum limit of speedup is bound by cost of the serial section. Hence, with this law it means the performance increase is always guaranteed if there are serial sections which can be parallelised.

3.4 Ukkonen's Algorithm

In this part, we discuss the Ukkonen's state of the art algorithm, which has been used as a benchmark to most Suffix Tree construction algorithms. This is because it is one of the most efficient sequential algorithms, which runs in linear time. Ukkonen proposed an $O(n)$ time algorithm for Suffix Tree construction [72]. We refer the interested reader to Gusfield [21] for further detailed reading with technical explanations including proofs. A detailed step-by-step process of constructing the Suffix Tree for a string $S = abaaba\$$ is also shown in Appendix C. We still provide an explanation of how the Suffix Tree is constructed in detail using the Ukkonen's algorithm. The algorithm is on-line in nature hence the construction can begin with first word and stop at any time. The construction of the tree is done in the most compact way possible. A terminal symbol is appended to the end of the sequence for the building process to stop. **This** special character is explained more later. The major difference of the algorithm with others is that of suffix links. Ukkonen's algorithm uses suffix links during the construction process. Every node can have, at least, one suffix link pointing to a node closer to the root node within the same sub-tree. If a branch has a string $\alpha\beta$, in which α is the initial word in of that string then a suffix link will be created pointing at β . Figure 3.9 illustrate more on suffix links for a string $s = cacao$ where for instance $\alpha = c$ and $\beta = ao$. More suffix links are not

shown for readability. Dotted arrows show suffix links, suffix counts are shown in red, node IDs are shown on top of suffix counts.

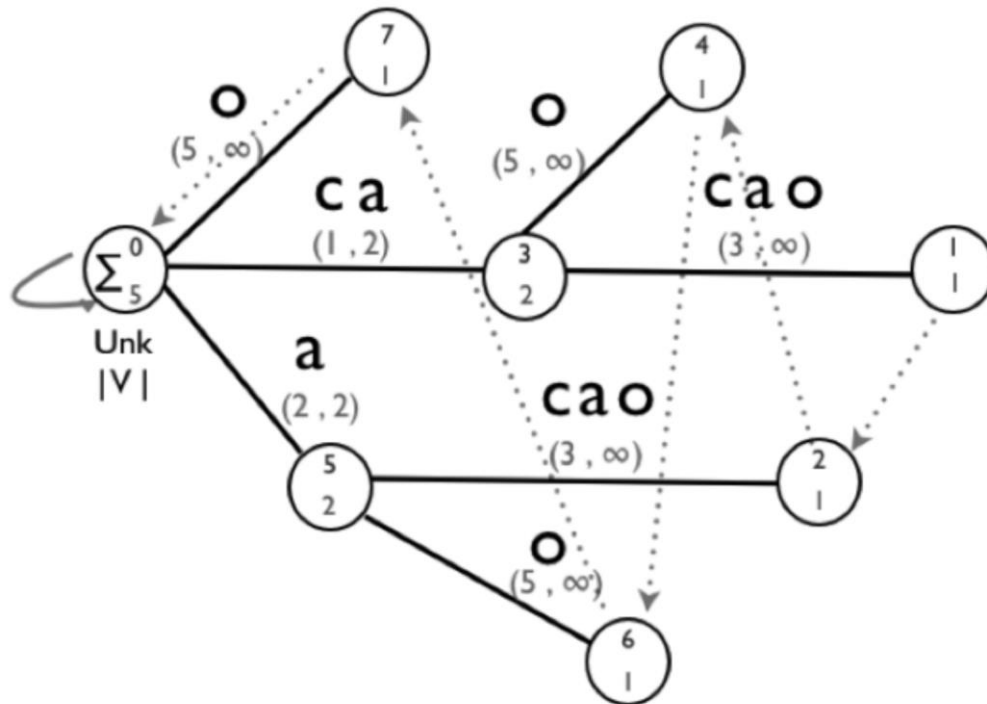


Figure 3.9 Suffix Tree with suffix links for string 'cacao'

Basically suffix links represent a pointer to another node in another edge of the Suffix Tree, which has the same sequence excluding the initial word. In general, the algorithm works by adding a word to the longest edge in the tree, begins at the end, then follows the suffix links back to the root node and appending the new word as a child node to every node which does not have that word as a child node until it gets to the root node. Therefore, the suffix links also add to faster creation of the Suffix Tree data structure. The method will not utilise the suffix links due to the nature of the construction. The construction is based on a generalised Suffix Tree rather than a Suffix Tree. Ukkonen algorithm can be written as shown in Figure 3.10.

Algorithm 1. Ukkonen's Algorithm**Input** : $S = s_1, \dots, s_n$ **Output:** suffix tree representation ST of S

```

1  $ST$  //initialise suffix tree, at this point it's empty
2 for  $i \leftarrow 0$  to  $n - 1$  do
3     Begin phase  $i + 1$ 
4     for  $j \leftarrow 0$  to  $i + 1$  do
5         Locate the path with the LCM with substring
6         if  $[S_{i+1}]$  is not at the end of the path
7         then
8             extend path by inserting the symbol //ensuring symbol is in the tree
9         endif
10    endfor
11 endfor

```

Figure 3.10 Ukkonen's Algorithm

The high-level Ukkonen's algorithm given in Figure 3.10 has time complexity of $O(n^3)$. However, Ukkonen made three observations which enable it to reduce time complexity to linear time. The initial observation is that of avoiding storage of edge labels explicitly in the Suffix Tree sometimes called *edge-label* compression. Instead, indices are used to represent the substrings ultimately saving a lot of space. These indices stipulate the start and end point of the substring in the original string. *Suffix links* (discussed above in this section) is another observation meant to reduce the time for navigating the Suffix Tree. Lastly, the *skip-count* trick, allows for constant time when traversing down one edge. This approach is easier explained with an illustration in Figure 3.11 to help the reader understand. Let $z =$ string between v and end of $(j - 1)$. A counter is initialised at 0, on every step from node $s(v)$ as shown on the right edge in Figure 3.11. Then append the

number of characters on it to the counter and if it's still less than $|z|$ then skip to the child, ultimately after 4 skips the end of suffix j is found.

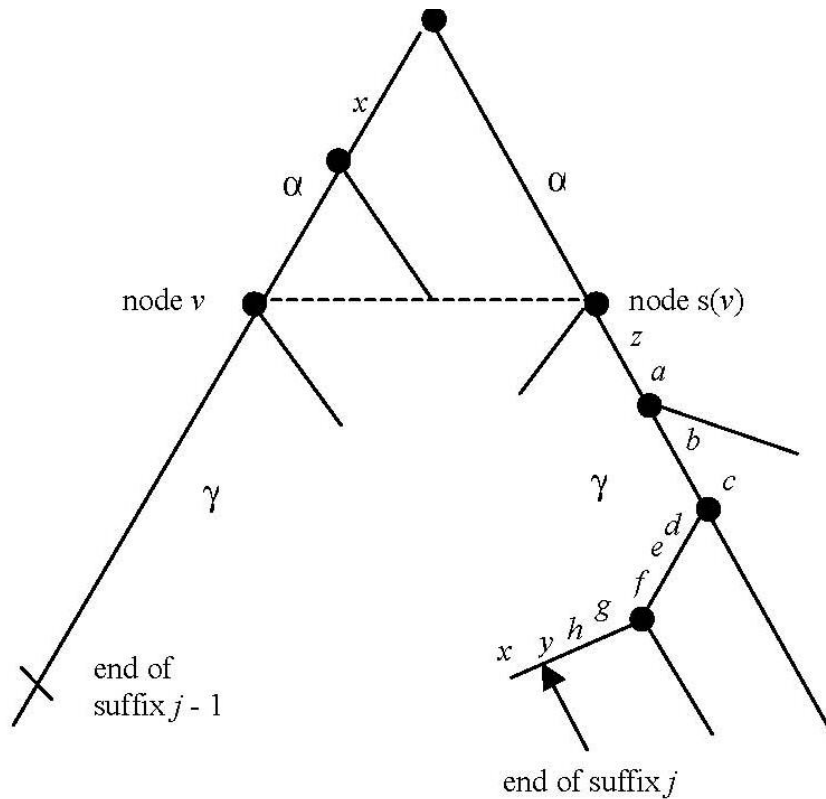


Figure 3.11 Illustration of Skip-count trick process

3.5 Algorithms

We discuss various types of algorithms and techniques from literature which have been used to solve similar problems in the following sections.

3.5.1 Suffix Tree Algorithms

Mreps is a tandem repeat searching algorithm implemented and developed by Kolpakov et al. [73]. The algorithm is able to detect all patterns in a string of size n in time $O(n)$ it is based on Lempez-Ziv factorization and heuristics. The strength of the algorithm lies in its relative speed, support for fuzzy repeats and has various parameters to control the execution. The execution control is possible through the use of the *resolution parameter*, which is used to control the fuzziness of the repeats. The algorithm implementation is publicly available online. The major limitations of this algorithm is that it cannot operate

on the entire human genome due to its high memory requirements since it uses an index-like structure also adopted from the Suffix Tree.

Stoye and Gusfield [74], proposed an algorithm which runs in $O(n \log n)$ time to produce all tandem repeats occurrences and in $O(n)$ space. The major strength of this algorithm is that, it is relatively faster as the Mreps, however it does not scale long sequences. This drawback is still based on the idea of Suffix Trees as its data structure, and no technique has been applied to reduce the Suffix Tree memory requirements. A Suffix Tree has high memory requirements; hence it is impractical to meet the memory requirements of this algorithm.

Abouelhoda [75], introduced a new variant data structure called the Suffix Array as a way to improve Stoye and Gusfield algorithm limitations of the Suffix Tree data structure. The program is called Vmatch, which implements their algorithm. It improved the space requirements of Stoye and Gusfield algorithm but was not sufficient enough considering that the big size of the complete DNA sequence which is around three billion base pairs. The suffix array data structure could not easily support other operations which are possible on the Suffix Tree such as inexact motifs, inference motifs and so forth [74, 76]. It is also fascinating to note that reduction of the space consumption was done in the similar time complexity as the one for Suffix Tree construction. The reduction was few bytes depending on the application at hand (20 to 5 bits). However, this reduction was considerably big in large sequences.

Sputnick [77] introduced an algorithm that uses combinatorial approach and search only for microsatellites. This recursive algorithm uses more like a sliding window approach by scanning through the sequence. The limitation for this algorithm is that it supports only searches for microsatellites, but it has been used in many research projects because of its fast performance and less memory requirements.

Manzini and Feragina [65] introduced an algorithm for creating a Suffix Array, using a technique called *deep - shallow sorting* which made use of a *deep* sorter for long common suffixes and *shallow* sorter for short suffixes. Most proposed construction algorithms for suffix arrays are inefficient when applied to a sequence with many repeats. Their

algorithm managed to overcome this dichotomy. Their algorithm has a time complexity of $O(n \log n)$ in the *worst case* scenario and with a space complexity of $O\left(\frac{n}{\sqrt{\log n}}\right)$ including input text and the data structure.

Another SA construction algorithm was presented by Valimaki et al. [16]. The program used the (CST) Compressed Suffix Tree. One main issue with CST is the high space and time requirements during its construction. For instance, for the whole DNA sequence, the construction takes 4 days, and the final CST takes approximately eight and half Giga Bytes and the maximum memory usage of about 24 Giga Bytes. This construction time was run on a 32 Giga Byte-memory machine. This shows the impracticality of the CST data structure. We improve this performance basing on truncating the generalised Suffix Tree.

Fast BWT algorithm uses block-wise suffix sorting and was introduced by Karkkainen [78]. The algorithm makes use of suffix arrays and BWT. The BWT is built from a Suffix Array and inherits the weaknesses of the data structure. However, they got rid of the full suffix array and computed the BWT. The process of BWT for a sequence of size n uses $O(n \log n + vn)$ time with $O\left(n \log \frac{n}{\sqrt{v}}\right)$ bits on memory consumption with the sequence and the BWT. It was found to be 2-3 times faster and with a construction of 1GB/hour better than the 300–400MB/hour on a PC than the one discussed above. However, this algorithm was never experimentally tested hence not implemented to be able to benchmark its actual performance. The algorithm was focussed only on the construction time and space requirements for the Suffix Array and not the complete algorithm for repeat finding.

Fischer et al. [79] proposed an algorithm used for pattern searching and uses the Compressed Suffix Array and LCP array. This algorithm outputs all the patterns of a sequence by specifying settings for parameters and input. It was implemented and tested on a 3.0 GHz CPU and 128 Giga Bytes of internal memory. The program was able to retrieve the whole *homo sapiens* genome in 39.8 hours (not including time for output operation and displaying the patterns) and used memory of 9.3 GB. We improve this running time for construction our data structure.

Kulekci et al. [80] algorithm employs a different data structure called the Wavelet Tree. This structure does not support all the string operations supported on the suffix tree and therefore, will not be considered for this study. It also uses the Suffix Array to generate an LCP array. This algorithm has been tested on desktop PC with 8GB main memory and 2.8 GHz four-core Intel@CoreTM i7-860 chip and took 17 hours to complete detecting all the patterns in the entire DNA sequence.

In conclusion, all algorithms discussed above still leave gaps of producing a reasonable running time on a desktop PC and this calls for continuous research to improve the repeat searching algorithms.

3.5.2 KMER-based Suffix Algorithms

We review the existing algorithms for construction of algorithms which employed the KMER technique. In simple terms, the KMER technique works by constructing a truncated Suffix Tree to a depth equal to the pattern size to be searched.

There are only three algorithms which use this technique of KMER for Suffix Tree construction according to the best of our knowledge. The first one, was introduced by [17]. In which they proposed a data structure which is a modification of the Suffix Tree called the truncated Suffix Tree (TST). The TST uses a sliding window technique of fixed size and is based on McCreight's [56] and Ukkonen's [72] algorithms. The TST is more useful when used with the LZ77 for its compression. Our proposed algorithm is not based on Ukkonen's algorithm but on a new technique aimed at reducing space requirements. For instance, a 3-TST of string $S = \{abaabaa\}$ is constructed as shown in Figure 3.12

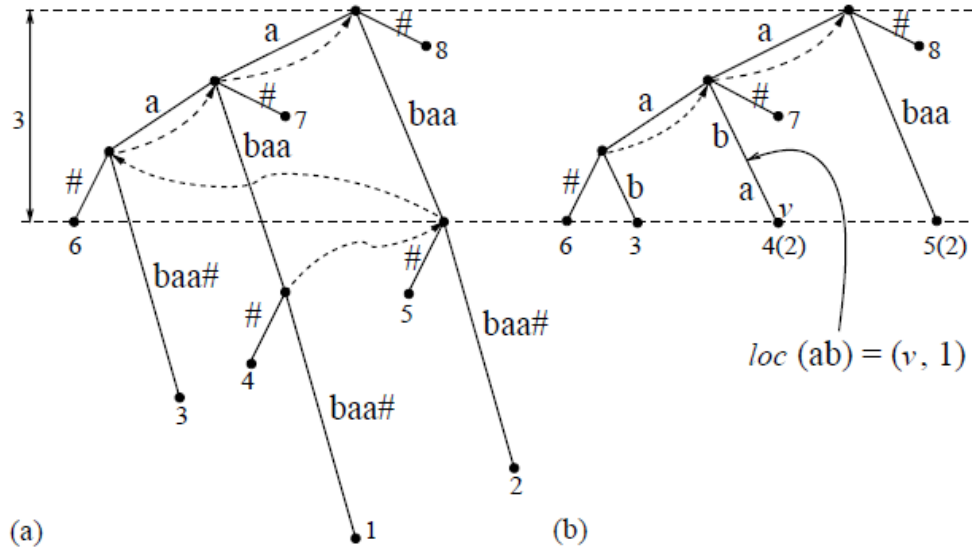


Figure 3.12 (a) Suffix Tree (b) 3-TST of abaabaa# [17]

The results recorded for the experiments conducted for the k -TST algorithm used input files ranging from 400KB to 800KB. The file sizes are comparatively smaller than those of genomic sequences. Perhaps this is due to the fact that the algorithm was designed for LZ77 compression contrary to our proposed algorithm which focuses on genomic sequences, which are huge. For instance, a file (728KB) with fixed window size of $k = 3$ have space requirements of 12,713KB (≈ 13 MB).

Schulz et. al. [18] presented an algorithm called k -Truncated Suffix Tree(k TST), which uses a similar technique as the TST algorithm. The k TST algorithm is derived from Ukkonen algorithm and works by building a k -deep tree that provides quick searches for small patterns of upto k characters long. This algorithm provides benefits in both memory usage and computational time especially in DNA or protein sequences. For instance, given the two strings $S_1 = \{ATAT\}$ and $S_2 = \{TAT\}$ [81], the k TST is constructed as shown in Figure 3.13, for $k = 3$.

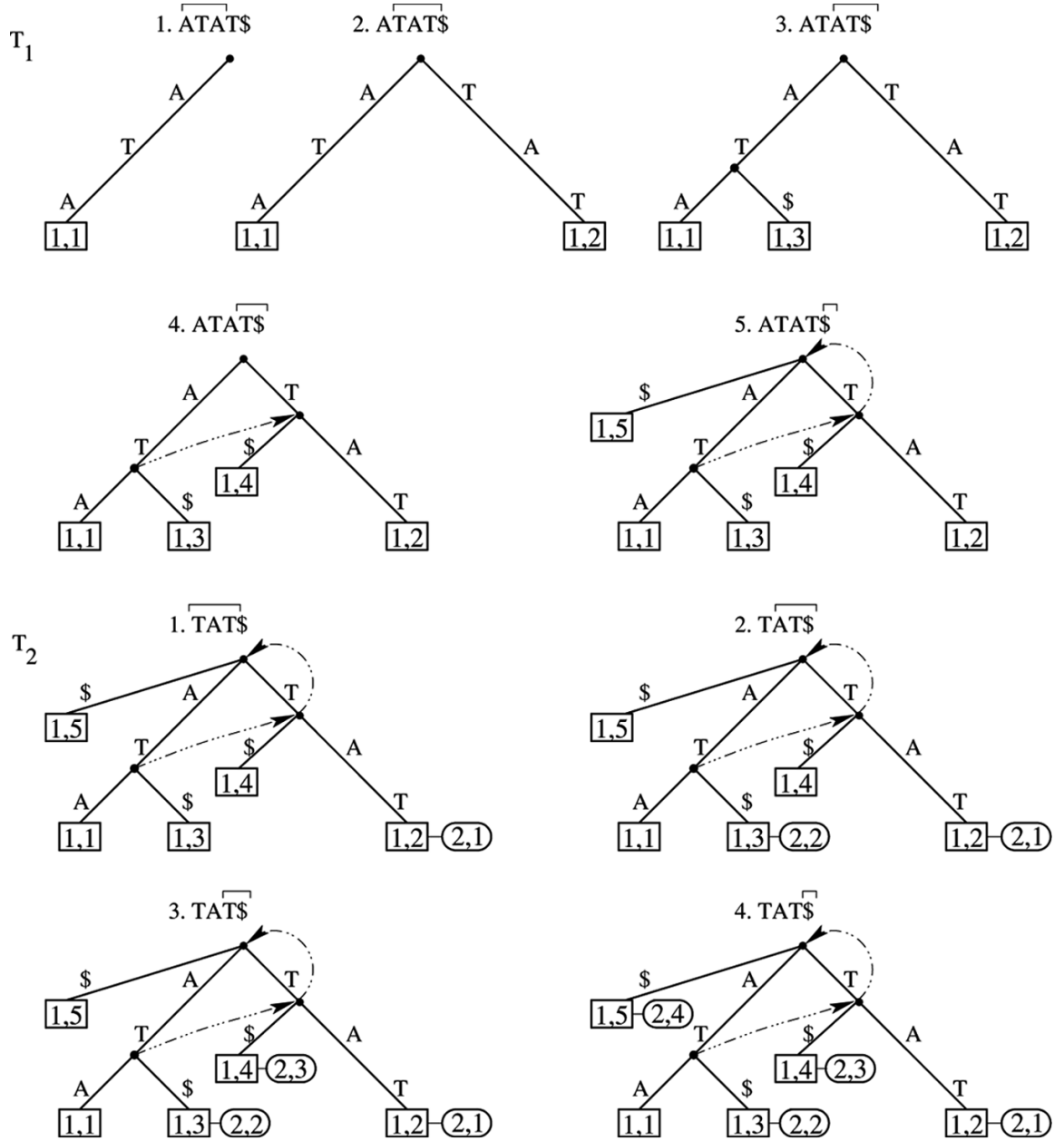


Figure 3.13 Construction of the generalised kTST [18]

The results obtained from the $kTST$ algorithm used varied input sizes from 0-10,000 sequences. For instance, the 10,000 sequences took 30 seconds to construct the TST with $k = 11$ and approximately took 1GB of memory for its construction.

Allali and Sagot [82] also introduced a variant data structure similar to the Suffix Tree called *k-factor tree*. The *k-factor tree* indexes all the elements of a string with a length not exceeding k . The data structure offers several advantages over the Suffix Tree for small values of k such as improved space complexity. The creation of the *k-factor tree* is based on the idea from Ukkonen [72]. The *k-factor tree* does not use fixed window or

strings but varied sizes upto k as shown in Figure 3.14. The extraction of substrings is represented by horizontal lines.

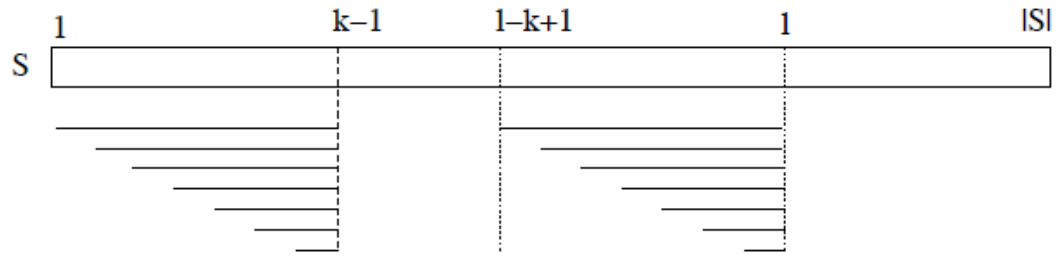


Figure 3.14 k-Factor tree extraction of substrings [82]

The results for the experiments which used biological sequences for this algorithm shows that a file of length 87164908 took time of 342.30s and performed better than the ordinary Suffix Tree by 7.35s for $k = 20$.

3.5.3 Parallel programming

Parallel execution allows us to run computationally intensive tasks in less time compared to sequential execution making it an attractive choice for such tasks. The challenging task is breaking down the computational task into parts which have no dependency on each other. Different processes or threads are created to execute each specific part however there are few challenges to this parallel approach:

- Breaking down a task into smaller parts which are not dependable is not a trivial task. The processes for each part might finish at different times and the part size may vary.
- Merging of results from different parts to come up with the final results adds additional overhead of processing
- Completion of sub-tasks (parts) is hugely dependent on the capacity of the machine

However, these challenges can be overcome by using MapReduce, a new programming model, which allows you to distribute computation across a cluster and the final results computed from the distributed computation [83]. Figure 3.15 shows how MapReduce works. In this study we are not going to apply MapReduce since we want to manually explore and control our algorithms' performance so we will manually assign the

processes to sub tasks. Furthermore, our algorithm is alphabet dependent, hence our computational sub tasks are known beforehand.

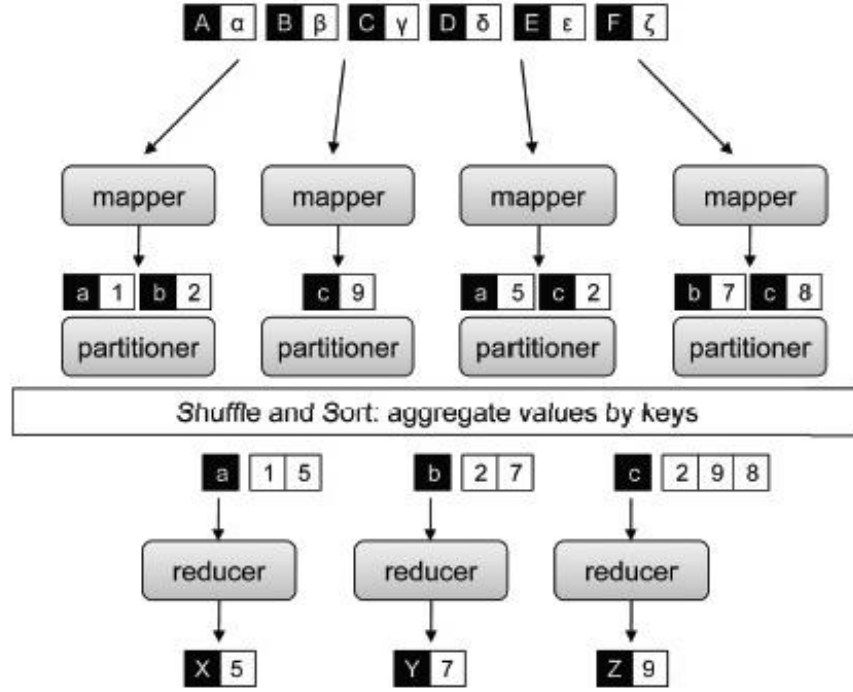


Figure 3.15 MapReduce Operation [1]

MapReduce model uses the Hadoop framework to execute computationally intensive tasks in parallel. The storage is shared and allows any type of data to be stored and provides recovery services. Hadoop also runs on relatively low cost hardware [1].

3.5.4 Parallel algorithms in Suffix Trees algorithms

In this section we discuss the critical review of the parallel algorithms used in the area of construction of Suffix Trees. Some work related to parallel construction of Suffix Trees has been done before, though most are theoretical [84, 85]. Apostolico et. al. [86] introduced the first non-serial algorithm for creation of Suffix Trees. The algorithm uses a technique called *naming* which is based on the doubling approach.

The algorithm has a time complexity of $O(\frac{1}{\epsilon} \log n)$, $O(\frac{n}{\epsilon} \log n)$ work and $O(n^{1+\epsilon})$ space using CRCW PRAM Model and with parameter values for ϵ of $0 < \epsilon \leq 1$. The algorithm is built on the CRCW Parallel RAM model. The construction algorithm constructs the Suffix Tree of a sequence of n characters with a time complexity of

$O(\log n)$ on n processors. The algorithm works more efficiently on all other types of alphabet except the binary alphabet.

Hariharan [84] proposed another improved parallel algorithm. This was the first parallel Suffix Tree algorithm which was work-optimal. The algorithm is based on the CREW Parallel RAM model, and has a complexity analysis of $O(\log^4 n)$ time, $O(n)$ for work and space complexity of $O(n)$ for a Suffix Tree of string S of length n for a fixed alphabet, such as binary strings. The algorithm achieves in $O(\log^4 n)$ time and $O(n \log |\Sigma|)$ for work and space complexities for a general alphabet where $|\Sigma|$ is the number of unique characters in S . Landau et. al. [85] also proposed a theoretical algorithm that executes in $O(\log n)$ parallel time using n cores. Practical details are not given for this algorithm.

Chen and Schmidt [87] proposed a Suffix Tree construction parallel algorithm which works on a computational grid. Suffixes are divided into fixed length parts by using a technique they called *prefix-based* technique. Construction of each Suffix Tree is done on each node using a greedy load balancing **algorithm**. The construction is based on Ukkonen's algorithm. To the best of our knowledge, no practical implementation of the algorithm was recorded for this algorithm.

Carvalho et. al. [88] presented an estimated partition algorithm with load balancing. This algorithm was addressing the problem of balanced partitions of an alphabetic trie when extracting structured motifs. This algorithm cannot be applied for cache-partitioning since it does not guarantee partition sizes produced.

Tsirogiannis [89] proposed an algorithm to address the weakness of Carvalho et. al. [88]. It is based on Chip-multiprocessors (CMPs). Their algorithm is cache-conscious to improve the cache performance and takes advantage of on-chip parallelism on CMPs. The illustration of the partitioning processing is shown in Figure 3.16. The algorithm also used various compression methods to further improve space requirements and ultimately trades space for cache performance. However, the algorithm targeted more on cache performance rather than space as in our objective for this study.

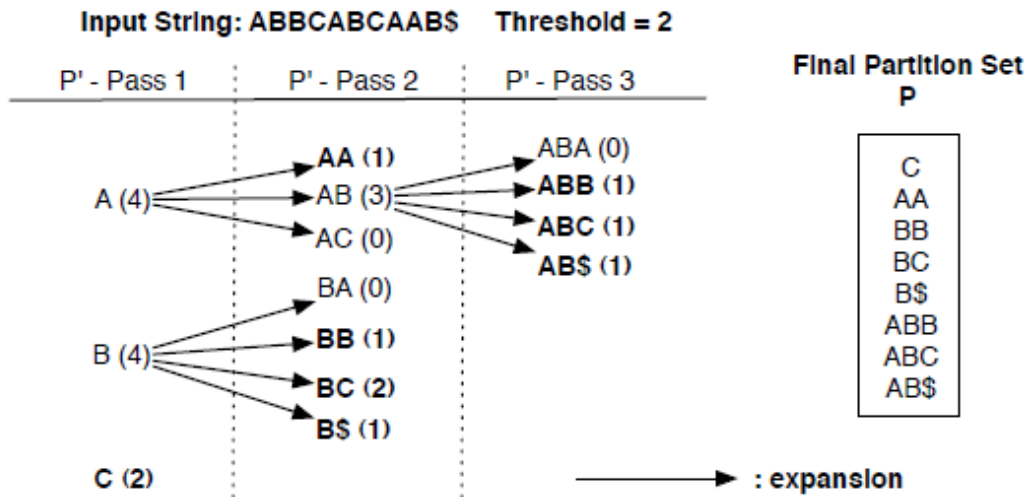


Figure 3.16 Prefix Partitioning

Recently, Satish et. al. [58] presented their practical parallel algorithm based on the MapReduce programming model. The algorithm vertically partitions the Suffix Tree and constructs the resulting sub-trees independently in parallel by modifying Ukkonen algorithm as shown in Figure 3.17

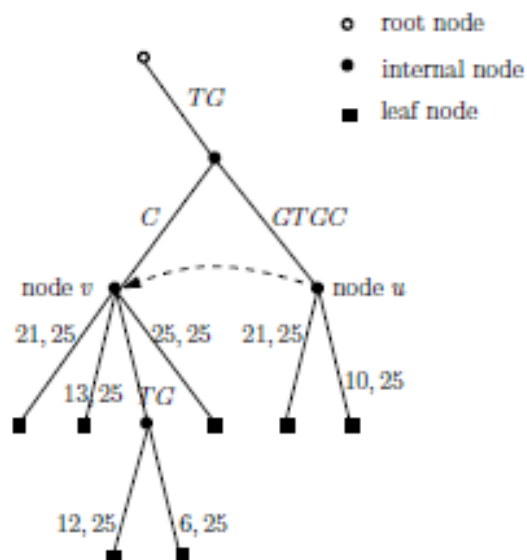


Figure 3.17 Sub-tree for S-prefix T and sample contextual suffix link [58]

As far as practical implementations of the Suffix Tree construction algorithms we consider the most relevant two, which are, Wavefront [90] and Elastic Range (ERA) [91]. The Wavefront splits a string into partitions and constructs the sub-trees. The partitioning process uses variable length prefixes ensuring that each partition begins with similar prefix. This guarantees that the independent sub-trees can be merged. Wavefront makes

use of sequential reads on the string to reduce the I/O cost. It's unfortunate that the code is not available because of the IBM policy. ERA presents a new approach which divides the sequence both horizontally and vertically. Vertical partitioning divides the Suffix Tree into smaller-trees which fit into the main memory using the variable length prefixes like the one done in [91]. ERA further combines the sub trees to share the input/output cost for the input string access. Then horizontal partitioning is done independently for each sub-tree in a top-down approach. Comin and Ferrares [92] proposed their algorithm which is a parallel Suffix Tree called Parallel Continuous Flow (PCF), which is designed specifically to work with long. The design of the algorithm is based on distributed memory model thus independent nodes. The algorithm is based on the idea of generating the tree from suffix array and the LCP. This conversion happens in linear time. Then the data dependencies are analysed as shown in Figure 3.18. The results show that PCF algorithm scales to the size of the input string as it grows, and was able to index the entire human genome in seven minutes with 172 nodes/processors. The study uses a shared memory model contrary to ERA algorithm. The choice is based on the fact that we take advantage of time requirements any way possible.

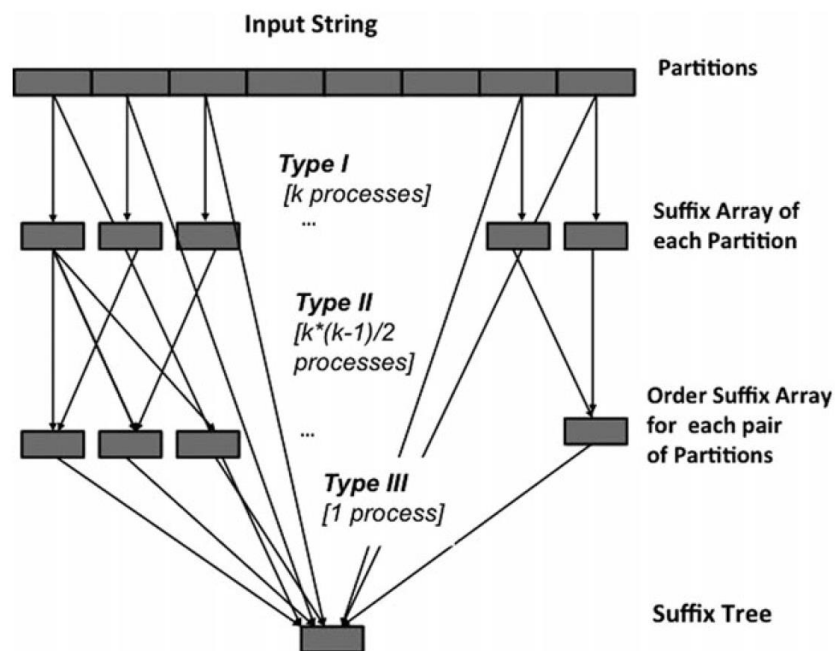


Figure 3.18 A workflow of data dependencies of PCF [92]

Most parallel algorithms in the past were theoretical, however, this algorithm proposes a practical algorithm which can be used to benefit the science community and for benchmarking purposes. All parallel algorithms from the past are based on the Suffix

Tree and not on the KMER based Suffix Tree. This study, therefore, presents the first algorithm on KMER based Suffix Tree.

3.6 Summary

The chapter covered selected literature which is related to this study. This included literature review of various algorithms and techniques. The discussion included the critical review of various algorithms and data structures. Further, techniques to be incorporated in the proposed algorithms. we also illustrated.

Chapter 4

KMER algorithm

4.0 Introduction

In this section, algorithm design is outlined, using the KMER approach. This research follows a formal methodology. We start by providing preliminaries, that is, notation used in our algorithm design followed by our algorithm in pseudo-code. The main objective of our KMER algorithm is to improve the challenge of high memory requirements in Suffix Trees. The KMER algorithm is a serial algorithm hence it does not intend to improve the time of the generalized Suffix Tree construction. This also forms part of our second research objective of improving the space component.

The justification for the choice of KMER algorithmic approach is based on the fact that previous related algorithms have been shown to perform better in terms of memory consumption [17, 18] than other approaches such as the one by Valimaki et al.[16], which uses a complete tree. Furthermore, the aim is to explore a solution which is able to run on standard desktop computers which usually have limited memory comparative to super computers and data servers. This approach, therefore, is a perfect fit providing a work around solution or a trade off with using standard computers with truncated Suffix Trees in conjunction with the sliding window technique.

4.1 Preliminaries

Let S be a string with a finite ordered alphabet of Σ . $S = s_1s_2s_3 \dots s_n$ are symbols in a string from Σ and $\$ \notin \Sigma$, where $\$$ is the terminal symbol. S has a substring $S_{i\dots j} = s_is_{i+1} \dots s_j$ where $1 \leq i \leq j \leq n$. A pattern can be defined as a substring over the string S . A pattern $P = p_1p_2p_3 \dots p_k$ appears at index j of string S , if and only if (iff) $p_1 = s_j, p_2 = s_{j+1}, \dots, p_k = s_{j+k-1}$. Given j satisfies $1 \leq j \leq n$, a prefix of S is any string $S_{1\dots j}$. Also given $1 \leq i \leq n$ [16], k is used to represent the length or size of the pattern so KMER refers to a pattern of size k . In this thesis we define a pattern as a substring to be searched in a string. A tree for a sequence δ is defined as a rooted tree with edges labelled with non-empty substrings of δ . Vertices are used to represent suffixes and the root labelled with an empty string. Each substring α can be represented as (β, γ) , where β node represents the prefix of string α and γ is the suffix of string α . Figure 4.1 illustrate the process of construction of the Suffix Tree from oo to ook .

We illustrate the sliding window technique on Figure 4.3

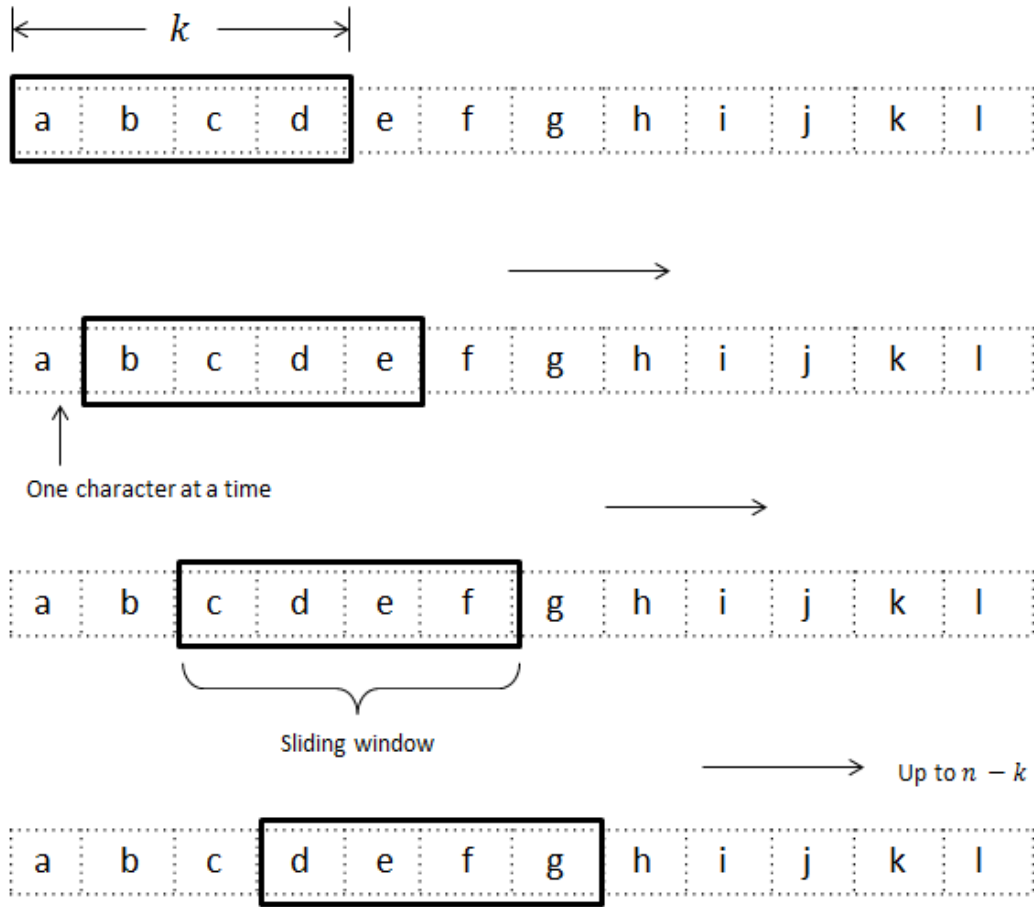


Figure 4.3 Sliding window method

So we have $(n - k) + 1$ substrings for every sequence of length n . In our design for the generalised tree construction, we use sliding window technique to help us extract fixed substrings only. After the extraction phase, substrings are added to the Suffix Tree. Our method for adding the substrings to the tree is different from other previous approaches [17, 18]. The previous approaches further generate suffixes from the substrings before construction, thereby increasing the memory requirements. For instance, given a substring $s = \{acgt\}$, suffixes of s are generated to give $S_{suf} = \{t, gt, cgt, acgt\}$. S_{suf} is used to add these suffixes to the Suffix Tree. In our case, we do not further generate suffixes from the substrings; rather only add s as it is. This creates memory savings and guarantees retrieval of all patterns of size k , provided k is known beforehand. The resultant tree from the previous researchers' approach would create a Suffix Tree with varying edge lengths whereas ours create edges of similar length.

4.3 KMER algorithm Design

This is the key area in our work from which our ideas will be based [93]. We start by discussing our algorithm for Suffix Tree construction by making use of an example in Example 4.1. Specifying the pattern size (k) is required to initiate the algorithm for the Suffix Tree construction. The next stage includes employing the sliding window technique as illustrated in Figure 4.6. The sliding window technique constructs the tree by shifting one character at a time using a window size of k . We use Example 4.1 to explain the concept in our algorithm.

Example 4.1

Given a sequence $S = \{ACGTCCTGG\}$ with $k = 4$, where k represents the pattern size and $n = 10$, where n represents the number of characters in S . $n = 10$ since we first append the terminal symbol $\$$ before any initial step of our algorithm. Our algorithm starts by retrieving the value of k , which is, in this case four. Our algorithm then uses the sliding window technique by sliding the window of size k on S character by character until $n - k$ position as shown in Figure 4.6. Each window in the sliding window process is a fixed substring. A set of all substrings generated from the sliding window process is created. S_{sub} represents a set of the substrings, hence, $S_{sub} = \{ACGT\$, CGTC\$, GTCC\$, TCCT\$, CCTG\$, CTGG\ \$\}$. The generalised Suffix Tree is constructed from S_{sub} . The Suffix Tree is constructed progressively substring by substring until the tree is complete. The complete tree has a maximum fixed height $h = 4$ which is k . Every edge in the tree from root to leaf is of length 4. The complete generalized (KMER) Suffix Tree is displayed in Figure 4.4.

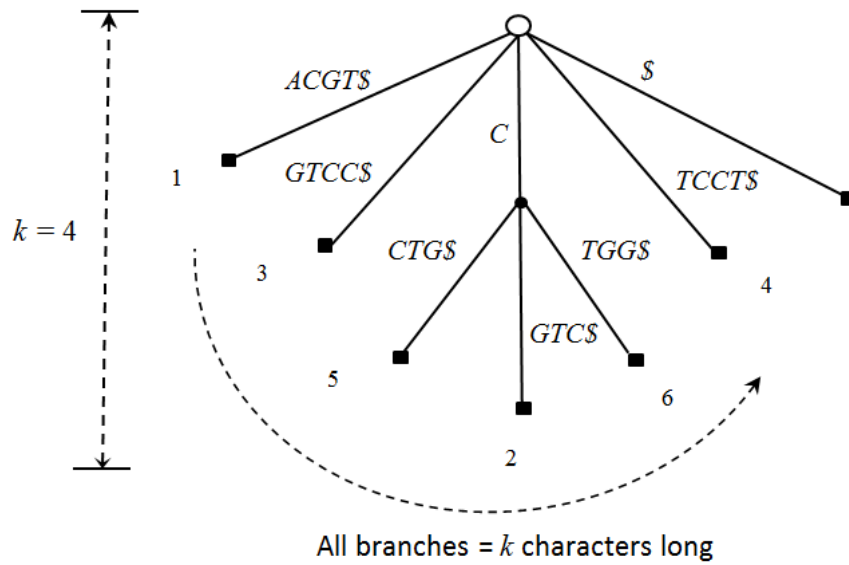
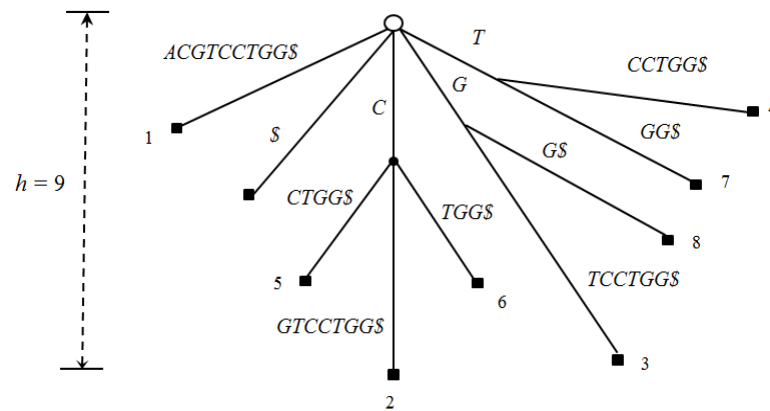


Figure 4.4 KMER Suffix Tree of S

Any pattern of size k can be located in $O(|p|)$ time, p represents the pattern to be searched. Note that, the final tree size has been reduced tremendously to a maximum fixed height of k . This reduction in size means reduction in space requirements for this tree as compared to the complete Suffix Tree in Figure 4.5. The complete Suffix Tree shown in Figure 4.5 has a maximum height of 9 with varying depths on different edges thereby increasing the space requirements of the Suffix Tree. The benefit of the complete Suffix Tree however is the ability to retrieve patterns of varying sizes. The KMER- based Suffix Tree will only be able to retrieve the patterns of a fixed size as specified before its initial construction of the time. This means the input of the patterns to be retrieved should be known beforehand, whereas with the complete Suffix Tree, the input size of the patterns to be searched is not needed.



All branches vary in height with maximum of $n = |S|$ characters long

Figure 4.5 Complete Suffix Tree

The completed Suffix Tree in Figure 4.4 is called the KMER-based Suffix Tree since the tree is based on the size of k and constructed using the modified sliding window technique. Therefore, a KMER-based Suffix Tree of size 5 for instance will be referred to as 5-mer based Suffix Tree.

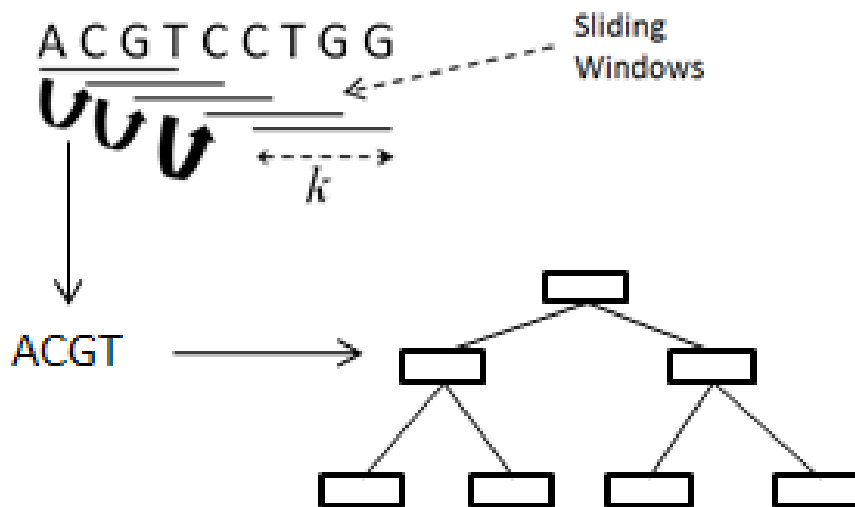
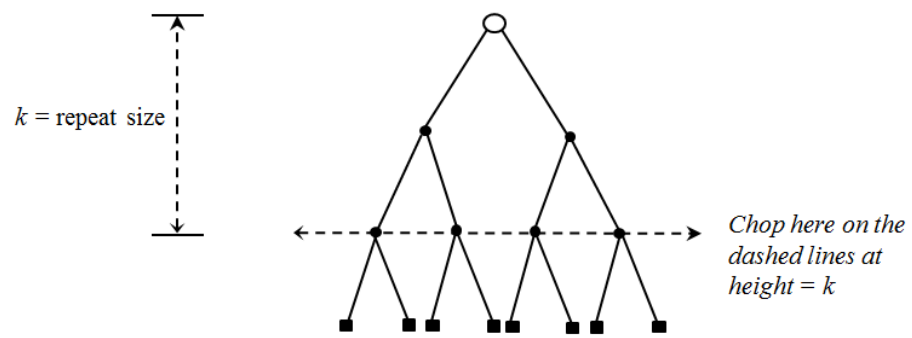


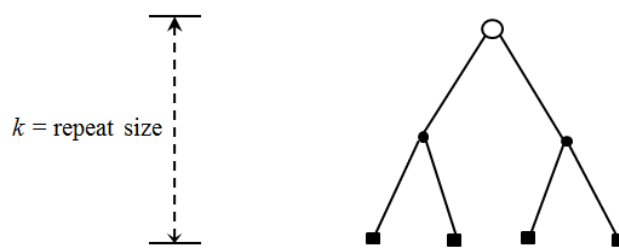
Figure 4.6 Sliding windows technique illustration as fixed substrings

Figure 4.7 Part a, attempts to illustrate our idea in a diagram. The ordinary Suffix Tree will always be bigger compared to our truncated generalized Suffix Tree, since our tree can only go up to height of k . The standard Suffix Tree goes up to size n (n represents the size of the sequence or string) with varying heights on different edges. Our algorithm represents a standard Suffix Tree which is chopped or truncated but with the same functionality or ability to retrieve all patterns of size k , as shown in Figure 4.7 Part b.

We now explain our algorithm, that is, Algorithm 2. Our algorithm is called KSA for KMER Search Algorithm. We have presented our algorithm in pseudo-code since it's the standard way of representing algorithms. The presentation is high-level, it leaves out a lot of unnecessary low-level implementation details. The KSA algorithm is shown in Figure 4.8. The algorithm starts by retrieving and setting the value of k which gives us the pattern size. All other variables or parameters are all set accordingly for later use (lines 1 – 5). The next step of our algorithm creates the sliding window of size k upto $n - k$ index. Each generated k -substring (*sub*) is entered into a set of k -substrings. The generated fixed k -substring is used to built the generalized Suffix Tree by inserting it onto the tree node by node with the procedural call of *insert* method (lines 6 – 10). The *insert* subroutine adds the suffixes to the generalised Suffix Tree up to $|sub| - 1$ steps. It is important to note that, as k -substrings are added to the tree, the suffixes of substrings are not included and this makes our algorithm unique and different to others proposed before, such as [17, 18] (lines 11 – 14). The last part of KSA algorithm outputs the complete KMER based generalised Suffix Tree (line 15). The constructed KMER based generalised Suffix Tree is of k –height and not n –height which roughly gives our Suffix Tree huge $n - k$ savings.



Part a



Part b

Figure 4.7 Part a) Standard Suffix Tree vs. Part b) KMER-based Suffix Tree

Algorithm 2. Algorithm k -mer Search Algorithm (KSA)

Input : S, n, k

Output: A generalized suffix tree representation ST of S of height k

```
1  $ST := \{\}$ 
2  $n := S.len$ 
3  $k := K$  //size of the repeat
4  $S_{sub} = \{\}$ 
5  $sub := []$ 
6 for  $i \leftarrow 0$  to  $n - k$  do
7      $sub = S[i, i + k - 1]$  //sliding  $k$ -size window
8      $S_{sub} = sub$ 
9      $ST.insert(sub)$ 
10 endfor
11 insert:
12     for  $j \leftarrow 0$  to  $|sub| - 1$  do
13          $add.node[sub[j]]$ 
14     endfor
15 return  $ST$ 
```

Figure 4.8 The KSA Algorithm

The complete research approach the KSA algorithm employs can be shown in a high level view in Figure 4.9. Figure 4.9 illustrates the complete chain, that is, from the point the DNA sequences are generated to the last part where our algorithm comes in before the pattern search process starts.

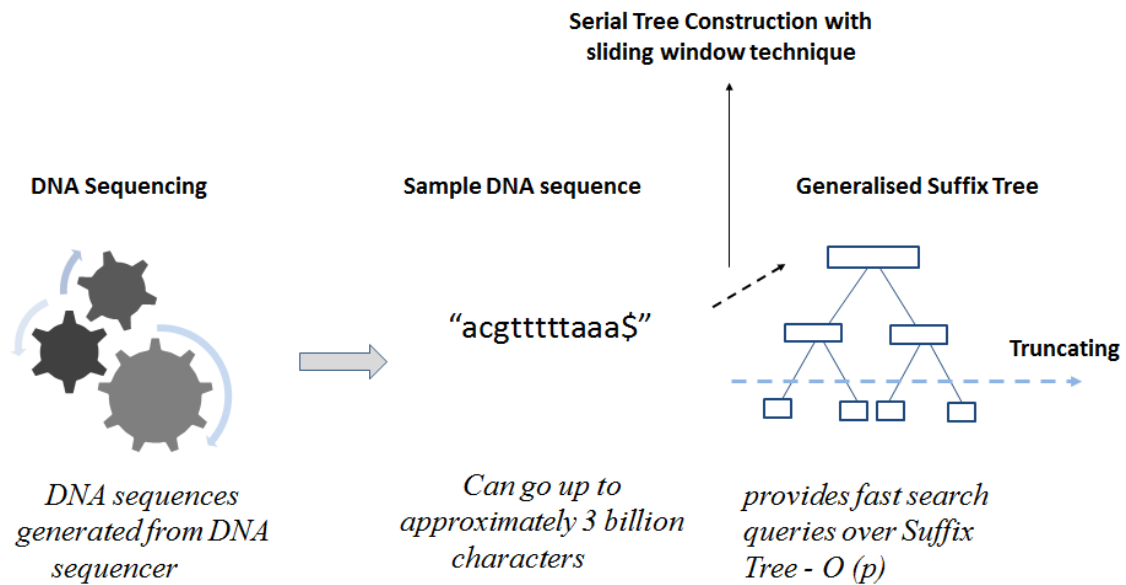


Figure 4.9 Design summary

4.4 Evaluation strategy

A triangulation approach is followed to make a comparative analysis of results obtained from the proposed algorithm [94]. This was done using experimental and theoretical analysis. The rationale is that these two approaches complement each other's limitations. For instance, complexity analysis does not give the full details of algorithm's performance, whereas the experimental analysis gives actual running time. Mathematical proofs were applied to validate complexities and concepts. Section 4.4.1 and 4.4.2 describes how theoretical analysis and experimental analysis were conducted respectively.

4.4.1 Theoretical Analysis

Theoretical analysis does not give a complete analysis of algorithms' **performance**. This is because it is an approximate metric. However, it provides an indication without bias of the underlying hardware, compiler or programming constructs used. There are basically two ways to characterise analysis of an algorithm, which are, time and space complexities. Another advantage of applying complexity analysis for evaluation is that, it is quantifiable.

In general, an algorithm is meant to create a bridge between the programmer and hardware. There are three ways to classify analysis of an algorithm, and these are, best case, average case and worst case complexities. Worst case is defined as the most number

of operations required in an algorithm. Average case is the normal or basic number of steps taken, and best case is the least number of steps taken for an algorithm to complete. In most cases, it is the worst case complexity that matters most, this is because it indicates the performance of an algorithm in its worst case scenario as a function of n , where n is the string size. For this reason, focus was on the worst case complexity analysis in the study. Both time and space complexity were calculated, basing on the Random Access Machine (RAM) model. This model was used because it is a widely accepted standard for analysis of serial algorithms.

4.4.2 Experimental Analysis

The experimental analysis provides actual performance of the executable version of an algorithm. However, it has the limitation of depending on many factors, such as, the programmer's skill, underlying hardware, choice of compiler, and so forth. There is currently no method for getting accurate performance predictions in one programming environment based on observations of execution times in another. The experimental analysis gives us more control over the trade-off between accuracy and generality when calculating or measuring predictions about performance. Recently there has been a growing interest in experimental analysis in the field of theoretical science. This has been due to the increasing recognition that theoretical results cannot give a full picture about the real world for algorithmic performance [95]. Most funding organisations now support this with the view to providing a path from theory into practice. We will conduct comparative analysis with python's version of Mark Carlson based on Ukkonen algorithm Suffix Tree construction [96]. Mark Nelson has a C++ implementation; however, the Python version is used since the development is also in Python. The following subsections: 4.4.2.1, 4.4.2.2 and 4.4.2.3 provide implementation details of the algorithm and the environment the experiments were conducted in.

4.4.2.1 Software tools

We implemented our KSA algorithm using Python 3.5.2 and Anaconda 4.1.1 for the experiments. Python is a very popular and powerful language in data science and has a growing adoption in the scientific community. The reason we used python is because it has powerful support libraries for the Suffix Tree data structure and has a very clean and compact syntax [97]. We also used the Anaconda distribution because it comes with more preinstalled packages for handling large amounts of data such as the DNA genome.

4.4.2.2 Test bed (Machine Specifications)

The experiments were conducted on HP Z820 Workstation with a 3.30 GHz sixteen-core Intel Xeon E5-2623 chip with 10 MB L3 Cache. The workstation is a multi-core and we did not apply any parallelism intentionally since the objective of the KSA algorithm was on reducing space requirements. The workstation is installed with Ubuntu 14.04LTS (64-bit) operating system with 32 GB main memory, swap space of 64 GB, and 1 TB Serial ATA Hard Disk Drive at 7,200 RPM. We opted for a machine with bigger main memory because the construction of a Suffix Tree requires a lot of memory. In other previous studies the peak memory reached 24GB [16].

4.4.2.3 Dataset description

Biological datasets were used to test space usage and usability of the algorithm. Genomic data were used because it is the real world data from which our algorithm is going to be used for. We used the chromosomes, the entire human genome, protein sequences and the rice chromosomes. All the DNA data were available in FASTA format which is the standard format for representing nucleotide sequences. The whole point was to use different dataset from different species so we can have a complete understanding of our algorithms' performance. The unique reason why we chose specifically Rice chromosome 12 was because it is the only chromosome with highest number of repeats for this plant.

The list of data sets used is provided below:

1. The human X-chromosome sequences sourced from NCBI [98].
2. The entire human genome sequence (Homo Sapiens) [98].
3. Protein sequences sourced from Pizza&Chili corpus (Gene DNA Sequences).
4. Rice Chromosome 12 (Genbank Accession AP008218)

We provide more information on the characteristics of our dataset in Table 4.1.

Table 4.1 Characteristics of dataset

No.	Dataset	Size (MB)	No. of Symbols $\approx$ (million)
1	X-chromosome	153,798	150
2	Complete Human Genome	3,200	3,000
3	Pizza&Chili corpus	200	200
4	Rice Chromosome 12	27	28

4.5 Summary

In this chapter we moved from understanding the memory drawback of the generalized Suffix Tree to a proposal of a data structure, which is a variant of the Suffix Tree called KMER-based generalized Suffix Tree. The algorithm for construction called KMER Suffix Tree (KSA) has been presented. The core operation of the KSA algorithm called sliding window technique has been discussed, together with its differences to the existing studies.

Chapter 5

PaST algorithm

5.0 Introduction

The last decade saw a new standard of computer architectures, that is, multicore architecture in desktop computers [23, 99]. The same period also saw a huge improvement in the availability of DNA sequences however analysing these DNA sequences demand a lot of processor power, creating a challenge on DNA analysis algorithms. This chapter introduces an algorithm which explores this performance challenge by taking advantage of the multicore revolution. To accomplish this, various algorithms and tree data structures will be modified to suit the multicore architectures [99]. We introduce a novel alphabet-dependent algorithm called Parallel Suffix Tree (PaST) algorithm which is KMER-based, that is, we provide a parallel solution to the KMER serial algorithm (Chapter 4) to improve the time component of Suffix Tree construction. This achieves the third objective of the research.

The claim that parallelisation of the serial algorithm will improve time requirements of the PaST algorithm is based on the fact that if more processors are programmed to work on divided tasks of the whole task, the total execution will be reduced. This is further mathematically proven in Lemma 1, shown in Example 5.1 and Amdahl's Law to validate the claim. Furthermore, parallelisation has been shown to improve performance of serial computing tasks or algorithms [100].

The research approach follows formal methodology. We start by providing preliminaries, that is, notation used in the algorithm design followed by the algorithm in pseudo-code. The purpose of the PaST algorithm is to produce a time-efficient algorithm for the KMER based generalised Suffix Tree construction.

5.1 Preliminaries

In this section, we introduce basics of the notation used in our algorithm and its evaluation. Performance gained in the proposed is calculated using the *speedup* metric. The speedup of an algorithm is $O(T_1/T_\infty)$, where T_1 represent time taken on 1 processor and T_∞ represents time taken on multiple processors. Concurrent Read and Exclusive Write (CREW) model will be used in our algorithm [101]. This model allows concurrent

read and exclusive write. The CREW Model is going to be used based on the PRAM model. Another model which can be used is Work-Depth Model. However, the PRAM Model can still be converted to Work-Depth Model using Brent' scheduling theorem, so either way will give us a similar result [101].

We introduce three concurrency keywords to show parallel execution, which are, *spawn*, *sync* and *parfor* as outlined in [101]. The three concurrency keywords are used to express logical parallelism, that is, the elements of the computation which should be run in parallel. The *parfor* construct is used to show a for-loop which is executed in parallel fashion. *Spawn* shows that a procedure call which precedes *spawn* will be executed in parallel. Lastly, *sync* indicates all *spawned* procedures that must complete before it executes in parallel.

We represent a string of size n by $S = S[0], \dots, S[n - 1]$, where $S[i]$ denotes the i the character in S . We represent an alphabet by $\Sigma = \{0, \dots, \sigma - 1\}$, σ represents the alphabet size. We denote the length of the pattern p , $(|p|)$ by k .

We introduce the properties of a Suffix Tree; these are characteristics every Suffix Tree has to satisfy by definition.

Property 1: Root edges can only store indices of different characters which belong to the alphabet of the Suffix Tree string.

Property 2: Every internal node has at least one child node.

Property 3: Each node is labelled by index positions generated from combining edge labels from root to current node

Despite the fact that the programming community has adopted the RAM model for serial algorithms, no single model for parallel algorithms has gained wide acceptance. This is the situation since parallel PCs can either have shared memory model or distributed memory model therefore vendors have not agreed on a standard model. Shared memory model is an architecture in which the memory is shared and accessed simultaneously by the different processes whereas distributed memory model each processor to have its own private memory [102]. However, each new desktop and portable PC is currently a shared memory parallel PC and this pattern gives off an impression of being going towards the shared memory model. In spite of the fact that the reality of the situation will

become obvious eventually, in this study we likewise adopted such a strategy or course of shared memory model. We give an illustration of the shared memory model in Figure 5.1.

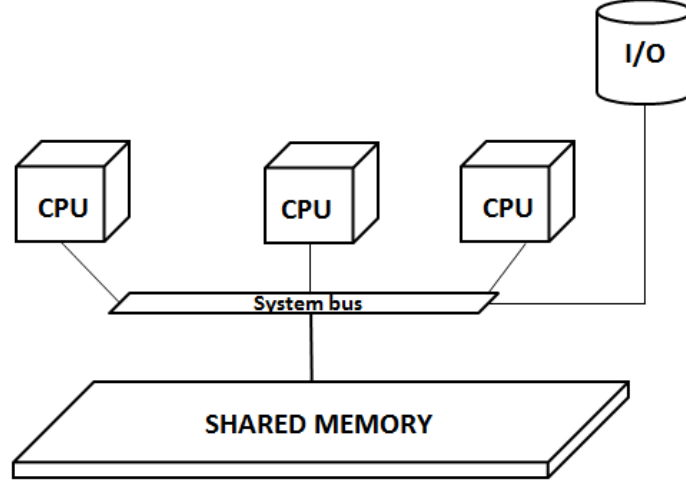


Figure 5.1 Shared Memory Model

5.2 PaST algorithm design

In this part we explain the design of our parallel algorithm (PaST) which is used for Suffix Tree construction. We do so with the assistance of an example. PaST adopts the KMER based generalized Suffix Tree introduced in Chapter 4. The construction is done branch by branch in parallel and every branch is a small sub-tree waiting to be merged to form a complete generalized Suffix Tree. We provide Example 5.1 to explain our design concept. In order to parallelize any algorithm, we first need to know which parts of a program require the most computation time. In our case it has been shown that construction of a Suffix Tree takes more time compared to the searching of patterns or repeats [80]. So we parallelise the construction part of our KSA algorithm.

Example 5.1:

Given a string $S = \{acgttacg \dots\}$ and $k = 4$, our algorithm starts by extracting the alphabet for S , which is $\Sigma = \{a, c, g, t, \dots, \lambda\}$, λ represents the last character in the alphabet. The next step involves generating sets in groups with same initial character c , where $c \in \Sigma$. Each set P_α , where P is the set containing all the suffixes starting with symbol α . Every

processor will process all sets. This processing happens independently in parallel for each set as shown in Figure 5.1

Initial Phase:

$$\begin{aligned}
 P_a &= \{acgt\} \wedge \{acg\} \notin P_a \\
 P_c &= \{cggt\} \wedge \{cg\} \notin P_c \\
 P_g &= \{gtta\} \wedge \{g\} \notin P_g \\
 P_t &= \{ttac, tacg\} \\
 P_n &= \{\lambda \dots, \lambda \dots\}
 \end{aligned}$$

Figure 5.2 Generating sets

The terminal symbol, \$, (as discussed in Chapter 2) is appended inorder to construct the explicit Suffix Tree. The following two rules are following to guarantee the explicit Suffix Tree:

1. No suffix of S is a prefix of a different suffix of S .
2. There is a leaf for each suffix of S .

Final Phase:

$$\begin{aligned}
 P_a &= \{acgt\$ \} \\
 P_c &= \{cggt\$ \} \\
 P_g &= \{gtta\$ \} \\
 P_t &= \{ttac\$, tacg\$ \} \\
 P_n &= \{\lambda \dots \$, \lambda \dots \$ \}
 \end{aligned}$$

Figure 5.3 Terminal Symbols are inserted in the final phase

Figure 5.3 shows the final independent sets generated, ready to be processed as sub trees in parallel. PaST proceeds by constructing the tree branch-wise independently for each set in parallel as in Figure 5.3. Construction of branches is done in parallel whereas

construction of each branch is done sequentially. This means the branches are constructed in the usual generalized Suffix Tree fashion, substring by substring following the sliding window technique. At the end, the completed branches are merged to a common root to form a complete generalized Suffix Tree for S as shown in Figure 5.5. This merging process is trivial and reduces the cost of expensive I/O thereby also contributing to less time requirements. Take note that, the merging procedure is also a source of problem since it increases the time requirements in previous studies however in this study the time requirements will be reduced by the fact that our merging process is not complex.

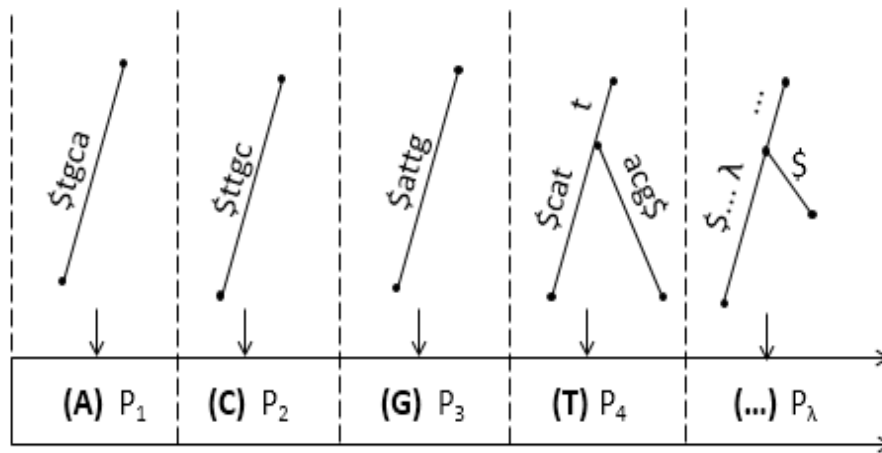


Figure 5.4 Branch-wise Parallel Construction of Suffix Tree of S

Our parallel novel idea is derived from the observation that all root edges of a Suffix Tree of sequence S , are derived from the alphabet of S , meaning each character $c \in \Sigma$ maps to a root edge in a one – to – one mapping. We prove and validate this in Lemma 1.

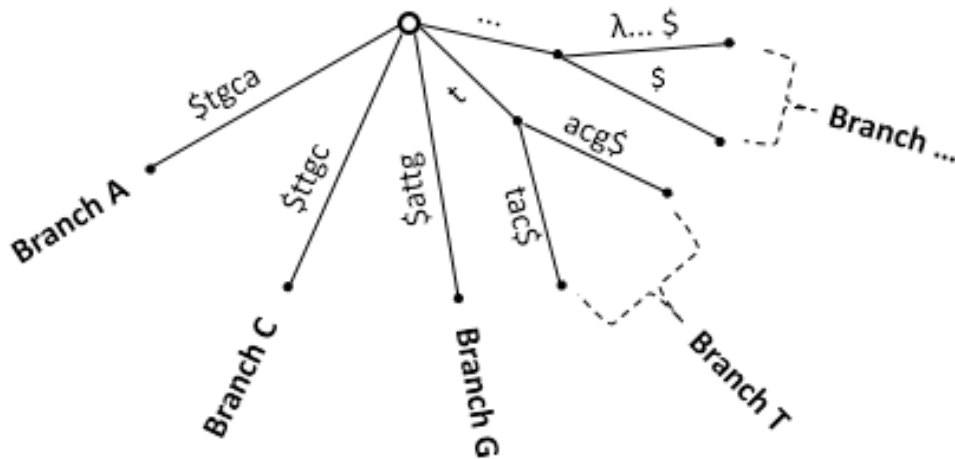


Figure 5.5 Merged Complete Suffix Tree (Example 5.1)

Lemma 1: If the size of alphabet of a sequence S , is $\sigma = |\Sigma|$, then the total number of root edges for the generalized Suffix Tree, T of S , is $\sigma + 1$, provided we have made use of all characters in Σ .

Proof: Let $S = \{abaabc\}$, then $\Sigma = \{a, b, c\}$ and $\sigma = 3$, given $k = 3$, that is size of pattern p , $|p|$. Thus we assume $e \neq 4$ where e is number of root edges of the generalized Suffix Tree T of S . So we then assume that, T has $e = 5$. We intend to prove that, $e = \sigma + 1$, given that $\sigma = 3$ (from above). We show this by constructing T for S in Figure 5.6. S has the following set of KMER substrings $\{aba, baa, aab, abc\}$ by Property 1 (Preliminaries section) which states that “Root edges can only store indices of different characters which belong to the alphabet of the Suffix Tree string”. We introduce our terminal symbol, $\$$, as standard practice for an explicit tree as earlier discussed, where $\$ < \Sigma$ and $\$ \notin \Sigma$. Finally we get $\{aba\$, baa\$, aab\$, abc\$, \$\}$ as our substrings to be added to T . We illustrate T in Figure 5.6.

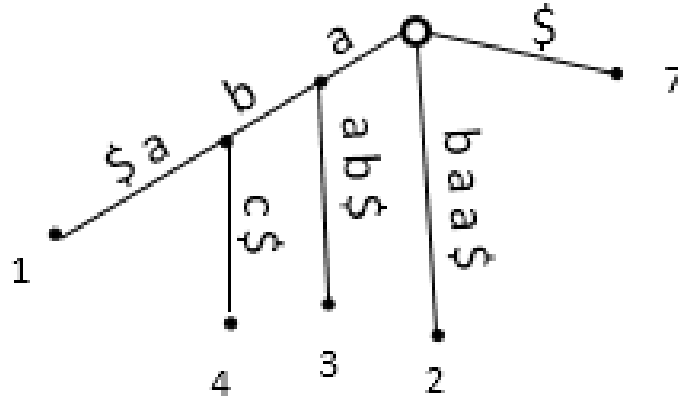


Figure 5.6 Suffix Tree of $\{abaabc\}$ with $|p|=3$

From Figure 5.6. it clearly shows that, $e \neq 5$, as assumed earlier but rather $e = 4$, which means $\sigma + 1 = 4$, therefore $\sigma = 3$ which is the numbers of characters of S .

In the next section the PaST algorithm is presented in Figure 5.7 in pseudo-code. A description of-parallel algorithm follows the algorithm.

Algorithm 3. Parallel algorithm for Suffix Tree construction (*PaST*)

Input : $S, n, k, \Sigma, |p|$,

Output: A generalized suffix tree representation ST of S

```

1  $A_{\Sigma=\{0,\dots,\sigma-1\}} := \{\}$ 
2  $\sigma := |\Sigma|$ 
3  $ST_b := ST_c := \{\}$  //initialise braches and complete suffix tree
4 parfor  $i \leftarrow 0$  to  $\sigma - 1$  do
5     for  $j \leftarrow 0$  to  $n - k$  do
6         if  $(A[i] = S[j])$  then
7             spawn  $ST_{b(A[j])} \leftarrow (S[j \dots j + (|p| - 1)])$ 
                        //construct tree per alphabet character
8         endif
9     endfor
10    sync
11 endparfor
12 for  $i \leftarrow 0$  to  $\sigma - 1$  do
13      $ST_c \in ST_{b(A[i])}$ 
14 endfor
15 return  $ST_c$ 

```

Figure 5.7 The PaST Algorithm

The description for our algorithm follows from the Example 5.1 given above. The construction of all branches of the generalized Suffix Tree is done in parallel. Each branch only needs its set of suffixes starting with the same initial character to carry out construction of a branch on its own. We made use of the observation from *Lemma 1* and *Property 1*. We begin by retrieving all the suffixes. This means the substrings, which start with same initial symbols are grouped to their respective sets only when $n \geq k$ (lines 4 – 6). This retrieval process is done in parallel and independently since each processor is searching for its matching alphabetic character in S since we are using the CREW model.

The next phase begins with independent construction of sub-trees by scanning through the set $ST_{b(A[j])}$ and picking each element for construction (lines 7 – 11). This construction follows a similar process for constructing the generalized Suffix Tree. The last phase involves merging the sub-trees to come up with a complete generalized Suffix Tree and this merging process is done sequentially (lines 12 – 13). We summarize how our algorithm fit in the DNA analysis process by making an illustration in Figure 5.8.

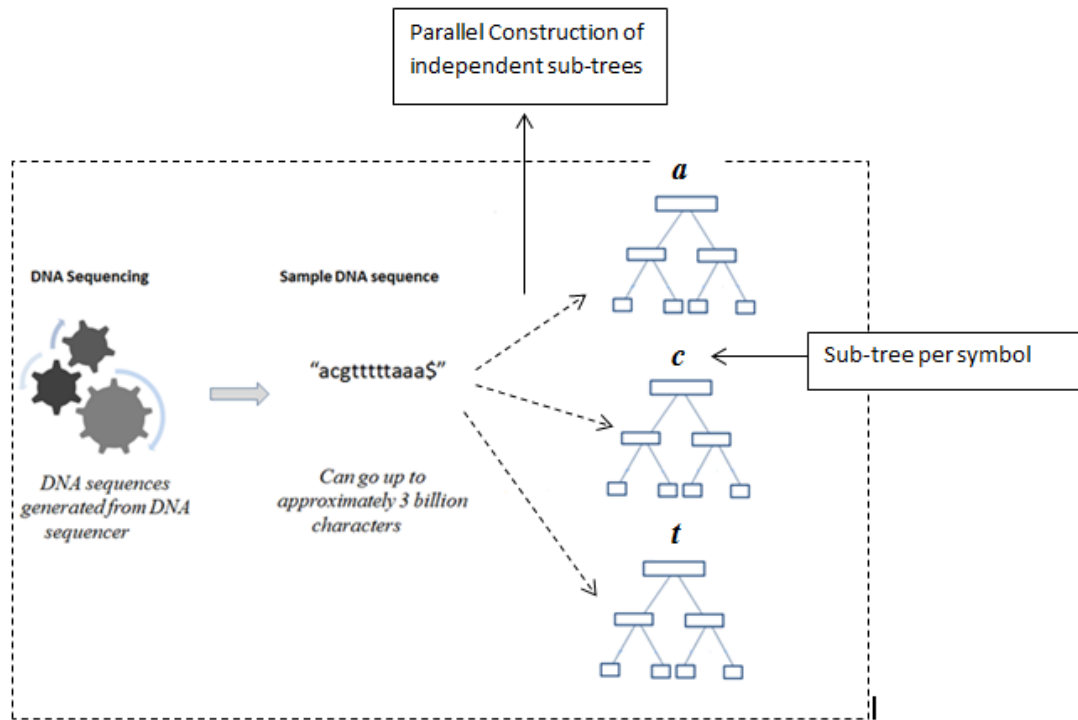


Figure 5.8 PaST Illustration

5.4 Evaluation Analysis

The software tools, test bed, platform, datasets, programming language used are similar to the one described in the previous section (Chapter 4 – Evaluation Strategy).

The PaST algorithm was implemented using Python 3.5.2 and Anaconda 4.1.1 for the experiments. Python is a popular and powerful language in data science and has a growing adoption in the scientific community. The reason for using python programming language is because it has a lot of parallel support libraries for the Suffix Tree data structure [97]. The Anaconda distribution was used because it comes with more preinstalled packages for handling large amount of data such as the DNA genome.

The implementation of PaST algorithm is a parallel implementation, that is, multiprocessing and multithreading python libraries were used to implement the parallelism part of the algorithm. The PaST algorithm makes use of the process-based parallelism. For instance, python synchronisation mechanisms such as locks, queues and other classes for spawning processes were used in the implementation [4].

5.5 Summary

In this chapter we have proposed and explored a novel approach to generalized Suffix Tree construction in parallel. A parallel algorithm called Parallel Construction of generalized Suffix Tree (PaST) was presented. PaST relies on the size of the patterns to be searched as input and construction of a truncated tree version of a fully-fledged generalized Suffix Tree in parallel. The core operation of this algorithm is the independent construction of sub-trees and the trivial merging of the sub-trees.

Chapter 6

Results and Discussion

6.0 Introduction

This chapter presents results and discussions of proposed algorithms. Mathematical evaluation of proposed algorithms is also discussed. The evaluation is twofold, that is, theoretical and experimental evaluation. Under theoretical evaluation we provide complexity analysis in Big O Notation and mathematical proofs for correctness of our algorithms. We present execution results of the implemented algorithms under experimental evaluation.

6.1 Theoretical Evaluation

The following sections provide theoretical evaluation of the algorithms which includes, complexity analysis using the Big O Notation and the related correctness proofs for the algorithms.

6.1.1 Computational complexity

In the following sub-sections, algorithms' complexities are presented, followed by formal proofs.

6.1.1.1 KSA Algorithm

Theorem 6.1. KSA takes $O(n)$ time and $O(n)$ space where n represents the size of the string S to be indexed and k represents pattern size.

Proof 6.1: Our algorithm executes $n - k$ times in the *for loop* (lines 6 – 10). Then executes $|sub| - 1$ times, which is k times, since the size of the substring is k . Then we have $((n - k) - k)$ times (k times for the *insert* function), since it is in the inner loop it will finally give us $O(nk)$ and hence $O(n)$ since k is a constant. If each symbol uses one unit of memory and each branch is k long that means, with $n - k$ substrings. Then the space complexity is $O(n)$ since k is a constant.

□

Proof 6.2: The time complexity is further proven as shown below: (Reference to Algorithm 2, that is, KSA Algorithm, Chapter 4 is made). Constants are assigned to each line in our pseudo-code for those which take constant time [38, 103].

Line 1: C_1

Line 2: C_2

Line 3: C_3

Line 4: C_4

Line 5: C_5

Line 6: $\sum_{i=0}^{n-k} C_6$

Line 7: $C_7 \cdot (\sum_{i=0}^{n-k} C_6)$

Line 8: $C_8 \cdot (\sum_{i=0}^{n-k} C_6)$

Line 9: $C_9 \cdot (\sum_{i=0}^{n-k} C_6)$

Line 10 & 11: C_{10}

Line 12: $\sum_{i=0}^{|sub|-1} C_{11}$

Line 13: $C_{12} \cdot (\sum_{i=0}^{|sub|-1} C_{11})$

Note that:

$$\sum_{i=m}^n c = n - m + c \quad (6.1)$$

From equation 6.1

$$\text{Line 6: } \sum_{i=0}^{n-k} C_6 = (n - k) + C_6 \quad (2)$$

$$\text{Line 12: } \sum_{i=0}^{|sub|-1} C_{11} = (|sub| - 1 + C_{11}) \quad (3)$$

We introduce a standard constant C for all constants:

Line 7, 8 & 9: $C_7 + C_8 + C_9 = C \cdot (n - k)$

Line 13: $C_{12} \cdot (|sub| - 1)$

Summing up all the costs together to get the time cost:

$$T(n) = (n - k) + C + (|sub| - 1) + C + C \cdot (n - k) + C \cdot (|sub| - 1)$$

Note. $|sub|$ is also a constant, $\therefore |sub| = C$

$$T(n) = 2n - 2k + C$$

Note. k and C are constants $\therefore 2k + C = C$

$$T(n) = 2n - C = n$$

$\therefore$ Time complexity = $O(n)$

□

6.1.1.2 PaST Algorithm

Theorem 6.2: The CREW PRAM PaST algorithm takes $O(n \log \sigma)$ time where n is the size of input sequence and σ is the alphabet size.

Proof 6.3: The SPAWN routine requires $\log \sigma - 1$ steps for line 4 (parfor) where $\sigma = |\Sigma|$ and is equivalent also to the number of processors to be used. Since SPAWN is in an inner loop which requires $n - k$ steps then the overall body (lines 4-10) requires $(n - k)$ times $(\log \sigma - 1)$ thus $(n - k)(\log \sigma - 1)$ plus $\sigma - 1$ steps for (lines 12 - 14). Therefore the overall complexity is $O(n \log \sigma)$.

□

6.1.2 Correctness Proofs

In the following sub-sections, we present formal proofs to prove the correctness of our KSA and PaST algorithms.

6.1.2.1 KSA Algorithm

The following is proven using the loop invariant for lines (6 - 10) of the KSA algorithm

Proof 6.3

Loop Invariant in this case: $i \leq n - k$

The invariant is checked and a proof for the correctness of the algorithm is provided.

Initialization: Before the first iteration $i = 0$, So $n = 1$ and $k = 1$ Then the S_{sub} will contain one element or substring since S will be $S[0,0]$, therefore only one substring will be inserted into the tree. Thus Invariant is satisfied.

Maintenance: This can be easily verified by checking the invariant after each iteration. In this case it is satisfied.

Termination: The proof of the correctness of our algorithm is given in the following paragraph.

When the loop terminates, then value of $i = (n + 1) - k$. In this case the loop invariant is satisfied again. This means all the substrings will be inserted into the tree. This is what we want our Algorithm to do. Thus our Algorithm is correct.

The loop invariant is proven for lines (12 – 14) of the KSA algorithm

Proof 6.4

Loop Invariant in this case: $i \leq k - 1$

The invariant is checked and a proof is provided to show that the algorithm is correct.

Initialization: Before the first iteration $i = 0$, So $k = 1$ Then the $|sub|$ will contain one character and this means only one character $node[sub[0]]$ will be added to the tree after the iteration. In this case it is satisfied.

Maintenance: This can be easily verified by checking the invariant after each iteration. In this case it is satisfied.

Termination: The proof of the correctness of our algorithm is given in the following paragraph.

When the loop terminates then value of $j = (k + 1) - k = k$, hence the loop invariant is satisfied. This means all characters will be added to the tree in a generalised Suffix Tree construction fashion. In this case the loop invariant is satisfied again. This means all the substrings will be inserted into the tree. This is what we want our Algorithm to do. Therefore, the algorithm is correct.

6.1.2.2 PaST Algorithm

The loop invariant is proven for lines (4 – 11) of the KSA algorithm

Proof 6.4

The Predicate or Loop Invariant in this case: $i \leq \sigma - k \wedge j \leq n - 1$

The invariant is checked and the algorithm proof of correctness follows. The checking process begins with the inner loop.

Initialization: Before the first iteration $j = 0$, So $n = 1$ and $k = 1$ Then the S_{sub} will contain only one element or one substring since S will be $S[0,0]$, therefore only one substring if size one will be inserted into the tree. For this follows the outer loop which is 0 on first iteration and σ is one, that is, the size of the alphabet since there is only one character in S so this runs in parallel as a single process to completion using the *parfor*. Thus Invariant is satisfied.

Maintenance: This can be easily verified by checking the invariant after each iteration. In this case both loops are satisfied.

Termination: The proof of the correctness of our algorithm is given in the following paragraph.

When the inner loop terminates then value of $j = (n + 1) - k$. In this case the loop invariant is satisfied again. Then the outer loop terminates and the value of $i = (\sigma + 1) - 1$ This means all the substrings will be inserted into the subtrees and finally merged. This is what we want our Algorithm to do. Therefore, the algorithm is correct.

6.2 Experimental Evaluation

In this sub-section results obtained from running the experiments under the configuration as described in the methodology sections of Chapter four and five are presented.

6.2.1 KSA Algorithm

Table 6.1 Space comparison between ST-based algorithm [2] and KSA algorithm

Dataset	Text Size (MB)	ST-Based (KB)	Our Algorithm (KB)
Rice 1	5	7,523,392	716,224
Rice 2	10	15,064,532	935,572
Rice 3	15	23,080,864	1,103,240
Rice 4	20	> 32,000,000	1,219,996
Rice 5	25	> 32,000,000	1,311,764
Prot. 1	10	15,397,672	706,852
Prot. 2	20	> 32,000,000	898,200
Prot. 3	30	> 32,000,000	1,058,024
Prot. 4	40	> 32,000,000	1,134,716
Prot. 5	50	> 32,000,000	1,180,444

The measurement used for the text size is megabytes. A text file of size 15MB contains approximately 15 million characters. Since this is ASCII code format, data are stored using one byte per symbol. The ones shown as '> 32,000,000' in Table 6.1 are estimates, this is because we have 32GB of memory available and our program did not terminate after several days possibly due to insufficient main memory. The construction of KSA algorithm is for $10 - mer$, that is, for $k = 10$.

Table 6.2 Space usage of for complete chromosomes and the whole genome

Dataset	Text Size (MB)	ST-Based (bytes)	Our Algorithm (KB)		
			$k = 5$	$k = 10$	$k = 15$
Rice Chr. 12	≈ 21.5	>32,000,000	43,780	1,257,628	23,281,280
X-Chr.	$\approx 151,855$	>32,000,000	159,488	1,429,320	>32,000,000
E.H.G.	$\approx 2,992,718$	>32,000,000	5,851,588	>32,000,000	>32,000,000

Rice Chr. 12 is Rice Chromosome 12, *X-Chr.* is the Human Chromosome X and *E.H.G.* is the Entire Human Genome. All columns show memory usage in kilobytes except 'Text Size' column which is megabytes. Peak memory usage is shown based on the value of k .

We made use of *memusg* tool to measure the peak memory usage. The tool is a standard tool to measure memory usage in UNIX environment. It is a ‘time’ like utility and is very accurate compared to similar tools [104]. In this study the UNIX system ‘time’ command utility was used to measure our programs’ execution time in seconds [105] and recorded as shown in Table 6.3.

Table 6.3 Time cost for complete chromosomes and the whole genome

Dataset	Text Size (MB)	ST-Based (seconds)	Our Algorithm (seconds)		
			$k = 5$	$k = 10$	$k = 15$
Rice Chr. 12	≈ 21.5	n/a	40.263	91.090	274.414
X-Chr.	$\approx 151,855$	n/a	279.484	584.902	n/a
E.H.G.	$\approx 2,992,718$	n/a	55,380.870	n/a	n/a

6.2.1.1 Observations and Discussions

A comparison on memory usage of the KSA algorithm was made with that of ST-based method [96]. Observations noted from this experimental study are discussed as follows:

1. The ST-based algorithm uses more memory than the KSA algorithm. The memory usage of the ST-based algorithm is ten times more than the KSA algorithm. This was noted on Protein and Rice data as shown in Table 6.1. This huge memory saving on the KSA algorithm made it possible to index the whole human genome using just less than 6GB of internal memory. The algorithm is therefore able to construct the generalized Suffix Tree of the entire human genome on a standard desktop PC with at least 6GB main memory, provided the value of k is such that $0 \leq k \leq 10$.
2. The KSA algorithm requires more main memory when $k \geq 16$. This is true on the X-Chromosome and on the complete human genome datasets. Another observation noted is that, the internal memory required for these datasets, is more than 32GB, that is, more than what was available on the workstation used in this experimental study.
3. Although execution time is not the main aim in this algorithm, it was

nonetheless noted that the construction time of all the datasets was less than 5 minutes except for the whole human genome which took close to 15 hours to complete both the construction and search for the 5 – mer pattern as shown in Table 6.3.

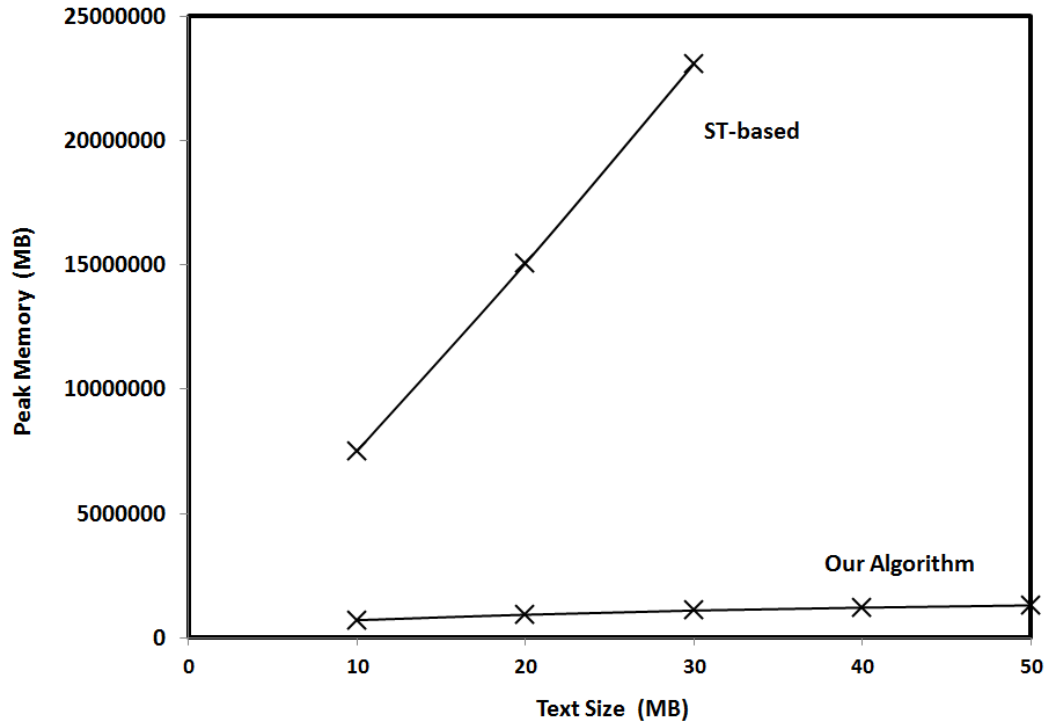


Figure 6.1 Space usage comparison on Rice sequences

Figure 6.1 displays memory usage of ST-Based method versus our KSA algorithm for $k = 5$. Memory is measured in megabytes (MB). The proposed algorithm consumes only 1.2GB of memory for 15MB of Rice dataset whilst ST-based algorithm requires peak memory of 24GB for the same dataset size (Table 6.1).

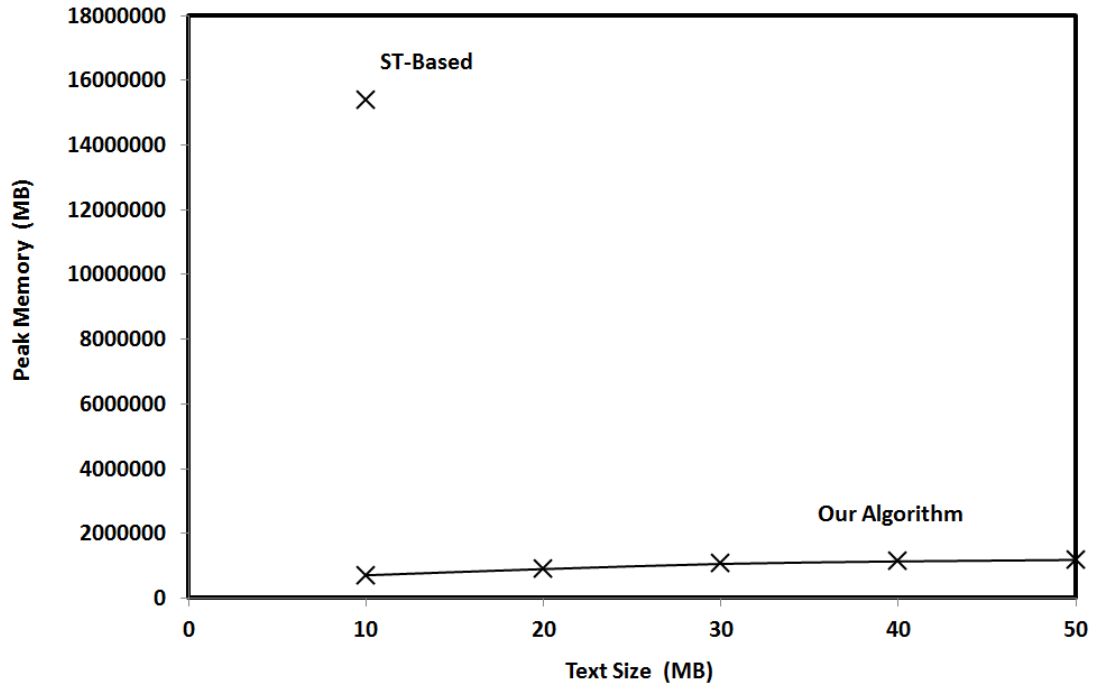


Figure 6.2 Memory usage on Protein sequences

Figure 6.2 shows peak memory usage of our algorithm versus ST-based method for $k = 10$. Memory is measured in MB. The ST-based algorithm did not terminate for a file size greater than 10MB. The assumption is that, possibly the algorithm required more internal memory than that which is available on the workstation, that is, 32GB. This observation is shown in Figure 6.2, for text size greater than 10MB. Figure 6.2 further shows that our algorithm continues to process the text files up to 50MB. Another further observation was noted that there was a huge difference of memory usage on 10MB text file on our algorithm versus ST-Based algorithm, approximately 15 times more.

Our results show improved performance to the generalized k -Truncated Suffix Tree (kTST) algorithm both in terms of memory and time requirements [18]. This is because the kTST algorithm supports up to $k = 11$, whereas ours support up to $k = 15$ and the input sizes used, are very small compared to ours which support relatively larger input sizes. For instance, kTST algorithm used 5000 nucleotides which is approximately 5KB and ours used at most 3GB, that is, the whole human genome. However, kTST also used varied sequences from 100 to 10,000. The kTST only used chromosomes 7 and 12 which are approximately 70MB each. Memory requirements were compared with the KSA algorithm for the same input size with $k = 6$ for kTST and $k = 10$ for our algorithm. K is used as $k = 10$ since in the experiments $k = 5$ and $k = 10$ is given keeping in mind

that $k = 10$ requires more memory than lower values of k . As shown in Table 6.2 the KSA X-chromosome used a peak memory of 1429.32MB whereas the kTST used 1817.17MB, about half gigabyte less. The kTST results are shown in Table 6.4.

Table 6.4 kTST algorithm results [18]

<i>Datastructure</i>	<i>Chromosome 7</i>			<i>Chromosome 12</i>		
	<i>Build</i>	<i>Search</i>	<i>Memory</i>	<i>Build</i>	<i>Search</i>	<i>Memory</i>
kTST	33.09	21.13	1817.17	27.56	17.12	1517.33
kTST (20-fold multi)	37.74	3.53	1272.40	32.11	2.95	1062.52
ST (4-byte)	–	–	–	–	–	–
ST (2-byte)	–	–	–	–	–	–
SA	–	–	–	–	–	–

Table 6.4 shows the experimental results for kTST algorithm, time is measured in seconds for build and search columns. Memory is measured in MB.

Table 6.5 TST algorithm results [17]

		File A	File B	File C	File D	File E
Truncated suffix tree	32	12713	95541	46130	90003	99088
	64	31429	100592	67447	90255	101682
	128	58562	103631	90174	90924	103816
	$k = 256$	88825	106560	107073	92004	104762
	512	126100	116284	108903	93122	105052
	1024	130763	124577	110542	93819	105052
Suffix tree		131073	129615	113915	94199	105052

A comparison is made to Truncated Suffix Tree (TST) algorithm which uses a slightly similar approach [17]. The experimental results for TST algorithm are shown in Table 6.5. File A is 728KB, File B is 444KB, File C is 460KB, File D is 448KB and File E is 404KB. The values of k used are larger possibly due to the fact that the size of the input files were very small. Therefore, no comparison of the input sizes is used. This is because the values of k used, have a huge variance. In terms of the input sizes the KSA algorithm seems to support huge file sizes which gives it more advantage, this is because most biological sequences are naturally huge in size.

6.2.2 PaST Algorithm

Execution time for the program is recorded for the construction only. The python's *time* function was used to measure Suffix Tree construction time in seconds as shown in Table 6.3 and Table 6.4.

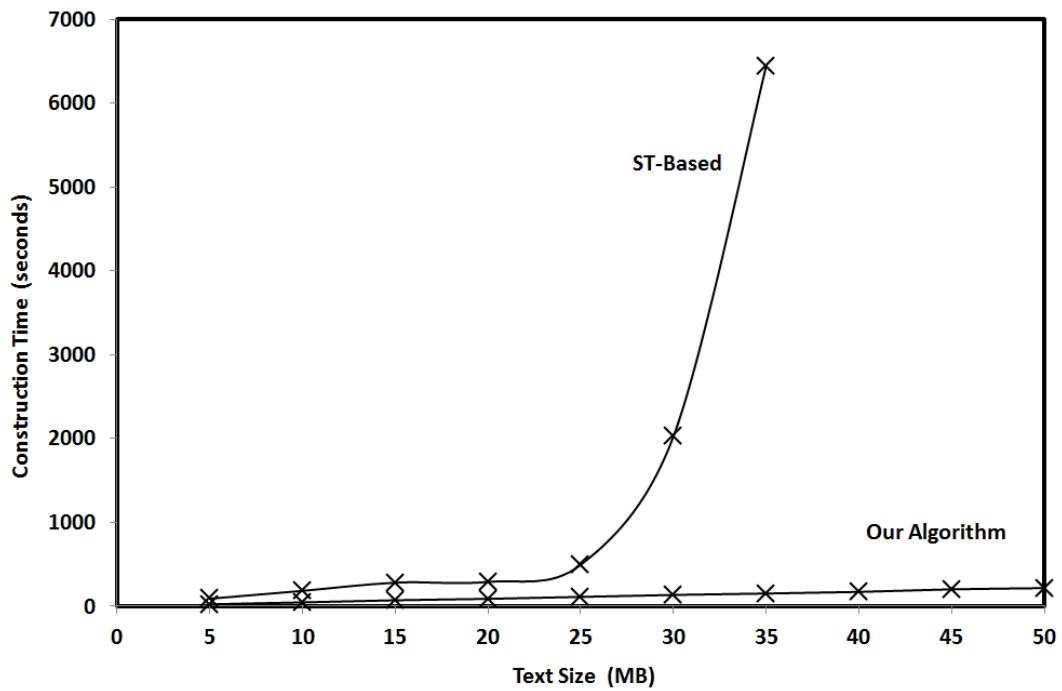


Figure 6.3 Construction time comparison on DNA sequences for pattern size of 5

Figure 6.3 shows execution times for construction of the generalized Suffix Tree for the PaST method and the ST-based method. The method was run with specification of $k = 5$ and the ST-based method was executed as is, since it does not employ the KMER technique which as in the KSA method.

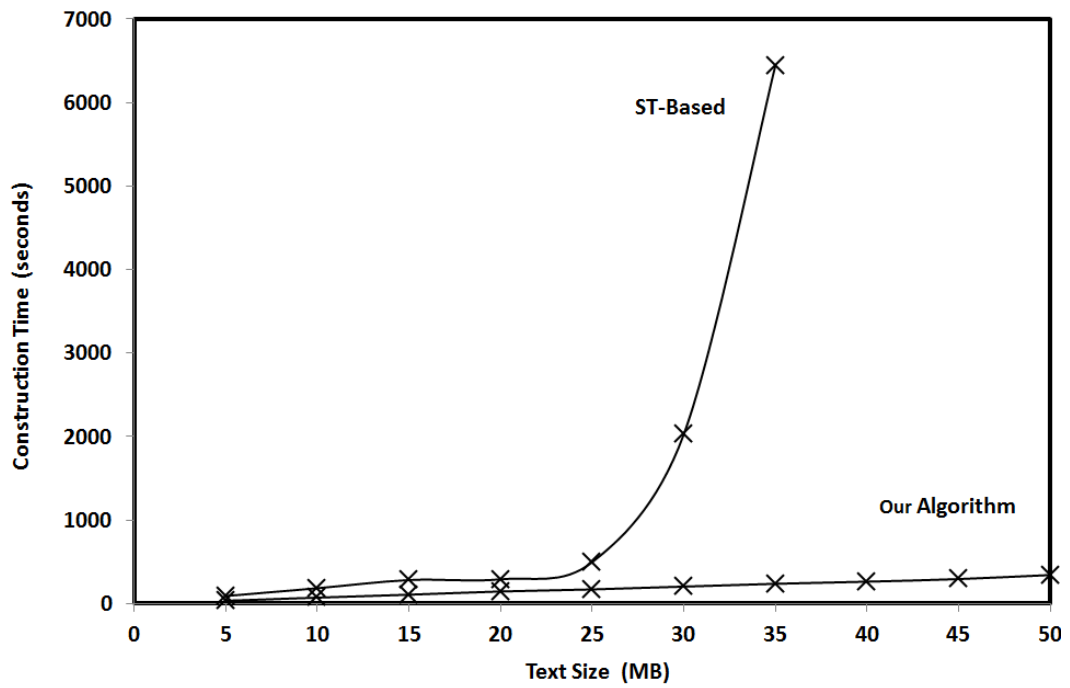


Figure 6.4 Construction time comparison on DNA sequences for pattern size of 10

Figure 6.4 shows construction times for generalized Suffix Tree for our method (KSA) versus ST-based method. The KSA method was executed with specification of 10 – *mer*, and the ST-based method was run with unlimited or unspecified *k* value since the algorithm does not need prior specification as with the KSA algorithm.

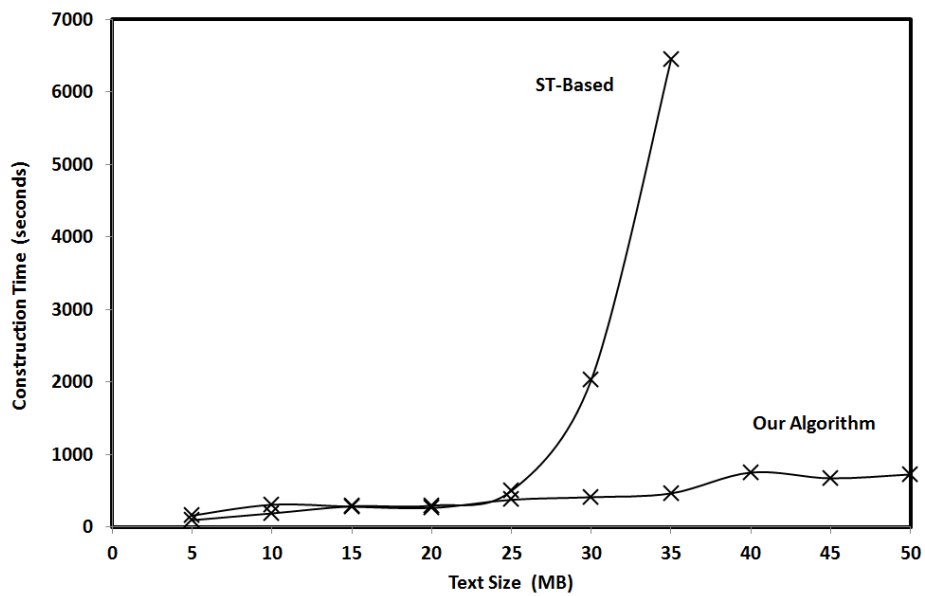


Figure 6.5 Construction time comparison on DNA sequences for pattern size of 15

Figure 6.5 shows execution times for construction of the generalized Suffix Tree for KSA algorithm versus the ST-based method. Our method was run with specification of 15 – *mer*, whereas the ST-based method was run as is since it does not require any specification as input prior to its execution.

The following Table 6.6 displays construction times in seconds to construct the generalised Suffix Tree based on values of $k = 5, 10$ and 15 . Construction times are shown for the two algorithms and the input size is also shown under the text size column starting from 5MB up to 50MB.

Table 6.6 Construction times of KSA vs. ST- Based algorithm

Text Size (MB)	ST-Based time (s)	Our algorithm (s)		
		$k = 5$	$k = 10$	$k = 15$
5	92	24	35	155
10	186	48	72	304
15	283	73	107	277
20	292	90	146	260
25	497	113	171	373
30	2028	136	204	409
35	6444	153	238	462
40	<i>n/a</i>	173	263	749
45	<i>n/a</i>	203	294	669
50	<i>n/a</i>	218	343	722

Table 6.7 depicts execution times for generalized Suffix Tree construction times for various datasets. The latter are full chromosomes and the entire human genome. The ST-Based algorithm did not run to completion for these datasets, even after several days, which we eventually terminated, and this behaviour is discussed in detail under observations section.

Table 6.7 PaST construction times for various genomic sequences

Dataset	Text Size (MB)	ST-Based (s)	Our algorithm (s)
			$k = 5$
X-Chromosome	≈ 150	<i>n/a</i>	678.180
Chromosome 1	≈ 226	<i>n/a</i>	1 054.110
Chromosome 2	≈ 235	<i>n/a</i>	1 088.860
Chromosome 3	≈ 193	<i>n/a</i>	862.330
Complete Human Genome	≈ 3000	<i>n/a</i>	12 944.460

With slight modification to the implementation, recordings of the search times were noted, which is the time it takes to search all the repeats of size k . The results are shown in Table 6.8 below.

Table 6.8 Search times for various repeat sizes

Dataset	Text Size (MB)	ST-Based	Our Algorithm(s)			
			$k = 10$	$k = 15$	$k = 20$	$k = 40$
Rice Chr. 12	≈ 21.5	<i>n/a</i>	299	391	413	722
X-Chr.	$\approx 151,855$	<i>n/a</i>	2 198	2 811	3 934	7 380
E.H.G.	$\approx 2,992,718$	<i>n/a</i>	28 244	39 321	<i>n/a</i>	<i>n/a</i>

The search times in the table does not include the time for construction. Usually this search operation is also done in DNA sequences to reveal some biological function. The *n/a* part on the results shows that the program was terminated after running for several days without completion.

6.2.2.1 Observations and Discussions

Observations and running times of our algorithm were made with that of ST-based technique [96]. The accompanying primary observations were noted from this test study:

1. PaST algorithm shows improved performance in terms of construction time. Our algorithm achieves a performance speedup of up to 15x over the sequential algorithm. It was observed that, the lower the value of k the more the performance speedup. This means, for instance, there is a improved performance speedup for $k = 5$ than for $k = 15$ as shown in Figure 6.6.
2. Our algorithm shows huge improvement when text size becomes larger than 25MB with reference to time cost as compared to the ST-based

algorithm. However the time cost for the ST-based method increases as text file becomes larger than 25MB. We have noted similar performance for both algorithms when file size is less 25MB. We explain this observation or phenomenon based on how parallelism works. Parallelism benefits or improvements are significant when there is small amount of computations due to context switches.

3. We further noted when file size becomes more than 35MB, the ST-based method does not complete which is possibly due to high memory requirements of the algorithm. With this file size ($> 35\text{MB}$) , our algorithm only requires to construct the generalised Suffix Tree of $k - \text{depth}$ which gives it a small memory foot print compared to constructing a complete tree of $n - \text{depth}$ (where n is size of string to be indexed) which would require high memory requirements.
4. The construction time of chromosomes and the entire human DNA was very impressive. The construction for the whole human DNA took approximately 3 and half hours to complete with $k = 5$ as shown in Table 6.6. Whereas for the ST-based method, took several days of which we had to terminate.
5. The last observation is on search times (Table 6.7). We realised that the search times were not impressive comparatively to other programs, possibly this is due to the fact that the algorithm was design with focus on reducing construction time and memory. Small repeats took reasonable times however larger repeats such size 40 took for instance approximately 2 hours to complete the searching in X-chromosome for all size 40 repeats.

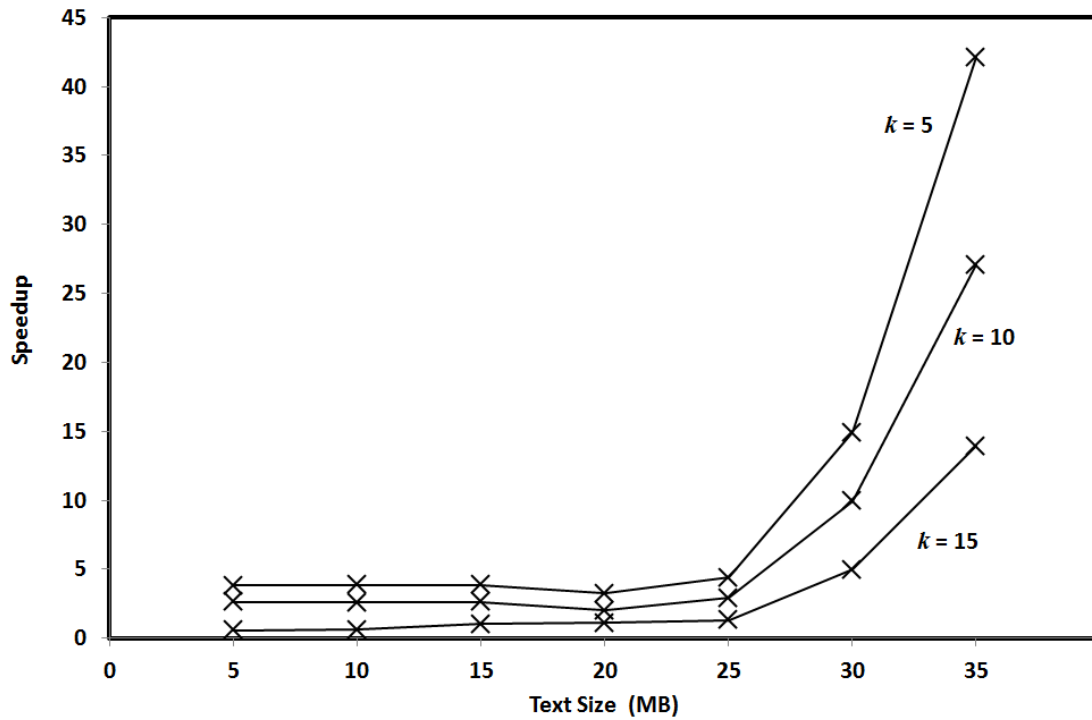


Figure 6.6 Speedup relative to text size

Figure 6.6 shows performance speed up for different values of k for varying input sizes. It can be noted that there is more performance gains for small values of k .

6.3 Summary

In this chapter we presented theoretical and experimental results for the proposed two algorithms followed by analysis and discussion of the results. The state of the art Ukkonen's algorithm was used as a benchmark to test our implementation of the KSA algorithm. The KSA algorithm was noted improved performance in terms of memory requirements enabling to index the whole human genome with only at least 6GB of main memory and the PaST algorithm shows improved performance in terms of time requirements with a performance speedup of up to 15x. All the proposed algorithms require specification of pattern size prior to construction of the tree.

Chapter 7

Conclusions and Recommendations

7.0 Introduction

This chapter summarises the study and makes suggestions for future research. It discusses the research which was undertaken and the challenges faced during the study. The main contributions of this research are also presented. The algorithm designs were presented in Chapter four and five. Chapter six presented the results and discussions of those results.

7.1 Challenges and Limitations

The main challenge faced during this research was the unavailability of software tools to benchmark with, in terms of their running times. This was faced when addressing Objective 1 and also on last objective. Instead we went ahead with the authors' results as recorded in their papers. It could have brought more understanding to run the other authors' software tools in a similar environment as in this study so as to make comparison detailed. However, the theoretical complexity analysis still gave us a comparative indication on the performance of the algorithms. Another challenge was low internal memory on our workstation, 32GB is a good memory but we would have wanted more for observation purposes of the behaviour of our algorithm for pattern sizes of greater than 16-mer on the complete human genome.

7.2 Conclusions and Future research

There is an exponential growth in the amount of biological sequences. In order to analyse such huge amounts of data, new space and time-efficient algorithmic techniques were necessary. Suffix Trees have been successfully applied extensively in string-related applications and their adoption in bioinformatics have increased [74]. A lot of attempts have been made to improve the Suffix Tree construction algorithms in biological sequence data, however most of these still do not scale well with these huge datasets. Hence, the challenge still remains. In this work we developed two algorithms, the first one deals with space requirements (KSA algorithm), and the second extends the KSA algorithm to deal with time requirements (PaST algorithm). The PaST algorithm is a parallel algorithm which is comparatively space and time-efficient for construction of generalised Suffix Trees. In other words, we came up with a data structure which is a variant of the generalised Suffix Tree (KSA), and applied parallelism to its construction

and searching of repeats (PaST). Our approach takes advantage of the knowledge of repeat size to be searched prior to construction and is an in-memory based approach, that is, it does not use external memory during its construction or execution.

PaST was found to be useful in finding small sized repeats similar to microsatellites since they stretch from size 2 to 6 long. Microsatellites are important since they can be used as DNA markers for different biological functions [77]. These types of repeats allow you to specify size before detecting them prior to the analysis.

In Chapter 3 Objective 1 was achieved by carrying out a survey with current DNA based data structures, techniques and algorithms and providing strategic research directions for better results.

In Chapter 4, The aim was to reduce the space consumption of the generalised Suffix Tree construction; In essence, we aimed to answer the following research question:

To what extent can the memory requirements be improved by truncating the Suffix Tree using the KMER technique?

In the same manner we also managed to achieve Objective 2:

Design a space-efficient algorithm for the generalized Suffix Tree construction using the sliding window technique

The main contribution in Chapter 2 is a serial algorithm for constructing a generalized Suffix Tree in a space-efficient manner based on the KMER specification. This study managed to reduce the memory usage comparatively to those algorithms of similar approach such as kTST and TST algorithms and achieved a linear time and space complexity.

In Chapter 5, the focus was on reducing time requirements during construction of the generalized Suffix Tree and search times. This addressed the following two research questions and objective respectively:

How can Suffix Tree construction algorithm(s) in DNA sequences be improved in a space and time-efficient manner?

How can the ubiquitous multicore architecture be best utilised to improve the running time of repeat finding algorithm(s)?

And the the third objective:

Design a parallel algorithm for KMER-based generalized Suffix Tree construction.

This study provided answers to the second research question by making use of *lemma 1* to help identify areas of parallelisation of the generalised Suffix Tree. The first research question took advantage of the KSA algorithm which was already space-efficient and extended to improve the time requirements by parallelisation. The main contribution of this chapter is a novel parallel algorithm which is space and time-efficient with a performance speedup of up to 15x over the state of the art sequential algorithm, Ukkonen algorithm [72]. 15x means that the performance gained is 15 times more than the best sequential algorithm.

As part of future research there is need to increase the size of k , that is, the size of the repeats to be searched whilst introducing another memory saving technique to allow larger values of k . There is also need to make the algorithm (KSA) flexible in terms of the alphabet size as compared to our fixed alphabet size approach. Furthermore, work still needs to be done to allow the algorithm to make use of all available processors on the computer rather than using the number of processors equivalent to the size of the alphabet. This can greatly increase performance of our algorithm. The size of the memory for the workstation computer used for experiments can be increased to support large KMER size. Another possible research is to create a user interface for PaST algorithm so as to make our algorithm more practical and widely used in biological applications.

The parallel algorithm can be further modified to explore application of Hadoop and MapReduce framework. Pointers were noted for not adopting MapReduce approach however this can still be explored for positive results.

Lastly, the PaST algorithm was created with the focus of reducing the construction time and memory, it is therefore useful to modify the algorithm to include search times that parallelisation of searching for repeats as seen from the brief results we recorded.

References

- [1] T. White, *Hadoop: The definitive guide*: " O'Reilly Media, Inc.", 2012.
- [2] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, pp. 52-59, 1997.
- [3] C. E. Leiserson and I. B. Mirman, "How to survive the multicore software revolution (or at least survive the hype)," *Cilk Arts*, vol. 1, 2008.
- [4] J. Palach, *Parallel Programming with Python*: Packt Publishing Ltd, 2014.
- [5] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, *et al.*, "The real cost of sequencing: scaling computation to keep pace with data generation," *Genome biology*, vol. 17, p. 53, 2016.
- [6] W. J. Ansorge, "Next-generation DNA sequencing techniques," *New biotechnology*, vol. 25, pp. 195-203, 2009.
- [7] V. Becher, A. Deymonnaz, and P. Heiber, "Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome," *Bioinformatics*, vol. 25, pp. 1746-1753, 2009.
- [8] E. C. Hayden, "The \$1,000 genome," *Nature*, vol. 507, p. 294, 2014.
- [9] N. Siva, "UK gears up to decode 100 000 genomes from NHS patients," *The Lancet*, vol. 385, pp. 103-104, 2015.
- [10] K. Wetterstrand. (2016, 15 February). *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. Available: www.genome.gov/sequencingcostsdata
- [11] R. Pokrzywa and A. Polanski, "BWtrs: a tool for searching for tandem repeats in DNA sequences based on the Burrows–Wheeler transform," *Genomics*, vol. 96, pp. 316-321, 2010.
- [12] A. M. Spitsina, Y. L. v. Orlov, N. y. N. Podkolodnaya, A. V. Svichkarev, A. I. Dergilev, M. Chen, *et al.*, "Supercomputer analysis of genomics and transcriptomics data revealed by high-throughput DNA sequencing," *Program systems: theory and applications*, vol. 6, pp. 157-174, 2015.
- [13] X. Liao, S. Peng, Y. Lu, Y. Cui, C. Wu, H. Wang, *et al.*, "Neo-heterogeneous Programming and Parallelized Optimization of a Human Genome Re-sequencing Analysis Software Pipeline on TH-2 Supercomputer," *Supercomputing Frontiers and Innovations*, vol. 2, pp. 73-83, 2015.
- [14] J. M. Martorell, "Barcelona Supercomputing Center: Science accelerator and producer of innovation," *Contributions to science*, pp. 5-11, 2017.
- [15] S. Memeti, S. Pllana, and J. Kołodziej, "Optimal worksharing of DNA sequence analysis on accelerated platforms," in *Resource Management for Big Data Platforms*, ed: Springer, 2016, pp. 279-309.
- [16] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit, "Engineering a compressed suffix tree implementation," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 2, 2009.
- [17] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park, "Truncated suffix trees and their application to data compression," *Theoretical Computer Science*, vol. 304, pp. 87-101, 2003.
- [18] M. H. Schulz, S. Bauer, and P. N. Robinson, "The generalised k-Truncated Suffix Tree for time-and space-efficient searches in multiple DNA or protein sequences," *International journal of bioinformatics research and applications*, vol. 4, pp. 81-95, 2008.

- [19] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger, "Computing the longest common prefix array based on the Burrows–Wheeler transform," *Journal of Discrete Algorithms*, vol. 18, pp. 22-31, 2013.
- [20] S. Wandelt, M. Bux, and U. Leser, "Trends in genome compression," *Current Bioinformatics*, vol. 9, pp. 315-326, 2014.
- [21] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*: Cambridge university press, 1997.
- [22] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483-485.
- [23] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, pp. 1369-1386, 2012.
- [24] U. C. Satish, "Parallel Suffix Tree Construction for Genome Sequence Using Hadoop," 2013.
- [25] N. Siva, "1000 Genomes project," ed: Nature Publishing Group, 2008.
- [26] K. Armstrong, "Can genomics bend the cost curve?," *Jama*, vol. 307, pp. 1031-1032, 2012.
- [27] G. Benson, "Tandem repeats finder: a program to analyze DNA sequences," *Nucleic acids research*, vol. 27, p. 573, 1999.
- [28] F. Fernandes and A. T. Freitas, "slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array," *Bioinformatics*, vol. 30, pp. 464-471, 2014.
- [29] G. Manzini, "An analysis of the Burrows—Wheeler transform," *Journal of the ACM (JACM)*, vol. 48, pp. 407-430, 2001.
- [30] G. Achaz, F. Boyer, E. P. Rocha, A. Viari, and E. Coissac, "Repseek, a tool to retrieve approximate repeats from large DNA sequences," *Bioinformatics*, vol. 23, pp. 119-121, 2007.
- [31] J. O. S. Fogh, "Pattern Matching Using Suffix Trays, Arrays and Trees," Aarhus Universitet, Datalogisk Institut, 2014.
- [32] R. Cole, T. Kopelowitz, and M. Lewenstein, "Suffix trays and suffix trists: Structures for faster text indexing," *Algorithmica*, vol. 72, pp. 450-466, 2015.
- [33] S. S. Skiena, *The algorithm design manual: Text* vol. 1: Springer Science & Business Media, 1998.
- [34] P. Bieganski, "Genetic sequence data retrieval and manipulation based on generalized suffix trees," 1995.
- [35] K. Monostori, A. Zaslavsky, and H. Schmidt, "Identifying overlapping documents in semi-structured text collections," in *Australasian Computer Science Conference*, 2000.
- [36] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing*, vol. 26, pp. 1343-1362, 1997.
- [37] J. H. Huang and D. Powers, "Suffix tree based approach for chinese information retrieval," in *Intelligent Systems Design and Applications, 2008. ISDA'08. Eighth International Conference on*, 2008, pp. 393-397.
- [38] T. H. Cormen, *Introduction to algorithms*: MIT press, 2009.
- [39] M. Maaß, "Suffix trees and their applications," *Ausarbeitung von der Ferienakademie*, vol. 99, 1999.
- [40] M. Sipser, *Introduction to the Theory of Computation* vol. 2: Thomson Course Technology Boston, 2006.
- [41] P. Seongbin and Y. Sojung, "Teaching some informatics concepts using formal system," *Informatics in Education-An International Journal*, pp. 323-332, 2014.

- [42] T. Cebrián and J. Edmonds, "How to think about algorithms," ed: Cambridge University Press, 2011.
- [43] E. Gafni, J. Naor, and P. Ragde, "On separating the EREW and CREW PRAM models," *Theoretical Computer Science*, vol. 68, pp. 343-346, 1989.
- [44] Z. Vetulani and J. Mariani, "Human Language Technology. Challenges for Computer Science and Linguistics," *Lecture Notes in Computer Science*, vol. 6562, p. 515, 2011.
- [45] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 26, pp. 53-56, 1983.
- [46] R. Paige, "Programming with invariants," *IEEE Software*, vol. 3, p. 56, 1986.
- [47] J. Bell, "Bibliography for the history of induction in mathematics," *Bernoulli*, vol. 10, p. 29, 2015.
- [48] J. J. Rotman, *Journey into mathematics: An introduction to proofs*: Courier Corporation, 2013.
- [49] M. Otte and N. Correll, "C-Forest: Parallel shortest path planning with superlinear speedup," *IEEE Transactions on Robotics*, vol. 29, pp. 798-806, 2013.
- [50] L. Cummings and C. Patel, "Chapter 3 Research methodology," in *Managerial Attitudes toward a Stakeholder Prominence within a Southeast Asia Context*, ed: Emerald Group Publishing Limited, 2009, pp. 53-87.
- [51] S. Kurtz and C. Schleiermacher, "REPuter: fast computation of maximal repeats in complete genomes," *Bioinformatics*, vol. 15, pp. 426-427, 1999.
- [52] O. Delgrange and E. Rivals, "STAR: an algorithm to search for tandem approximate repeats," *Bioinformatics*, vol. 20, pp. 2812-2820, 2004.
- [53] K. U. Schulz and S. Mihov, "Fast string correction with Levenshtein automata," *International Journal on Document Analysis and Recognition*, vol. 5, pp. 67-85, 2002.
- [54] K. Oflazer, "Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction," *Computational Linguistics*, vol. 22, pp. 73-89, 1996.
- [55] R. De La Briandais, "File searching using variable length keys," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959, pp. 295-298.
- [56] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM (JACM)*, vol. 23, pp. 262-272, 1976.
- [57] P. Weiner, "Linear pattern matching algorithms," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, 1973, pp. 1-11.
- [58] U. C. Satish, P. Kondikoppa, S.-J. Park, M. Patil, and R. Shah, "MapReduce based parallel suffix tree construction for human genome," in *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, 2014, pp. 664-670.
- [59] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, pp. 935-948, 1993.
- [60] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo, "Position heaps: A simple and dynamic text indexing data structure," *Journal of Discrete Algorithms*, vol. 9, pp. 100-121, 2011.
- [61] M. Kempf, R. Bayer, and U. Guntzer, "Time optimal left to right construction of position trees," *Acta Informatica*, vol. 24, pp. 461-474, 1987.
- [62] M. E. Majster and A. Reiser, "Efficient on-line construction and correction of position trees," *SIAM Journal on Computing*, vol. 9, pp. 785-807, 1980.
- [63] P. Deoghare, "Space efficient string search algorithms and data structures," Colorado State University. Libraries, 2016.
- [64] R. Grossi, "A quick tour on suffix arrays and compressed suffix arrays," *Theoretical Computer Science*, vol. 412, pp. 2964-2973, 2011.

- [65] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, pp. 33-50, 2004.
- [66] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Linear time Lempel-Ziv factorization: Simple, fast, small," in *Annual Symposium on Combinatorial Pattern Matching*, 2013, pp. 189-200.
- [67] M. Crochemore, L. Ilie, and W. F. Smyth, "A simple algorithm for computing the Lempel Ziv factorization," in *Data Compression Conference*, 2008, pp. 482-488.
- [68] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "An alphabet-friendly FM-index," in *String Processing and Information Retrieval*, 2004, pp. 150-160.
- [69] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098-1101, 1952.
- [70] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, pp. 26-37, 2009.
- [71] R. R. Loka, "Serial computing is not dead," *Computer*, vol. 43, pp. 8-8, 2010.
- [72] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249-260, 1995.
- [73] R. Kolpakov, G. Bana, and G. Kucherov, "mreps: efficient and flexible detection of tandem repeats in DNA," *Nucleic acids research*, vol. 31, pp. 3672-3678, 2003.
- [74] J. Stoye and D. Gusfield, "Simple and flexible detection of contiguous repeats using a suffix tree," *Theoretical Computer Science*, vol. 270, pp. 843-856, 2002.
- [75] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "The enhanced suffix array and its applications to genome analysis," in *Algorithms in Bioinformatics*, ed: Springer, 2002, pp. 449-463.
- [76] M. Federico and N. Pisanti, "Suffix tree characterization of maximal motifs in biological sequences," in *Bioinformatics Research and Development*, ed: Springer, 2008, pp. 456-465.
- [77] C. Abajian, "Sputnik: DNA microsatellite repeat search utility," *Program available at: <http://epressoftware.com/pages/sputnik.jsp>*, 1994.
- [78] J. Kärkkäinen, "Fast BWT in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, pp. 249-257, 2007.
- [79] J. Fischer, V. Makinen, and N. Valimaki, "Space efficient string mining under frequency constraints," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, 2008, pp. 193-202.
- [80] M. O. Kulekci, J. S. Vitter, and B. Xu, "Efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet tree," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, pp. 421-429, 2012.
- [81] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel, "Practical methods for constructing suffix trees," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 14, pp. 281-299, 2005.
- [82] J. Allali and M.-F. Sagot, "The at-most k -deep factor tree," 2004.
- [83] R. K. Menon, G. P. Bhat, and M. C. Schatz, "Rapid parallel genome indexing with MapReduce," in *Proceedings of the second international workshop on MapReduce and its applications*, 2011, pp. 51-58.
- [84] R. Hariharan, "Optimal parallel suffix tree construction," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, 1994, pp. 290-299.
- [85] G. M. Landau, B. Schieber, and U. Vishkin, "Parallel construction of a suffix tree," in *International Colloquium on Automata, Languages, and Programming*, 1987, pp. 314-325.

- [86] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, "Parallel construction of a suffix tree with applications," *Algorithmica*, vol. 3, pp. 347-365, 1988.
- [87] C. Chen and B. Schmidt, "Constructing large suffix trees on a computational grid," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 1512-1523, 2006.
- [88] A. M. Carvalho, A. L. Oliveira, A. T. Freitas, and M.-F. Sagot, "A parallel algorithm for the extraction of structured motifs," in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 147-153.
- [89] D. Tsirogiannis and N. Koudas, "Suffix tree construction algorithms on modern hardware," in *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 263-274.
- [90] A. Ghoting and K. Makarychev, "Serial and parallel methods for i/o efficient suffix tree construction," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 827-840.
- [91] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis, "ERA: efficient serial and parallel suffix tree construction for very long strings," *Proceedings of the VLDB Endowment*, vol. 5, pp. 49-60, 2011.
- [92] M. Comin and M. Farreras, "Efficient parallel construction of suffix trees for genomes larger than main memory," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 211-216.
- [93] C. R. Kothari, *Research methodology: Methods and techniques*: New Age International, 2004.
- [94] T. Greenfield, *Research methods for postgraduates*: John Wiley & Sons, 2016.
- [95] W.-c. Feng and P. Balaji, "Tools and environments for multicore and many-core architectures," *Computer*, vol. 42, pp. 26-27, 2009.
- [96] M. Nelson, "Fast string searching with suffix trees," *Dr. Dobbs's Journal*, pp. 115-119, 1996.
- [97] H. P. Langtangen, *A primer on scientific programming with Python* vol. 2: Springer, 2009.
- [98] R. C. NCBI, "Database resources of the National Center for Biotechnology Information," *Nucleic acids research*, vol. 41, p. D8, 2013.
- [99] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, and D. Seco, "Efficient wavelet tree construction and querying for multicore architectures," in *Experimental Algorithms*, ed: Springer, 2014, pp. 150-161.
- [100] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE transactions on knowledge and data engineering*, vol. 26, pp. 97-107, 2014.
- [101] J. Jada, *An introduction to parallel algorithms*: addison Wesley, 1992.
- [102] D. E. Lenoski and W.-D. Weber, *Scalable shared-memory multiprocessing*: Elsevier, 2014.
- [103] M. T. Goodrich, R. Tamassia, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Data Structures & Algorithms in Java," ed: John Wiley and Sons, Inc, 2007.
- [104] J. Clark, "memusg" *Github*, 2012.
- [105] A. Robbins, *Unix in a Nutshell: A Desktop Quick Reference-Covers GNU/Linux, Mac OS X, and Solaris*: " O'Reilly Media, Inc.", 2005.

Appendix A

KSA algorithm

We show only the code listing for the core part of our KSA Algorithm

```
class Generalised_Suffix_Tree:

    def __init__(self):
        self.root = collections.defaultdict()

    def add(self, substring): # adds a substring in to the tree
        current = self.root
        for character in substring:
            current = current.setdefault(character, {})
        current.setdefault("$") # adds a terminal symbol to
each substring in to the tree
        # @parameter {string}, prefix
        # @return type,{boolean}
        # @Returns if there is word in trie structure
        # @that starts with the given prefix.
    def _prefix_(self, prefix): # not necessary but can be
used to find or search a prefix
        current = self.root
        for character in prefix:
            if character not in current:
                return False
            current = current[character]
        return True

    def search(self, pattern): # takes pattern as a string
parameter
        current = self.root
        for character in pattern:
            if character not in current:
                return False # returns boolean false if not
found
            current = current[character]
        if "$" in current:
            return True # returns boolean true if not found
        return False
```

```

# Now test the class
#os.system('stat -c%s dna.fa > size.txt')
#with open('size.txt','r') as file1:
#    file_size=int(file1.readline())

if __name__=="__main__":
    p = subprocess.Popen("stat -c%s dna.fa",
        stdout=subprocess.PIPE, shell=True) #@dna.fa - you specify the
        textfile to be read or indexed to calculate the size of the file
        (file_size, err) = p.communicate()

    file_size=int(file_size)-1

    pattern='TAAAT' #specify the repeat or pattern to
    be searched here - the repeat also detects the size of tree and
    K value so the tree will be of K height

    k=len(pattern)

    gst = Generalised_Suffix_Tree()

    data=[]

    n = (int(file_size)-k)+1
    with open('dna.fa','r') as myfile: #@dna.fa - you specify
    the textfile to be read or indexed
        data=myfile.read()
    #print ('n is',n)
    #print data1
    for i in range(n):

        #myfile.seek(i)
        #sub=myfile.read()# Go to the 3rd byte before the
end
        gst.add(data[i:i+k]) # @data[i:i+k] the sliding
window size which is being moved one character one by one by the
for-loop
        #print data1[i:i+repeat_size]

```

Appendix B

PaST algorithm

We show only the code listing for the core part of our PaST Algorithm

```
lock=multiprocessing.Lock()

class Generalised_Suffix_Tree:

    def __init__(self):
        self.root = collections.defaultdict()

    def add(self, substring): # adds a substring in to the
tree
        current = self.root
        for character in substring:
            current = current.setdefault(character, {})
        current.setdefault("$") # adds a terminal symbol to
each substring in to the tree

    # @parameter, {string} prefix
    # @return type, {boolean}
    # @Returns if there is word in trie structure
    # @that starts with any given prefix.
    def _prefix_(self, prefix): # not necessary but can be
used to find or search a prefix
        current = self.root
        for character in prefix:
            if character not in current:
                return False
            current = current[character]
        return True

    def search(self, pattern): # takes pattern as a string
parameter
        current = self.root
        for character in pattern:
            if character not in current:
                return False # returns boolean false if not
found
            current = current[character]
        if "$" in current:
            return True # returns boolean true if not found
        return False

def sub_tree(var):
    #global data
```

```

global gst
global k
gst = Generalised_Suffix_Tree()
for i,item in enumerate(data):
    if item==var:
        lock.acquire()
        gst.add(data[i:i+k])
        lock.release()
print (gst._prefix_(pattern))

if __name__=="__main__":
    global data
    data=[]
    p = subprocess.Popen("stat -c%s chr3.fa",
stdout=subprocess.PIPE, shell=True)

#@dna.fa - you specify the textfile to be read or indexed to
calculate the size of the file

    (file_size, err) = p.communicate()

    file_size=int(file_size)-1

    pattern='CCCAA'          #specify the repeat or pattern to
be searched here - the repeat also detects the size of the tree
and the value K so the tree will be of K height

    repeat1='CCTAACCCTAACCCTAACCCTAACCCCTAACCCCTAA'

# not necessary, its for testing purposes

    k=len(pattern)

    n = (int(file_size)-k)+1

    #n = int(file_size)

    with open('chr3.fa','r') as myfile:

        #@dna.fa - you specify the textfile to be read or indexed

            data=myfile.read().replace('\n','')
        #sub_tree_T()

        start = default_timer()

        Alphabet=['A','C','G','T']

```

```
p=multiprocessing.Pool(4)

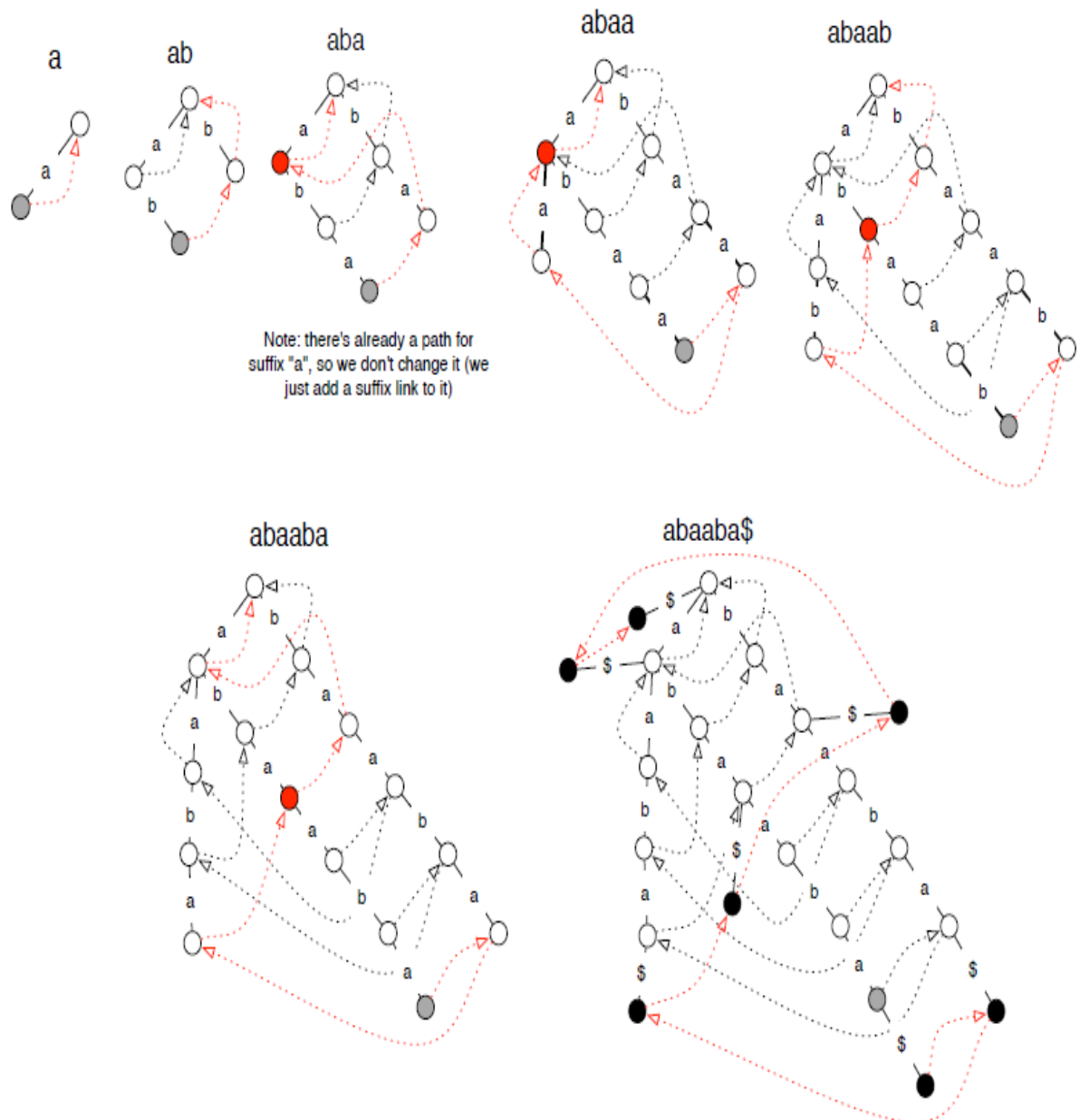
p.map(sub_tree,Alphabet)

p.close()

p.join()
```

Appendix C

Step by step construction using Ukkonen Algorithm



Appendix D

Sample Screenshot – KSA Algorithm

```
fkaniwa@fkaniwa1: ~
user 5m20.300s
sys 3m18.724s
fkaniwa@fkaniwa1:~$ cls
No command 'cls' found, but there are 18 similar ones
cls: command not found
fkaniwa@fkaniwa1:~$ clear

fkaniwa@fkaniwa1:~$ /home/fkaniwa/memusg ./KSA.py
Completed construction
True
memusg: vmpeak: 1137744 kb
fkaniwa@fkaniwa1:~$ time ./KSA.py
Completed construction
True

real 2m19.963s
user 2m18.890s
sys 0m0.968s
fkaniwa@fkaniwa1:~$ ls -l
total 30812
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Aug 21 12:55 Desktop
-rw-r--r-- 1 fkaniwa fkaniwa 48 Apr 9 15:05 dna1.fa
-rw-r--r-- 1 fkaniwa fkaniwa 31457280 Aug 31 2017 dna.fa
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Documents
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 4 07:30 Downloads
-rw-r--r-- 1 fkaniwa fkaniwa 8980 Apr 3 14:15 examples.desktop
-rwxrwxrwx 1 fkaniwa fkaniwa 3289 Apr 9 15:05 KSA.py
lrwxrwxrwx 1 fkaniwa fkaniwa 29 Apr 29 18:43 Link to Untitled Folder -> /home/fkaniwa/Untitled Folder
-rwxrwxrwx 1 fkaniwa fkaniwa 2794 Sep 4 2012 memusg
-rw-r--r-- 1 fkaniwa fkaniwa 31 Aug 16 23:53 memusg_readme.txt
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Music
-rwxrwxrwx 1 fkaniwa fkaniwa 3354 Aug 21 12:25 PaST.py
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Aug 21 12:55 Pictures
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Public
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 29 17:25 seqs
-rw-r--r-- 1 fkaniwa fkaniwa 2277 Apr 9 15:05 ST-based.py
-rw-r--r-- 1 fkaniwa fkaniwa 7569 Apr 9 15:05 suffix_tree.py
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Templates
-rw-r--r-- 1 fkaniwa fkaniwa 48 Apr 9 15:05 test.fa
-rwxrwxrwx 1 fkaniwa fkaniwa 504 Mar 18 21:01 test.sed
drwxrwxr-x 2 oracle oinstall 4096 Apr 12 22:42 tmpdatabases
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Videos
fkaniwa@fkaniwa1:~$
```

Running time and Peak memory
for the PaST algorithm

Appendix E

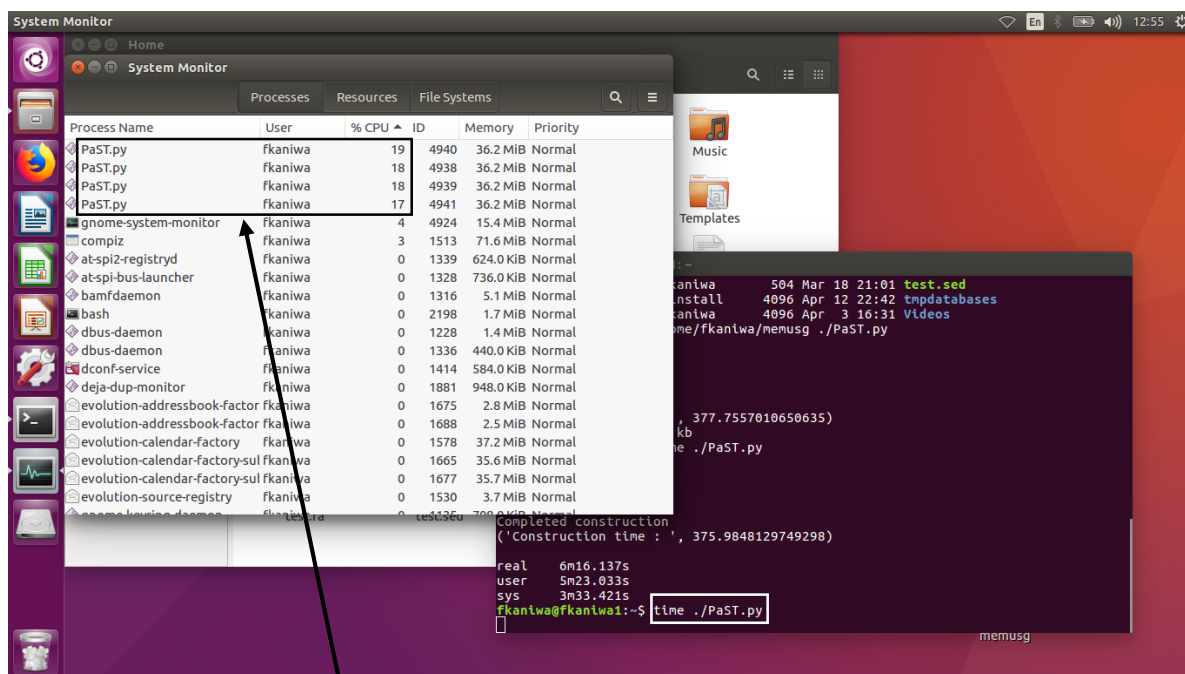
Sample Screenshot – PaST Algorithm

```
fkaniwa@fkaniwa1: ~  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Aug 16 23:39 Desktop  
-rw-r--r-- 1 fkaniwa fkaniwa 48 Apr 9 15:05 dna1.fa  
-rw-r--r-- 1 fkaniwa fkaniwa 31457280 Aug 31 2017 dna.fa  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Documents  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 4 07:30 Downloads  
-rw-r--r-- 1 fkaniwa fkaniwa 8980 Apr 3 14:15 examples.desktop  
-rwxrwxrwx 1 fkaniwa fkaniwa 3289 Apr 9 15:05 KSA.py  
lrwxrwxrwx 1 fkaniwa fkaniwa 29 Apr 29 18:43 Link to Untitled Folder -> /home/fkaniwa/Untitled Folder  
  
-rwxrwxrwx 1 fkaniwa fkaniwa 2794 Sep 4 2012 memusg  
-rw-r--r-- 1 fkaniwa fkaniwa 31 Aug 16 23:53 memusg_readme.txt  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Music  
-rwxrwxrwx 1 fkaniwa fkaniwa 3354 Aug 21 12:25 PaST.py  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Pictures  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Public  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 29 17:25 seqs  
-rw-r--r-- 1 fkaniwa fkaniwa 2277 Apr 9 15:05 ST-based.py  
-rw-r--r-- 1 fkaniwa fkaniwa 7569 Apr 9 15:05 suffix_tree.py  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Templates  
-rw-r--r-- 1 fkaniwa fkaniwa 48 Apr 9 15:05 test.fa  
-rwxrwxrwx 1 fkaniwa fkaniwa 504 Mar 18 21:01 test.sed  
drwxrwxr-x 2 oracle oinstall 4096 Apr 12 22:42 tmpdatabases  
drwxr-xr-x 2 fkaniwa fkaniwa 4096 Apr 3 16:31 Videos  
  
fkaniwa@fkaniwa1:~$ /home/fkaniwa/memusg ./PaST.py  
False  
False  
False  
True  
Completed construction  
(Construction time : 377.7557010650635)  
memusg: vmpeak: 687584 kb  
fkaniwa@fkaniwa1:~$ time ./PaST.py  
False  
False  
False  
True  
Completed construction  
(Construction time : 375.9848129749298)  
  
real 6m16.137s  
user 5m23.033s  
sys 3m36.747s  
fkaniwa@fkaniwa1:~$
```

Running time and Peak memory
for the PaST algorithm

Appendix F

Sample Screenshot – Parallel Processing



Four (4) processes running independently on each processor respective to the DNA alphabet for the PaST algorithm

