AN EXPERIMENTAL STUDY ON NON-OVERLAPPING INDEXING

by

Leon Zhou

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2026

© Copyright by Leon Zhou, 2026

Table of Contents

List of Tables . . . . . **iv**

List of Figures . . . . . **v**

Abstract . . . . . **vi**

Acknowledgements . . . . . **vii**

Chapter 1 Introduction **1**

1.1 Previous Work 2

1.2 Our Work 3

1.3 Organization 3

Chapter 2 Preliminaries **5**

2.1 Machine Model 5

2.2 Notation and Definitions 5

2.3 Suffix Tree 6

Chapter 3 Non-Overlapping Indexing in Linear Space and Optimal Time **10**

3.1 Aperiodic Patterns 10

3.1.1 Radixsort Based On Suffix Array Partitioning 13

3.2 Periodic Patterns 17

3.2.1 Data Structure For Periodic Patterns 17

3.2.2 Proving Linear Space Usage 21

Chapter 4 Experiments and Results **22**

4.1 Implementation 22

4.2 Experimental Setup 22

4.3 Experimental Results 27

4.4 Conclusion 29

Chapter 5 Conclusions **34**

Bibliography **35**

List of Tables

3.1 Multi-level Partition of The String ACADABABABABABA$. The shadings indicate different blocks within a level. 15

4.1 Three solutions for pattern searching of which the suffix tree ones are used. 23

4.2 The datasets used in the experiments 24

4.3 Overall ratio of the occurrences to non-overlapping occurences . 26

4.4 Percentage of patterns that occur in each text 26

4.5 Time to run pattern search on query pattern sets of size 10000. The times shown are in milliseconds per query. 28

4.6 Construction time (in min) of the suffix trees implementations . 28

4.7 Space costs (in bytes per symbol) of the suffix trees for the texts on length 250 million 29

List of Figures

2.1 An example to illustrate the notation on trees 6

2.2 A suffix tree 7

2.3 A suffix tree of a string with no '$' added and missing leaf nodes 5 and 6. 8

3.1 Suffix tree for string ACADABABABABABA$. The index on a leaf is the starting position of its corresponding suffix in the text. The locus node of the string 'BAB' is shaded in gray. . . 14

3.2 Suffix Tree For String ACGACACGACACGAGAGCACACGA-CACGACACGACAC$ 19

4.1 Space and time performances on longprd 30

4.2 Space and time performances on random 31

4.3 Space and time performances on medprd 32

4.4 Space and time performances on shortprd 33

Abstract

In this thesis, we perform an experimental study on solutions to the non-overlapping indexing problem. In this problem, we are given a text T, and our task is to build an index of T, such that, given a pattern P, we can efficiently report the maximal set of non-overlapping occurrences of P in T. Our experimental study shows that the theoretical optimal solution of Cohen and Porat (Range Non-overlapping Indexing, ISAAC 2009) indeed speeds up queries significantly compared to the strategy of applying a greedy algorithm to the occurrences returned by a suffix tree.

Acknowledgements

I would like to thank my supervisor Dr. Meng He for constantly giving me professional guidence. He often provided multiple ways of solving problems as well as backup plans in case one plan did not work. I am grateful to be mentored by such a kind, patient and professional supervisor. I would also like to extend gratitude to Dr. Kaiyu Wu for proofreading my thesis, as well as Dr. Peter Bodorik and Dr. Alexander Brandt for serving on my defense committee. I appreciate and acknowledge the support of my family for their encouragement and the financial support during my graduate studies.

Chapter 1

Introduction

In any context of managing growing amounts of text, we are often interested in retrieving the occurrences of a pattern string in a document or in a large text. Some common scenarios where there is a growing need of text search are encyclopedias and biological text such as deoxyribonucleic acid (DNA) sequences. In addition to telling us where substrings appear in the text, in cases such as in text processing, it also enable solutions to a multitude of problems like document retrieval [25], plagiarism detection [5] and related problems such as authorship attribution as well as many more problems. In addition to obtaining all pattern occurrences, researchers have also studied the reporting of a maximum set of pattern occurrences that do not overlap [14, 21]. In applications like biological sequence analysis and text compression, non-overlapping occurrences of patterns are of high importance.

Regarding the problem of *non-overlapping indexing*, which builds a text index that facilitates the reporting of the non-overlapping occurrences of a query pattern, the solution of Cohen and Porat. [14] achieves a query time of $O(m + nocc)$ which is theoretically optimal, where m is the length of the query pattern and $nocc$ is the number of non-overlapping occurrences reported. The space usage of the solution is $O(n)$ words where n is the length of the text. Some other theoretical results were previously achieved which have reasonable tradeoffs between space usage and query time [21]. There are many experiments done on various text indexing problems in which overlapping occurrences are reported [20, 36, 3]. Surprisingly, there is little indication of experiments done on the optimal theoretical result of the problem of non-overlapping indexing. Hence, experiments are needed to be done in this study to evaluate the performance of the optimal solution of Cohen and Porat [14] in practice.

1.1 Previous Work

We now go over results on non-overlapping indexing in which n and m denote the length of the text T and the pattern P, respectively. The number of non-overlapping occurrences of P in T is denoted by *nocc*. First and foremost is the work by Cohen and Porat [14] that is implemented in the experiments of this study. It uses $O(m + nocc)$ query time with a space usage of $O(n)$ words and a preprocessing time of $O(n \lg \lg n)$. A solution by Ganguly et al. [21] uses a compressed text index CSA of T that, without requiring access to T, can answer a pattern query P in $search(P) + sort(nocc) + O(nocc \cdot t_{SA})$ query time and $O(n \lg \sigma)$ bits of space, where σ is the alphabet size of T, t_{SA} is the time needed to access an arbitrary entry of the suffix array or the inverse suffix array of T using CSA, $search(P)$ is the time needed to compute the suffix range of P, and $sort(k)$ is the time for sorting k numbers in $\{1, 2, ..., n\}$.

An extension to the non-overlapping indexing problem is the problem of *range non-overlapping indexing* where we are to return the number of non-overlapping occurrences of P in $T[i, j]$ for a pattern P and a range $[i, j]$ given at query time. This problem is solved by Lewenstein et al. [28] using $O(m + nocc_{ij} \lg \lg n)$ query time and $O(n \lg n)$ words of space, where $nocc_{ij}$ is the number of non-overlapping occurrences of P in T in the index range $[i, j]$. This problem is also solved by Cohen and Porat [14] using query time $O(m + \lg \lg n + nocc_{ij})$ in $O(n \lg n)$ space. In Ganguly et al. [21], a full-text index is created that takes $O(n(\lg n)^{\varepsilon})$ words of space, where ε is an arbitrary positive constant, so that when given a pattern P and a query range, the number of non-overlapping occurrences of P in the range can be reported in $O(m + nocc_{ij})$ time.

The problem of returning the number of non-overlapping occurrences of pattern P in the text T (without reporting them) is known as the *string statistics* problem. This problem was solved by Brodal et al. [12] using the Minimal Augmented Suffix Tree (MAST). The MAST is obtained from the suffix tree using augmentation rules. The construction time of the MAST is $O(n \lg n)$ and the space used is $O(n)$ words. The query time is $O(m \lg \sigma)$ for a σ-sized alphabet which is desirable when *nocc* is large as the query time does not depend on *nocc*. MAST, however, does not report occurrences of P in T.

1.2 Our Work

In this thesis, experimental studies were done to implement seq and radix which are based on the work of Cohen and Porat [14]: radix does all the sorting of pattern occurrences by radixsort using a multi-level grouping strategy of the entries of the suffix array. seq then uses the method of radix on aperiodic patterns but uses precomputed sequence lists to get non-overlapping occurrences of periodic patterns (the definition of periodic pattern will be given in Section 2.2). These two implementations are compared with st which uses a suffix tree to collect the pattern occurrences and then sorts them simply using quicksort, after which non-overlapping occurrences are retrieved through a greedy iterative algorithm. DNA strings are used for our experimental string data.

In this study on experiments of getting the pattern non-overlapping occurrence information, the solution that uses quicksort to sort pattern occurrences, st, has noticably slower query times than the advanced solutions radix and seq. seq also has slightly faster query time than radix. In practice on regular texts where there are not a lot of periodic substrings, radix achieves a balance between space usage and query time.

Before running this experimental study, we also conducted some theory work. For the radixsorting approach of Cohen and Porat [14], we designed a variant that is practical for implementation. For their approach on speeding up queries on periodic patterns, we worked out their omitted details of ensuring linear space and also utilized a newer research result called Run's Theorem [7] to simplify space analysis. These are also presented in this thesis.

1.3 Organization

Chapter 2 elaborates on some of the previous works that are related to string searching. Other background information such as the notation and machine model will also be introduced. Chapter 3 goes over an implementation of an optimal solution to the problem of non-overlapping indexing based on the work of Cohen et al. [14] that differentiates whether a pattern is aperiodic or periodic. Chapter 4 first introduces us to the setup used in the experiments like the source of texts and query pattern set

inputs as well as the theory that are tested in the experiments. It then goes over everything relevant to the experiment such as the results, implications and explanation as well as analysis and conclusions. Chapter 5 concludes the thesis.

Chapter 2

Preliminaries

This chapter provides the background information needed to understand the work on non-overlapping indexing starting with the machine model as well as notation and definitions. It then goes over the data structures, algorithms and concepts used in existing solutions. It will also provide history of the suffix tree and mention operations that are done on it.

2.1 Machine Model

The model used in the thesis is the word RAM (random access machine) model. Under this model, one word has $w = \Theta(\lg n)$ bits. Reading and writing word-sized operands as well as bitwise and arithmetic operations on them are constant time operations. One feature of the model is that it allows random access at all times. If, say, an array has N elements and each element uses w bits, one can perform read, write, arithmetic, and bitwise operations on any element in constant time.

2.2 Notation and Definitions

In this thesis, unless stated otherwise, $\lg n$ means base-2 logarithm with argument n. $gcd(u, v)$ of two numbers u and v is defined to be the greatest common divisor of u and v. For the problem of text search, m denotes the pattern length, and n denotes the corresponding text length unless otherwise specified. σ denotes the alphabet size.

To describe solutions to non-overlapping indexing, we additionally define the following terminology. Any string written as R^a means a non-empty string R repeated (or concatenated one after another) a times where a is a nonnegative integer. For non-empty strings t and y, $t.y$ means t concatenated with y. The *period*, P, of a string S is a substring of S such that S can be written as $P^a P'$ where P' is a (possibly empty) proper prefix of P and a is an integer greater than or equal to 1. For

example, if S = 'CARCARCA', then P = 'CAR', P' = 'CA' and $a = 2$. With these definitions, we define S as a *periodic string* if $a \geq 3$ or $a = 2$ and P' is non-empty (which means it is possible to have three occurrences of S in a text string that overlap with each other) or it is *aperiodic* otherwise. For example, the strings 'AGATAGAT' and 'GTACGA' are aperiodic. The strings 'ATGATGATG' and 'TACTACT' are periodic. An example of a text that has 3 overlapping occurrences of 'ATGATGATG' is 'CGATGATGATGATGATGCC'.

Some definitions on parts of a tree are needed as well. The *leaf size* of a node, v, is the number of leaves that has v as its ancestor. The *subtree size* of v is the number of nodes in the tree rooted at v. The *level* of v is the number of nodes on the path from v (inclusive) to the root (exclusive). For example, in Figure 2.1, the leaf size of node y is 2. The subtree size of node x is 10. The level of node z is 2.

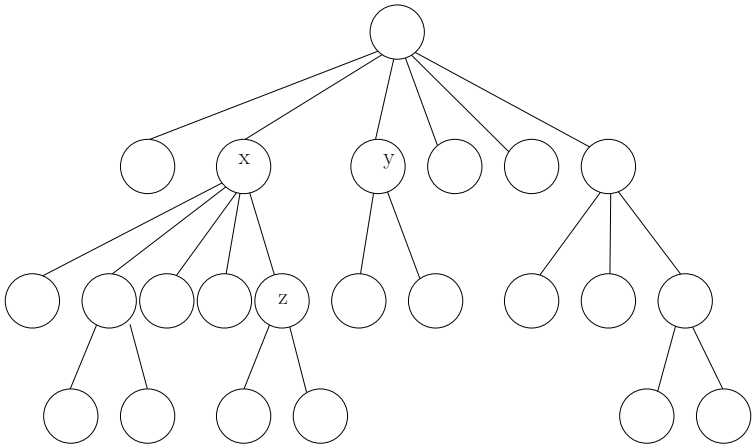


Figure 2.1: An example to illustrate the notation on trees

2.3 Suffix Tree

The suffix tree is an invaluable data structure for string searching and indexing. It is a compact trie that stores all the suffixes of a string in its leaf nodes. A trie is a tree data structure for accessing the keys stored to get their corresponding values. The compactness means that every node (except possibly the root) is branching. Each edge in the suffix tree is labeled with a nonempty substring of the text, which can be encoded by their starting and ending positions in T (see Figure 2.2). Each node x is associated with a string, $l(x)$, that is the concatenation of the substrings appearing

on the path from the root of the suffix tree to node x. The leaves of the tree represent suffixes of the string, and the j^{th} leaf from left to right is the j^{th} smallest suffix in lexicographical order. The keys are the suffixes of the string which are associated with the leaves. The values are the indices of the starting positions of the suffixes in the string. Therefore, the keys and corresponding values that a node stores can be obtained by getting the leaves in the subtree of that node.

In order to let all the suffixes be represented as leaves, the string has a '$', which is not a character that appears anywhere else in the string, appended to the end after which the suffix tree is constructed. This is because if '$' is not added, suffixes may be represented by internal nodes if they are contained in larger suffixes.

Also see the work of Farach [17] on the issues of '$' not being appended. As an example, see Figure 2.2 and Figure 2.3, respectively, for examples of the suffix tree of the word 'AGATCAT' with and without '$' appended. In the second example, a triangle at a point on the suffix tree indicates that the substring from the root node to that point forms a suffix.

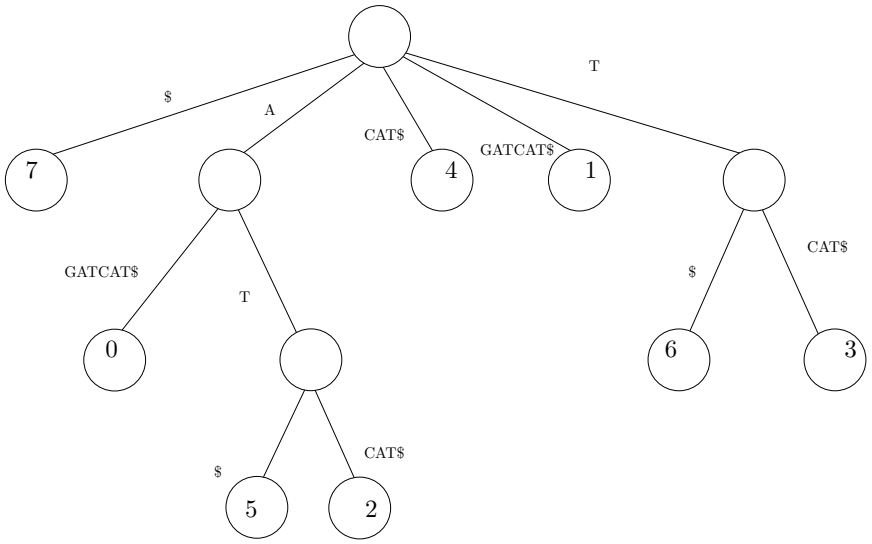


Figure 2.2: A suffix tree

Finally, if we store the leaf labels, which are starting positions of their corresponding suffixes of T, from left to right, in an array, we would obtain the suffix array, SA, of T. For example, the suffix array of T = 'AGATCAT$', whose suffix tree is shown in Figure 2.2, is $SA = \{7,0,5,2,4,1,6,3\}$. By this definition, we can see that if $SA[i] = j$,

then the i-th smallest suffix of T in lexicographic order among all suffixes of T, starts at position j in T.

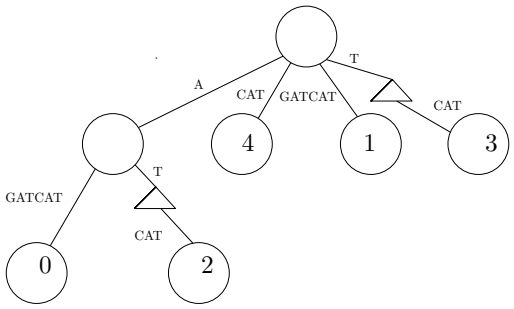


Figure 2.3: A suffix tree of a string with no '$' added and missing leaf nodes 5 and 6.

History of The Construction of The Suffix Tree

In 1973, Weiner presented the $O(n \lg n)$-time algorithm for constructing suffix trees [37]. Subsequent improvements were made by McCreight in 1976 [23] to build a compressed trie of all the suffixes of a string S and in 1995 by Ukkonen [35] to provide an online construction of suffix trees. The latter uses $O(n)$ words of space and $O(n)$ construction time for a constant sized alphabet and $O(n \lg n)$ construction time generally. In 1997, Farach gave the first suffix tree construction algorithm which uses $O(n)$ optimal time for any integer alphabet [17].

Implementations and Operations on The Suffix Tree

Definitions are needed in this context. Let us go over the definition of the locus node of a string S in a suffix tree st of T, where T is a string. The *locus node* of a string S is the highest node s such that $l(s)$ is prefixed by S if S is contained in T, or null if such a node does not exist. To find the number of non-overlapping occurrences of a pattern, P, we find the locus node of P. This is done by following the characters of P in the trie until its locus node u is found. If u does not exist, we report that the number of occurrences and non-overlapping occurrences of P is zero. Otherwise, we get all the occurrence indices of P in the string by retrieving the suffixes stored in the leaf descendants of u and keep track of the number of occurrences, *occ*.

To get the number of non-overlapping occurrences, *nocc*, the occurrence indices

must be sorted. After sorting is done, to get the number of non-overlapping occurrences, we find the next non-overlapping occurrence index that does not overlap with the last one found by going through the sorted list taking another $O(occ)$ time. Hence, the indices are sorted using quicksort in $O(occ \lg occ)$ time.

Chapter 3

Non-Overlapping Indexing in Linear Space and Optimal Time

In a work done previously by Cohen and Porat [14], one optimal solution in terms of time and space was achieved for non-overlapping indexing. Their data structure uses $O(n)$ space and answers a query in $O(m + nocc)$ time, where m is the pattern length, n is the text length and $nocc$ is the number of non-overlapping occurrences of the pattern in the text. In their work, different solutions were provided to deal with the periodic and aperiodic pattern types separately, which we will describe in Section 3.1 and Section 3.2 separately, with additional examples. The lemmas and theorems are restated from their paper, though Section 3.1.1 describes an alternative practical approach for sorting occurrences of aperiodic patterns which we designed. Additionally, Section 3.2 goes into the details on how to design the data structures to actually store period sequences in linear space, which is omitted in their paper, and we further simplify the space analysis using Run's Theorem [7].

3.1 Aperiodic Patterns

When the query pattern P is aperiodic, the suffix tree is used to get all the leaves that represent occurrences of the given pattern. After we get the occurrence indices from the corresponding leaves, we need to remove the overlapping occurrences. This can be done by sorting the indices and then going over the sorted list and filtering for overlapping occurrences. However, for any integer occ, sorting $O(occ)$ items takes $O(occ \lg occ)$ time. In order to optimize the sorting time, we use a modified radixsort method to be described next. There are two ways to use the radixsort. One method is based on leaf nodes of a subtree, and the other method divides the suffix array into groups using grouping size g where g takes on different values. Each value of g corresponds to a different grouping level. The subtree method, which can be found in Theorem 1 of Cohen and Porat [14], is their proposed solution that needs

modifications when sorting. As will be seen, the method using the suffix array division for the radixsort offers a more practical solution than that using the subtree.

Let us first go over the original solution of Cohen and Porat [14] for aperiodic patterns. The following lemma shows that the number of non-overlapping occurrences of an aperiodic pattern in T is linear in the total number of its occurrences.

Lemma 1. *For an aperiodic pattern, $occ = O(nocc)$. Specifically, $occ/2 \leq nocc$.*

Proof. Let P be an aperiodic pattern. For this proof, we show that the distance, d, between two consecutive occurrences of P must be greater than or equal to a half of the length, m, of P.

Assume to the contrary that there exist two consecutive occurrences of P in T whose starting positions differ by d with $2d < m$. Let t be the starting position of the first of these two occurrences of P in T. Let r be the substring $T[t..t+d-1]$. We prove that P can be written as $r^a.r'$ where $a = \lfloor m/d \rfloor$ and $r' = P[ad..m-1]$ (which is an empty string if $m = ad$) is a prefix of r. The proof will be done using induction on i to show that $P[id..(i+1)d-1] = r$ for $i = 0, 1.., \lfloor m/d \rfloor - 1$. In the base case, $i = 0$ and $P[id..(i+1)d-1] = P[0..d-1] = T[t..t+d-1] = r$. As an inductive hypothesis, assume that $P[id..(i+1)d-1] = r$ for some integer i, $0 \leq i < \lfloor m/d \rfloor - 1$ and we prove it for $i+1$. Since $T[t..t+m-1]$ and $T[t+d..t+d+m-1]$ are both occurrences of P, the two length-d substrings of T with starting position $t+id$ and $t+id+d = t+(i+1)d$, respectively contained in these two occurrences, must also match. This precisely means that $r = P[id..(i+1)d-1] = P[(i+1)d..(i+2)d-1]$ which is what we would like to prove.

To show that r' is a prefix or r, we can first use similar reasoning to show that $P[(a-1)d..(a-1)d+|r'|-1] = P[ad..m-1] = r'$. Then since $P[(a-1)d..(a-1)d+|r'|-1]$ prefixes $r = P[(a-1)d..ad-1]$, r' is a prefix of r.

This means that P can be written as $r^a.r'$. Given the assumed value of d, $a = \lfloor m/d \rfloor \geq 2$. There are two cases where in the first case, $a > 2$ and in the second case $a = 2$. In the first case, since r repeats at least three times, P must be periodic which is a contradiction. In the second case, r' must be non-empty because if it is empty then $d = m/2$, which conflicts with our assumption of d. A non-empty r' when $a = 2$ also makes P periodic. This contradiction means that the distance between two consecutive occurrences of P must be a half of the length of P rounded up or

larger. If one looks at every other occurrences of P starting from the first occurrence, they will be non-overlapping as they are separated by at least m characters. Hence, throughout the text string, $nocc$ will be at least $\lceil occ/2 \rceil$. Hence, $occ = O(nocc)$ which is what we set out to prove. □

We will use Lemma 1 for the time bounds of Lemma 2.

Lemma 2. *All occurrences of an aperiodic pattern P in a text T can be reported and sorted in occurrence order in $O(m+nocc)$ time using $O(n)$ space where m is the length of P, n is the length of T, and $nocc$ is the number of non-overlapping occurrences of P in T.*

Proof. Given a text, T, and a query pattern, P, we first use the suffix tree of T to get all the occurrences, occ, of P in $O(m + nocc)$ time where $nocc$ is the number of non-overlapping occurrences of P in T. Note that since $occ = O(nocc)$ by Lemma 1, $O(occ)$ and $O(nocc)$ can be used interchangably. For sorting all the occurrences, using radixsort, a renaming method on the suffix tree is needed. Here, the left-to-right order of the leaves of the suffix tree is consistent with the lexicographic order of the corresponding suffix among all suffixes of T. This can be thought of as a suffix array since the location that the i^{th} leftmost leaf contains is exactly $SA[i]$. Saving each such location takes $\lg n$ bits since there are n leaves.

The terms positional occurrences, starting positions and occurrence indices will be used interchangably. To sort the occurrence indices in $O(occ)$ time, we store for each leaf the rank of the occurrence index in T of the suffix that it represents in different subtrees containing it. These subtrees have different numbers of leaves. The largest subtree has $n^{1/2}$ leaves, the next smaller one has $n^{1/4}$ leaves, then the next smaller has $n^{1/8}$ and so on until the smallest subtree has a constant number of leaves in it. For each leaf, its rank in the subtrees containing it are packed into words and stored in that leaf. Let us take a look at the space usage. In each leaf, we store: $\lg n$ bits for its corresponding suffix array entry, $\lg n^{1/2}$ bits for its rank in the subtree with $n^{1/2}$ leaves including this leaf, $\lg n^{1/4}$ bits for its rank in the subtree with $n^{1/4}$ leaves including this leaf, ... This sums as follows per leaf: $\lg n + \lg(n^{1/2}) + \lg(n^{1/4}) + \lg(n^{1/8}) + ... \leq 2 \lg n$ bits per leaf. These entries can be packed into a constant number of words and the needed rank value can be extracted using a constant number

of bit operations. Since we have n leaves, in total $2n \lg n = O(n \lg n)$ bits of space or $O(n)$ words of space are used.

If we want to get the occurrences of P sorted by their positions in T and if there are occ occurrences, we find the node, n_{st}, such that the subtree rooted at n_{st} is the smallest subtree that has $n^{2^{-l}}$ leaves for some integer l which contains all the occurrences of P. This subtree has up to occ^2 leaves. n_{st} can be found in $O(m)$ time. Note that this is because the locus node of P is either n_{st} or a descendant of n_{st} since all the occurrences of P are in the subtree of n_{st}. One needs to collect the ranks of the occurrence indices of P in the occ leaves that correspond to the occurrences of P. As n_{st} has at most occ^2 leaves, each rank is in $[1, occ^2]$. Radixsort can sort these ranks in $O(occ)$ time since it can be used to sort z numbers from the universe $[1, z^d]$ in $O(z)$ time for any positive constant integer d; the key is that z is used as the base of the numbers in the sorting algorithm instead of base 10 so that each number can be encoded using a constant number, d, of digits. The sorted suffix array values as well as the number of occurrences and non-overlapping occurrences can then be retrieved in the sorted list of leaves in $O(occ)$ time. Hence, the query time is $O(m + occ)$.

Hence, the time and space bounds claimed in this lemma are proved. □

However, there is a shortcoming to this subtree approach. The issue here is that for a leaf node L, there may not exist an ancestor, A, of L such that the size of the subtree rooted at A is close to $n^{2^{-l}}$ for a certain positive integer l. For example, an ancestor A_1 of L has $n^{1/64}$ leaves in its subtree and A_1 has a sibling A_2 that has $n^{1/4}$ leaves in its subtree. The issue that could occur is that there may not exist an ancestor of L with the number of leaves between $n^{1/64}$ and $n^{1/4}$ in its subtree. This gap can get arbitrarily large and in those cases it would be very difficult to find a good subtree size that contains L. There does not seem to be an easy fix. Thus, we use the modified radixsort approach below which partitions the suffix array recursively to address this.

3.1.1 Radixsort Based On Suffix Array Partitioning

In our alternative approach, we perform $O(\lg \lg n)$ levels of partitions of the suffix array. The level-1 partition divides the suffix array into $n^{1/2}$ blocks, each containing $n^{1/2}$ suffix array entries. The level-2 partition then divides the suffix array into $n^{3/4}$

blocks, each of size $n^{1/4}$, and so on. In general, the level-l partition divides the suffix array into $n^{1-2^{-l}}$ blocks of size $n^{2^{-l}}$. This partitioning is done until the block size is a constant. We assume $n^{2^{-l}}$ is an integer for simplicity on any level l.

We then define a conceptual array $LGA[1...L][0..n-1]$ where L is the number of partition levels. In particular, $LGA[i][j]$ stores the relative rank of the occurrence index of $SA[j]$ among the entries in the level-i block containing it, i.e. $SA[j]$ is the $LGA[i][j]$-th smallest entry in block $SA[\lfloor j/n^{2^{-i}} \rfloor n^{2^{-i}}..(\lfloor j/n^{2^{-i}} \rfloor + 1)(n^{2^{-i}}) - 1]$. Let us analyze the space usage of LGA. For any suffix array entry, $\lg(n^{1/2})$ bits are needed to store its level 1 rank within its interval in LGA, $\lg(n^{1/4})$ bits are needed to store its level 2 rank within its interval and so on. This sums as follows for this suffix array entry: $\lg(n^{1/2}) + \lg(n^{1/4}) + \lg(n^{1/8}) + ... \leq \lg n$ bits. Hence, to store all the suffix array rank values in LGA, up to $n \cdot \lg n$ bits $= O(n)$ words are used. See Table 3.1 for an example of the suffix array and LGA for the string 'ACADABABABABABA$' where blocks of size 2 are not partitioned further. Figure 3.1 shows the suffix tree of the string.

In the experimental implementation, each leaf additionally stores the integer value

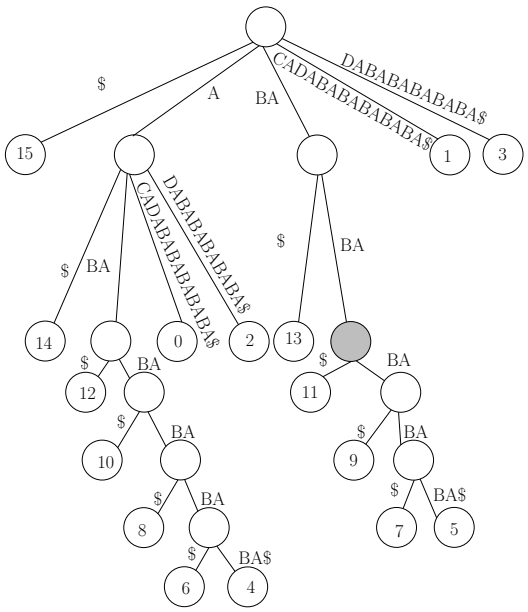


Figure 3.1: Suffix tree for string ACADABABABABABA$. The index on a leaf is the starting position of its corresponding suffix in the text. The locus node of the string 'BAB' is shaded in gray.

Table 3.1: Multi-level Partition of The String ACADABABABABABA$. The shadings indicate different blocks within a level.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$SA[i]$	15	14	12	10	8	6	4	0	2	13	11	9	7	5	1	3
$LGA[1][i]$	3	2	1	0	3	2	1	0	0	3	2	1	3	2	0	1
$LGA[2][i]$	1	0	1	0	1	0	1	0	0	1	1	0	1	0	0	1

$SA[v]$ if it is the v^{th} leaf from the left or equivalently if the suffix that it represents is the v^{th} smallest suffix lexicographically. This index is needed so that we can get the range of suffix array entries that store the occurrence indices of P. One can first get *occ* after obtaining the locus node of P (by finding the number of its leaf descendants). Let us define integers a and b such that the starting positions of the occurrences of P are in $SA[a..b]$. Then the number of occurrences, *occ*, of P in T is $b-a+1$. Note that the smaller a level is, the bigger its corresponding block size will be. We need the largest level, l, such that $occ \leq n^{2^{-l}}$ ($n^{2^{-l}}$ is the block size); i.e. $\lfloor \lg \log_{occ} n \rfloor$. When l is chosen this way, the starting positions of the occurences of P are in one or two blocks in SA and radixsort is efficient by setting the base to be *occ* as to be shown later. Next we will show how l is used in LGA with query pattern P.

There are two cases to consider. The simpler case is when indices a and b are in the same level-l block of SA. To sort the occurrence indices of P, we can sort their corresponding ranks in $LGA[l][a..b]$ using radixsort after which the original sorted suffix array values can be obtained to filter for non-overlapping occurrences using the folklore greedy algorithm. The other case is when a and b are in different blocks. Here P will have occurrences in one interval from a to $a_h - 1$ where a_h is the next multiple of $\lfloor n^{2^{-l}} \rfloor$ bigger than a. The rest of the occurrences are then found in the next block at indices a_h, $a_h + 1$,..., b. Let us define occ_1 and occ_2 to be $a_h - a$ and $b + 1 - a_h$, respectively, which will be used in the query time analysis. In this latter case, the ranks in both blocks first need to be sorted separately. Then one can merge and recover the sorted suffix array values from the sorted ranks. Then filtering for non-overlapping occurrences is done by means of the folklore greedy algorithm.

Let us analyze the query time for this radixsort using LGA. Recall that, given that z is a positive integer, radixsort can be used to sort z numbers from the universe

$[1, z^d]$ in $O(z)$ time for any positive constant integer factor d. Hence, we set the radix sorting base to be $z = occ$. The ranks to be sorted are occ entries in $[1, n^{2^{-l}}]$ which are contained in $[1, z^2]$ since $n^{2^{-l}} \leq z^2 = occ^2$ (which means $d = 2$).

Hence, in the first case, radixsort will take up to $O(occ)$ time. Mapping to get the original suffix array values take another $O(occ)$ time. Thus, in the first case, $O(occ)$ time is needed in total. The same reasoning for the intervals being contained in $[1, z^2]$ is used in the second case too with sorting being done in two intervals.

Indeed, in the second case, doing radixsort on the first and second intervals each also takes $O(occ)$ time by using occ as the base for radixsort. Then merging and mapping the ranks to obtain the ordered suffix array values take another $O(occ)$ time. Hence, also in the second case, $O(occ)$ time is needed in total. Therefore, the query time of using radixsort with LGA will always be $O(occ)$. Before using LGA, $O(m)$ time is needed to find the locus node of P in the suffix tree, making the total query time be $O(m + occ)$.

To illustrate how this radixsort algorithm works when using this array-based method, let us take an example. Let us do a search for string pattern 'BAB' as a pattern for the text in Table 3.1. For the text example, 'ACADABABABABABA$', there are 4 occurrences of 'BAB' i.e., $occ = 4$. In Figure 3.1, the locus node of 'BAB' is shaded gray. Recall that we use the largest level, l, with the block size, $n^{2^{-l}}$, larger than or equal to occ. Level 1 has the block size, 4, that satisfies the requirement while level 2 does not. Hence, we use the level 1 block of LGA rank values. Here $a = 10$ and $b = 13$ which will be the interval of rank values needed from LGA. a and b are in two different blocks at level 1. Thus, we are in the second case. From the first block we get the ranks 2 and 1 and store them in a rank array R_1. From the second block we get the ranks 3 and 2 and store them in another rank array R_2. We then sort both rank arrays and then merge the corresponding suffix array values (which can be directly obtained by mapping from rank values) into another array ME in increasing order. The contents of ME are then $5, 7, 9$ and 11 in that order. From ME, one can get the number of occurrences, 4, and, more importantly, the number of non overlapping occurrences, 2, by way of the folklore greedy algorithm.

3.2 Periodic Patterns

For periodic patterns, we will now go through the $O(n)$ space and $O(m+nocc)$ query time solution of Cohen and Porat [14]. In their approach, they store precomputed information called period sequences which essentially mark consecutive occurrences of periods of potential periodic patterns. However, their paper does not go into the details on the design of a data structure that stores these period sequences in linear space. A naïve approach would result in too much space: as a cyclic shift of a period can still possibly be the period of another periodic pattern, one can get rid of the initial characters of a period sequence to record a sequence of consecutive occurrences of this shifted pattern, but storing both sequences can blow up the space easily. Thus, we worked out the details of one possible data structure solution for storing all such information in linear space. When doing so, we also utilize a newer result called Run's Theorem [7] which greatly simplifies the proof that these data structures use linear space. We will first present the data structures and show how the query time is achieved in Section 3.2.1. Then, we will analyze the space cost in Section 3.2.2.

3.2.1 Data Structure For Periodic Patterns

To show the space and time usages of the data structures in this section, some terms relevant to the suffix tree are needed. First, we give the notation of a string formed from the root to a given node. For a node u, we define $l(u)$ to be the string obtained by concatenating the edge labels in the path from the root to u. A node, w, in the suffix tree such that $l(w)$ is a periodic string is called a *period node*. In the suffix tree, the following is the definition of a *period path*. It is a path between the root and a period node, u, such that there is no period child node, v, of u such that $l(u)$ and $l(v)$ have the same shortest period. The *period path length* is the number of nodes in a period path. For a path, A, in a suffix tree, $L(A)$ is defined to be the string formed by concatenation of edge labels on A top down and $|L(A)|$ is the string length of path A.

Example 1. *For example, in Figure 3.2, the path starting from the root and ending at the leaf node labelled 18 is a period path with the period 'ACACG'. As an example of $l(u)$ for a node u shaded blue, l for that node is the string 'ACAC'. l for the node*

shaded red is the string 'ACACGA'.

We now discuss how to get all the periodic substrings contained in the texts and store them. For this some more terminology is needed. In a text string T, a subsequence, marked as $[s, e]$, where s and e respectively denote its start and end indices in T is said to be a *period sequence* of a period p if it is of the form $p'p^cp''$, where c is a nonnegative integer, satisfying the following conditions:

1. $e - s + 1 > 2|p|$;
2. p' is a possibly empty proper suffix of p;
3. p'' is a possibly empty proper prefix of p;
4. It cannot be extended left or right while preserving shortest period p.

The sequence length is $e - s + 1$. The *period length* of a period sequence is the length of the period of the sequence.

Let us see an example.

Example 2. *Set p = 'ACACG'. Let text T be 'ACGACACGACACGAGAGCACAC-GACACGACACGACAC', the example used in Figure 3.2. The period sequences for p are $[0, 13]$ and $[18, 36]$. For the period sequence, $[0, 13]$, the suffix p' is 'ACG' and the prefix p'' is 'A'. The length of this sequence is $13 - 0 + 1 = 14$. For the other period sequence, the suffix, p', is empty and the prefix, p'' is 'ACAC'. The length of this latter sequence is $36 - 18 + 1 = 19$.*

Let us now go over what data structures are used. First some terminology is needed. For each period sequence $[s, e]$ with shortest period p, define its *canonical period*, $\bar{p}$, to be $T[s..s + |p| - 1]$. Further, define L_q to be the list of all period sequences whose canonical period is q, in decreasing order of length. Let L and L' be two such lists of period sequences whose canonical periods are circular shifts of each other. L and L' are then said to be *related lists*. Let s_1 and s_2 be two strings where s_2 is a circular shift by r right rotations done to s_1. Then we say that the *offset* of s_2 with respect to s_1 is r.

We then store each group of related nonempty lists in a collection. For each such collection C, let L_1, L_2, ..., L_k be the lists in C. The offset, o_k, is stored at the

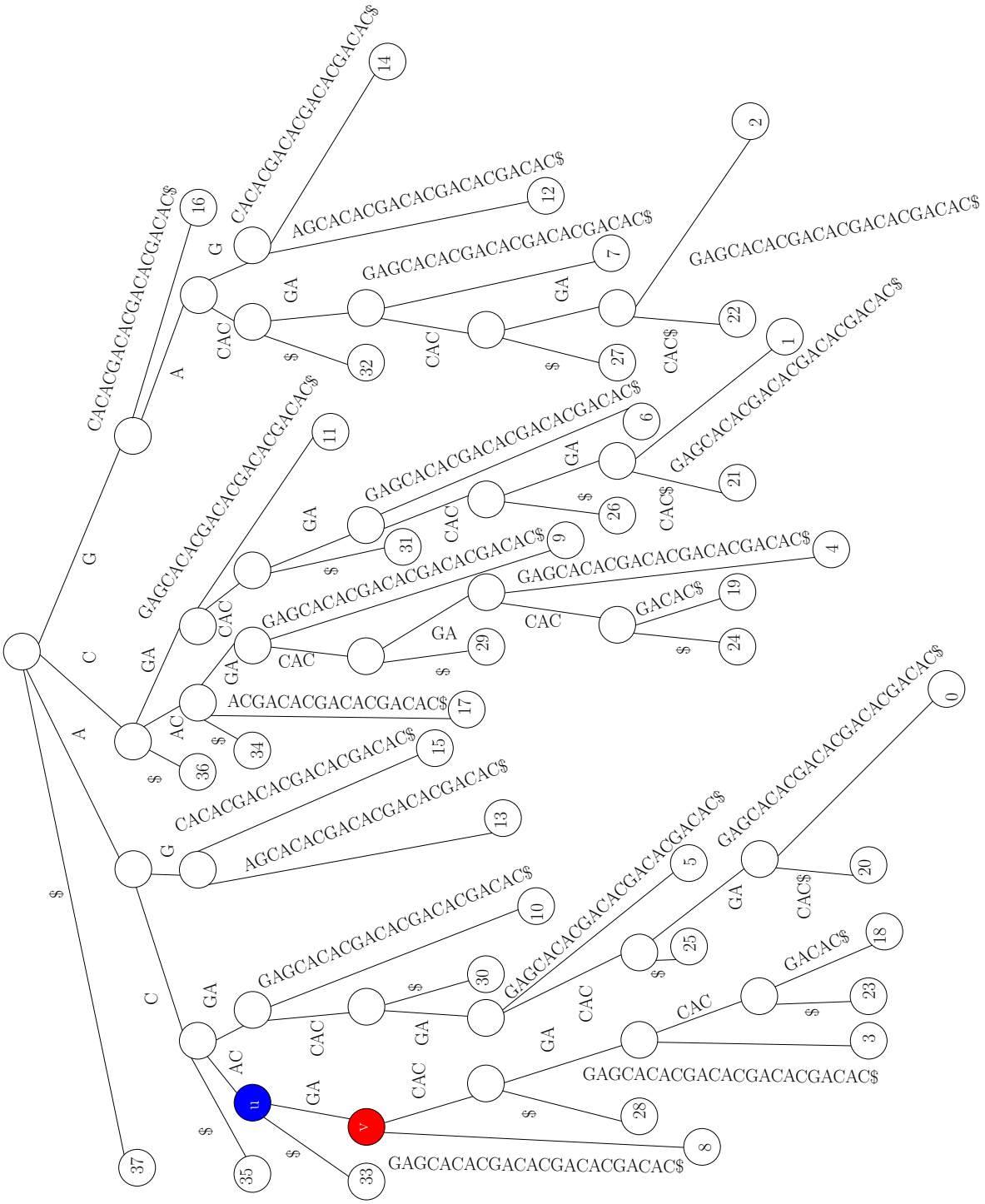


Figure 3.2: Suffix Tree For String
ACGACACGACACGAGAGCACACGACACGACACGACAC$

beginning of each L_k which is the number of right shifts done to the canonical period of L_1 to get the canonical period of L_k. Thus, $o_1 = 0$.

For each period node v, let P be the shortest period path containing v whose shortest period is p. Let C be the collection in which each list, L, of period sequences has p or a cyclic shift of p as its canonical period. Then we store in v a pointer to C.

Example 3. *Take T = 'ACGACACGACACGAGAGCACACGACACGACACGACACAC\$', the example used in Figure 3.2. Let us take a look at the collection, C, that is pointed to by the node shaded red, v. The following is the content of the collection C that the pointer of node v points to: List L_1 (with p_1 = 'ACGAC' as its canonical period) has 0 as offset and contains $\{[0, 13]\}$. List L_2 (with p_2 = 'ACACG' as the canonical period) has 2 as offset and contains $\{[18, 36]\}$.*

The following lemma goes over the time of retrieving all the non-overlapping occurrences of a periodic pattern using the collection containing all the period sequences of the period of that pattern.

Lemma 3. *Let p be the shortest period of a periodic pattern P. Let C be the collection such that the canonical period of each of its lists is p or a cyclic shift of p. Then, all the non-overlapping occurrences of P can then be retrieved from this collection in $O(|p| + nocc)$ time where nocc is the number of non-overlapping occurrences.*

Proof. Consider a periodic sequence $[s, e]$ with period length pl, and a periodic pattern P whose length is m. If the canonical period of $[s, e]$ is the same as the shortest period of P, then the starting positions of the non-overlapping occurrences of P in $[s, e]$ are of the form: $s+i \cdot step$, where $step = \lceil m/pl \rceil \cdot pl$ and $0 \leq \mathrm{i} \leq \lfloor (e - s + 1)/step - 1 \rfloor$, all of which are easily calculated. To use this, we first calculate the offset o_P of p with respect to the canonical period, p_1 of list L_1 in C by computing its starting index in $P_1 = p_1.p_1$ in $O(|p|)$ time using the KMP algorithm [29]. Let o_j be the offset stored with L_j. Once we know o_P, we will know which indices in each period sequence in any list L_j to start counting for $nocc$ which is calculated as $o_P - o_j$ if $o_j \leq o_P$ and at $p_l - (o_P - o_j)$ otherwise. All occurrences are reported in each period sequence whose sequence length is at least m. Since the number of all occurrences reported is the number of non-overlapping occurrences, the total time for reporting all the occurrences is $O(nocc)$ once o_P is calculated. □

We now show how pattern querying is done on a periodic pattern P. It was shown how a query time of $O(m + nocc)$ was achieved for a non-periodic pattern. We will now show how this time is achieved on a periodic pattern P.

Lemma 4. *Using the data structure described above, all non-overlapping occurrences of a periodic pattern can be retrieved in $O(m + nocc)$ time.*

Proof. Given a query pattern, P, we go to the locus node, v, of P in the suffix tree in $O(m)$ time. There are two cases. This first case is when v is a period node (or if $l(v)$ is a periodic string). In that case, we retrieve the collection that the pointer stored in v points to and apply Lemma to compute the non-overlapping occurrences of P in T.

In the second case, v is not a period node. Then P only has non-overlapping occurrences. In that case, we report all its occurrences in the text. In both cases, $O(m + nocc)$ time is used. □

3.2.2 Proving Linear Space Usage

We now prove that the space usage is linear. This is done by using the result of Bannai et al [7]. A period sequence as we have defined is a run in their work. By Run's Theorem, which they have proved, there are at most $O(n)$ runs that can be in a text string T of length n. Since our data structures represent each period sequence in $O(1)$ words, the total space cost is $O(n)$. Thus, we have the following result.

Theorem 1. *Given a text string T of length n, there is an $O(n)$-space data structure that can report the non-overlapping occurrences of any query pattern P in T in $O(m+nocc)$ time, where m is the pattern length and nocc is the number of non-overlapping occurrences.*

Chapter 4

Experiments and Results

This chapter presents the experiments on solutions to the non-overlapping indexing problem. It will compare and analyze the time and space trade-offs of the solutions. It will also test if the performance of the experiments match with how their theoretic bounds compare to each other.

4.1 Implementation

All the suffix tree constructions used in the experiments were done using the algorithm of Kärkkäinen et al. [27]. The suffix tree based solutions were implemented in three ways with the naive solution, st (Section 2.3), using the quicksort algorithm to sort the pattern occurrence indices. After sorting, each non-overlapping occurrence index is found by first reporting the first occurrence index and then using the greedy algorithm to report each occurrence index that does not overlap with the last reported occurrence.

The other two solutions are from Cohen and Porat [14]: radix uses their radixsort approach, as summarized in Section 3.1, to sort occurrences of any pattern before running the greedy algorithm, periodic or not, while seq is a full implementation of their solution, including both radixsort for aperiodic patterns and precomputed period sequences for periodic patterns.

Table 4.1 describes the solutions implemented in this thesis.

4.2 Experimental Setup

The strings used in experiments are DNA of viruses and bacteria (cov, eco and sal). See Table 4.2. We provide our experimental data online [1]. The alphabet size for our strings is four. They are the letters A for Adenine, C for Cytosine, G for Guanine and T for Thymine, which are the base nucleotides for representing DNA sequences

abbr.	description
st	This is a solution based on suffix trees that uses $O(n)$ words of space to answer queries in $O(n_{occ} \lg n_{occ} + \|P\| \lg \sigma)$ time. It sorts occurrences using the quicksort method; see Section 2.3 on suffix trees
radix	This is a solution based on suffix trees that uses $O(n)$ words of space to answer queries in $O(n_{occ} + \|P\| \lg \sigma)$ time. It sorts occurrences using radixsort for all patterns regardless of whether or not it is periodic; refer to Section 3.1 and the work of Cohen and Porat [14]
seq	This is a solution based on suffix trees that uses $O(n)$ words of space to answer queries in $O(n_{nocc} + \|P\| \lg \sigma)$ time. It sorts occurrences using radixsort on aperiodic patterns and, for periodic patterns, it goes through a sequence of indices to get the non-overlapping occurrences; refer to Chapter 3 and the work of Cohen and Porat [14]

Table 4.1: Three solutions for pattern searching of which the suffix tree ones are used.

which code the genetics for all organisms. However, the strings were obtained from files that had characters not in the alphabet; the text sources of cov, eco, and sal had alphabet sizes of 5, 6 and 11, respectively. Though the alphabets of the source for the texts are larger than four, only four letters ('A', 'C', 'G' and 'T') are selected as filtering out the other characters does not affect the integrity of the string DNA of the viruses string built. The characters that are not A, C, G or T represent bases that could not be identified as one of these characters. For example, Ns represent a gap. Other characters that are not 'A', 'C', 'G' or 'T' include 'U', 'N', 'R', 'Y', 'K', 'M', 'S', 'W', 'B', 'D', 'H' and 'V'. In the experiment, it was thought to leave them out to avoid ambiguity codes. Furthermore, as the alphabet size increases, the likelihood of getting a large number of occurrences with a big ratio of occurrences to non-overlapping occurrences diminishes. Hence, we stick to an alphabet size of four through this entire experiment. With 250 million characters needed for the strings, the number of characters gone through the original text lengths for cov, eco and sal were 250,004,300, 250,000,300 and 251,186,163, respectively due to filtering out character that were not 'A', 'C', 'G' or 'T'.

DNA Text	Length	Description
cov	250,000,000	DNA sampled from strains of streptococcus [2]
eco	250,000,000	DNA sampled from ecoli [2]
sal	250,000,000	DNA sampled from salmonella bacteria [2]

Table 4.2: The datasets used in the experiments

We constructed four sets of query patterns, shortprd, medprd, longprd and random, and each query set has 10,000 patterns. The first three sets consist of periodic patterns. To generate them, a parameter d is used to set the maximum period length of any generated pattern. All generated pattern sets have patterns generated by an integer variable, p, between (and including) 1 and d. Specifically, p is set to be the period length of a pattern. l is the length of this pattern which is between (and including) 6 and 20. d is set to $\lfloor l/2 \rfloor$, $\lfloor 2l/3 \rfloor$ and $\lfloor 3l/4 \rfloor$ to generate, respectively, the shortprd, medprd and longprd pattern sets. Algorithm 1 shows how each pattern set is generated given the parameter d.

Algorithm 1 works as follows. We initialize an empty set of patterns. Next, it will be described how a pattern is generated on any given iteration (for 10,000 iterations as set on line 3). Lines 4 and 5, respectively, generate the pattern length, l, and period length, p, where l is chosen randomly between 6 to 20 and p is chosen randomly between 1 and d. Lines 6 to 8 generate a number, M, between 0 and 4^p where a larger M corresponds to period string *temp* being lexicographical bigger. Lines 11 to 22 create *temp*. M can be thought of as a number in base 4 with p digits. From lines 11 to 22, a string is assigned to *temp* based on M. Specifically, for any integer index u, where $1 \leq u \leq p$, a '0' being the u^{th} digit of M means an 'A' as the u^{th} character of *temp*. Likewise, a '1' indicates a 'C', a '2' a 'G' and a '3' a 'T'. Then lines 24 to 27 create the pattern, *pattern*. Note that *pattern* is generated by repeating *temp* as *temp* is the period of *pattern*. In particular, the letter at index t, where $1 \leq t \leq l$, of *pattern* is $temp[(t\text{-}1)\%p\text{+}1]$. On line 28, *pattern* gets added to the pattern set, *patternset*.

The fourth query set, random, has pattern lengths from 6 to 20 with each character

Algorithm 1 Algorithm that generate pattern sets

Output: A set of patterns where each pattern has a period length at most d

1: $patternset \leftarrow \emptyset$ (empty array of strings)
2: $i \leftarrow 1$
3: **while** $i \leq 10000$ **do**
4: Generate a pattern length l, chosen uniformally at random, between 6 and 20
5: Generate a period length p, chosen uniformally at random, between 1 and d
6: $P \leftarrow 4^p$
7: $L \leftarrow$ large Integer, chosen uniformally at random, between 0 and 2147483647
8: $M \leftarrow (L\%P)$
9: $pattern \leftarrow emptystring$
10: $temp \leftarrow emptystring$
11: **while** $M > 0$ **do**
12: **if** M % 4 =0 **then**
13: Append 'A' to temp
14: **else if** M % 4 =1 **then**
15: Append 'C' to temp
16: **else if** M % 4 =2 **then**
17: Append 'G' to temp
18: **else**
19: Append 'T' to temp
20: **end if**
21: $M \leftarrow M/4$
22: **end while**
23: $j \leftarrow 0$
24: **while** $j \leq l - 1$ **do**
25: $pattern \leftarrow pattern + temp[j\%temp.length()]$
26: $j \leftarrow j + 1$
27: **end while**
28: add *pattern* to *patternset*
29: return *patternset*
30: **end while**

in the pattern generated randomly. Table 4.3 shows the ratio of occurrences to non-overlapping occurrences for the texts cov, eco, and sal and pattern sets shortprd, medprd, longprd and random. The *overlapping ratio* is obtained by dividing the sum of the total number of occurrences of all the patterns in the query set by the sum of the non-overlapping occurrences. Note that as d (the maximum period length of a pattern) increases, it makes less overlap of *pattern* with itself or lower periodicity which will result in a lower overlapping ratio. Furthermore, Table 4.4 shows the percentage of patterns that occur in each text. We also found that 84.50%, 51.57%, 43.66%, and 0% patterns in shortprd, medprd, longprd, and random are periodic strings, respectively.

Text	shortprd	medprd	longprd	random
cov	1.05719	1.03147	1.02896	1.00187
eco	1.04156	1.02087	1.01910	1.00168
sal	1.05315	1.02711	1.02486	1.00170

Table 4.3: Overall ratio of the occurrences to non-overlapping occurences

Text	shortprd	medprd	longprd	random
cov	28.15%	36.03%	38.71%	44.84%
eco	27.06%	35.09%	37.32%	44.86%
sal	27.94%	35.90%	37.88%	43.75%

Table 4.4: Percentage of patterns that occur in each text

Our experiments were carried out using a server with an Intel(R) Xeon(R) Gold 6234 CPU @ 3.30GHz and 128GB of RAM running at 2933 MHz. The kernel being run is a 4.15.0-54-generic x86 64 kernel. The C++ programs were compiled using g++ 7.4.0 with -O3 flags. All programs are coded without using non-standard libraries.

4.3 Experimental Results

Firstly, in the experimental study, three different solutions to non-overlapping indexing were implemented to see how the two advanced solutions, radix and seq, compared with each other and with the naive suffix tree solution, st, using quicksort. In these first experiments comparing all the solutions, 10,000 patterns are used from the four query sets (constructed in Section 4.2) in searches on 250 million length texts. Tables 4.4, 4.5 and 4.6 present the query time, construction time and space usage of all three solutions. The space cost is measured by bytes per symbol which is the total space usage in bytes divided by the length, n, of the string.

One will immediately notice that radix has much faster query time than st with st using about 1.3 more query time on average per query than radix. This is since quicksort which st uses takes time $O(occ \lg occ)$ whereas the radixsort used in radix takes time $O(occ)$. However, the space usage of radix compared to st is also about 1.34 times more.

It is seen that the period sequence based method, seq, performs noticably better than radix in the experiments on pattern sets shortprd, medprd and longprd as can be seen in Table 4.5. We go over some theory on how the distribution of periodic patterns affects the query times of radix and seq. As mentioned before, radix does radixsort on all patterns whereas seq uses a method based on precomputing period sequences on a pattern if it is periodic and does radixsort on the pattern otherwise. seq is generally thought to perform better than radix. This is because radix takes $O(m + occ)$ time where occ is the number of occurrences of the pattern and seq takes $O(m + nocc)$ time where $nocc$ is the number of non-overlapping occurrences of the pattern but they are close as seen in Table 4.3.

As expected, seq has slightly faster time than radix on pattern set random as the data in random has much lower overlapping ratio. This explains the small difference in query time since both solutions use the radixsort method on the more common aperiodic pattern cases. We see from Table 4.3 that the average ratio of occ to $nocc$ is not much greater than 1 (between 1.04 and 1.06) even for pattern set shortprd, and for random, the difference between occ and $nocc$ is within 0.2%. The space usage of seq is practically 1.1 times that of radix and as seen in Table 4.6, and radix only has a construction time which is about a quarter that of seq. This means seq may

Pattern Set	Text	st	radix	seq
shortprd	cov	0.28113	0.15749	0.15264
	eco	0.26092	0.14923	0.14639
	sal	0.26986	0.14984	0.14710
medprd	cov	0.46009	0.38234	0.36324
	eco	0.44093	0.35887	0.34013
	sal	0.44129	0.38304	0.35318
longprd	cov	0.51283	0.40812	0.38034
	eco	0.45193	0.39098	0.36192
	sal	0.46893	0.40919	0.38874
random	cov	0.84456	0.61286	0.60943
	eco	0.81091	0.60526	0.58132
	sal	0.77456	0.60238	0.60129

Table 4.5: Time to run pattern search on query pattern sets of size 10000. The times shown are in milliseconds per query.

Text	st	radix	seq
cov	1.6134	4.9049	18.4595
eco	1.5787	5.0407	19.3943
sal	1.5944	4.9915	19.5368

Table 4.6: Construction time (in min) of the suffix trees implementations

not be a necessary or an attractive solution in many experimental cases. With respect to construction times (Table 4.6), st has the least processing time since it constructs the suffix tree with no additional structures that the advanced solutions use. The construction time of seq is about four times more than radix on average since it implements the data structures for periodic sequences on highly periodic patterns in addition to the ones for radixsort. In our experiments, seq has construction time almost four times that of the radix solution on average.

Regarding the space usage (see Table 4.7), st uses the least amount of space since it does not use any other data structures other than suffix tree. radix uses a 2-d array where one dimension is used for each grouping level (Section 3.1.1). On average, radix uses about 68.6% more space than st. seq uses the data structures of radix but also

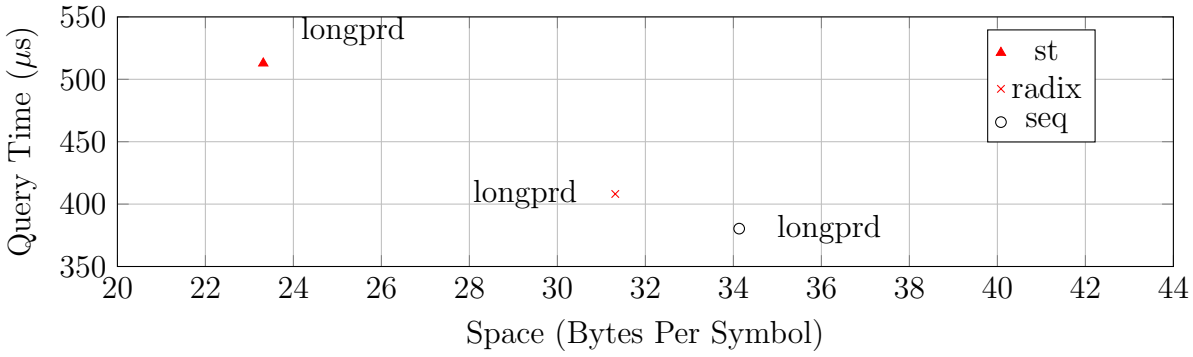
Text	Number Of Nodes	st	radix	seq
cov	489309535	23.3158	31.3158	34.1317
eco	490446149	23.3886	31.3886	34.1906
sal	489574138	23.3327	31.3327	34.1770

Table 4.7: Space costs (in bytes per symbol) of the suffix trees for the texts on length 250 million

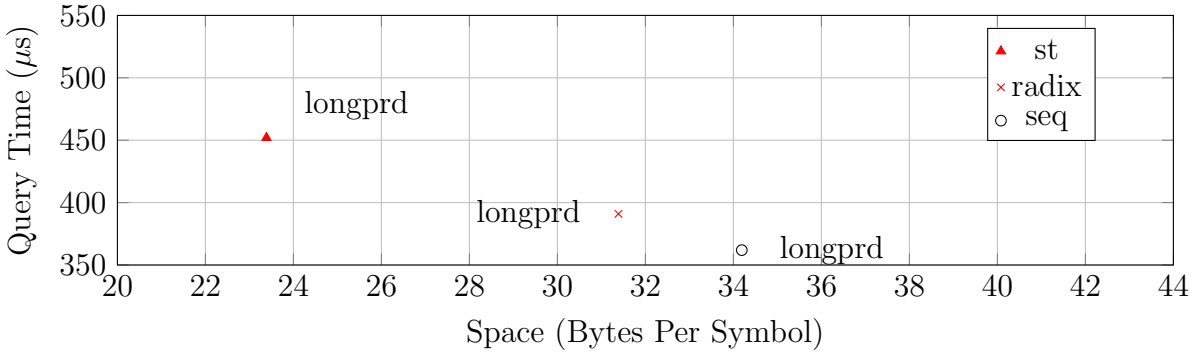
additional structures for periodic patterns, making it the implementation that uses the most space. If one takes a look at Figures 4.1, 4.2, 4.3 and 4.4 which plot the tradeoffs between query time and space cost, we see that in all cases st takes the longest average query time. It is also seen that seq invariably has the fastest average query time. However, due to the extra space usage of seq compared to that of rad, in most cases it may be preferable to use rad as can be seen in the figures. This is since in many practical cases, there may not be many periodic substrings in a text or periodic patterns as queries. Furthermore, the construction time of seq is huge compared to that of rad as can be seen in Table 4.6.

4.4 Conclusion

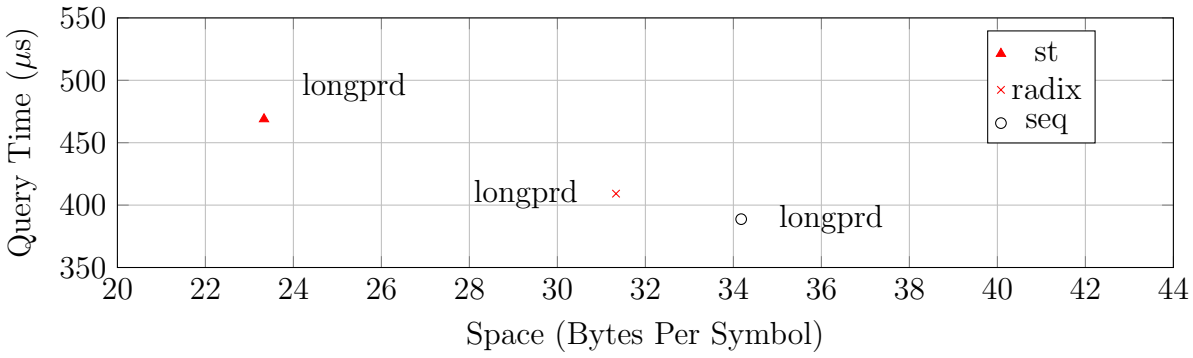
As hypothesized, the theoretical query time usage matches that in the experiments. For example, the query time of the seq solution is a bit faster than the radix solution and both advanced solutions have query time that are noticeably faster than that of st. However, unless there are sufficiently many periodic patterns so that the query time of seq is noticably faster than the other solutions, the radix solution is considered the best due to avoiding the extra cost of storing the sequences information (see Table 4.7) and arrays used for processing the periodic sequences. More processing time for sequences is also needed for seq which is also a drawback if a pattern set does not have many periodic patterns. The space usages were as expected too. This is seen, for example, in the solutions for the space usages of radix taking 8 more bytes per symbol (for the 2-d array of ranks of each suffix array value in all levels) than that of the corresponding st solutions. The seq solution takes a couple of bytes per symbol more than radix as it also stores sequence occurrence information on certain locus nodes for periodic processing.



(a) cov

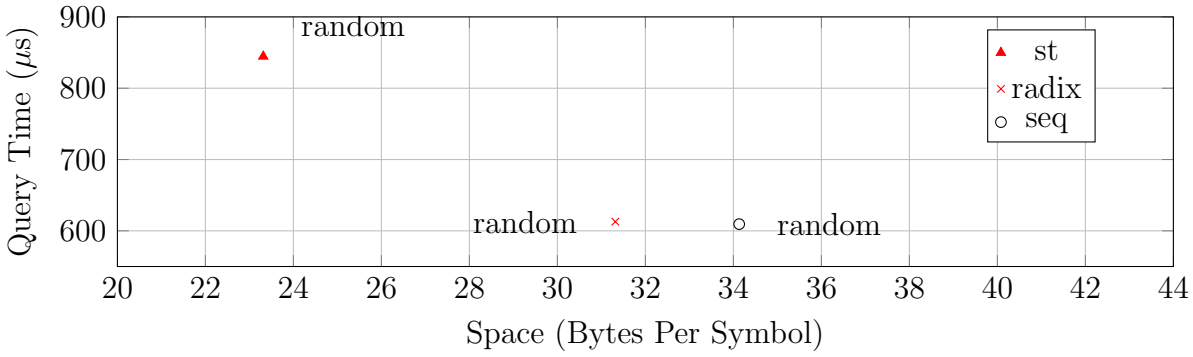


(b) eco

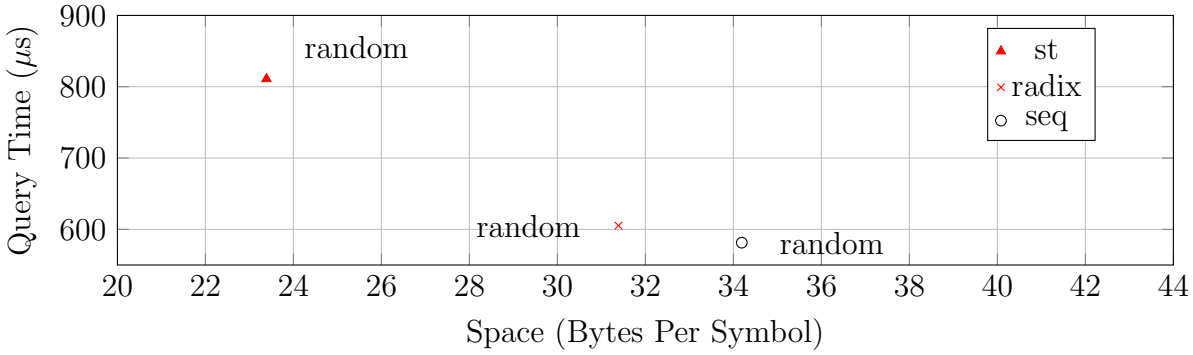


(c) sal

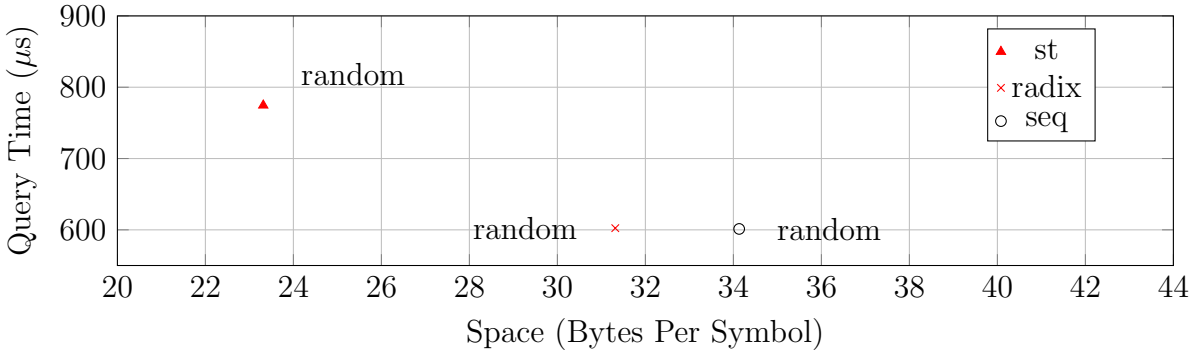
Figure 4.1: Space and time performances on longprd



(a) cov-st, radix and seq on random

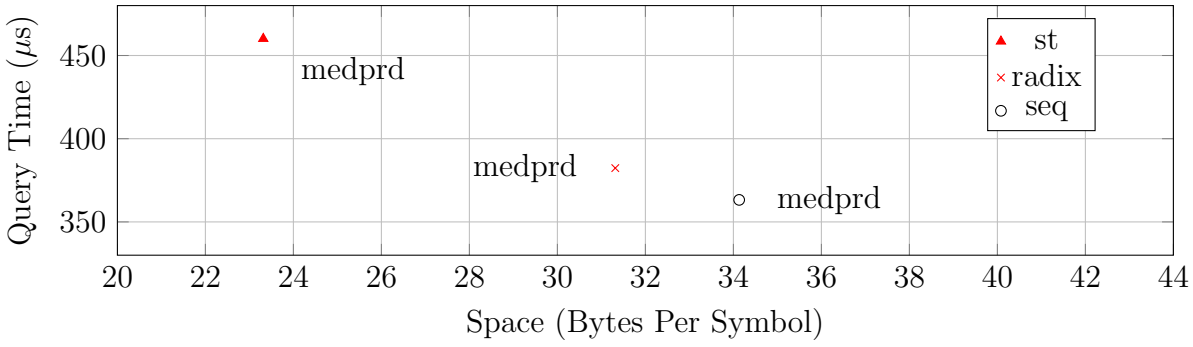


(b) eco

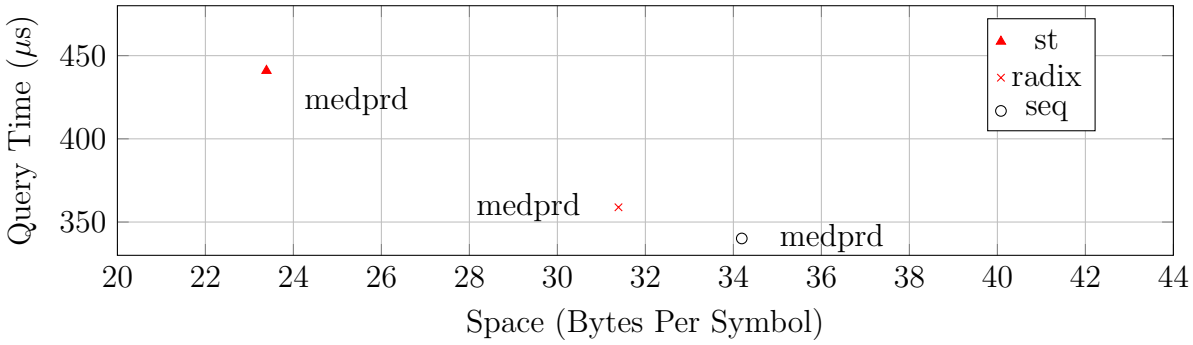


(c) sal

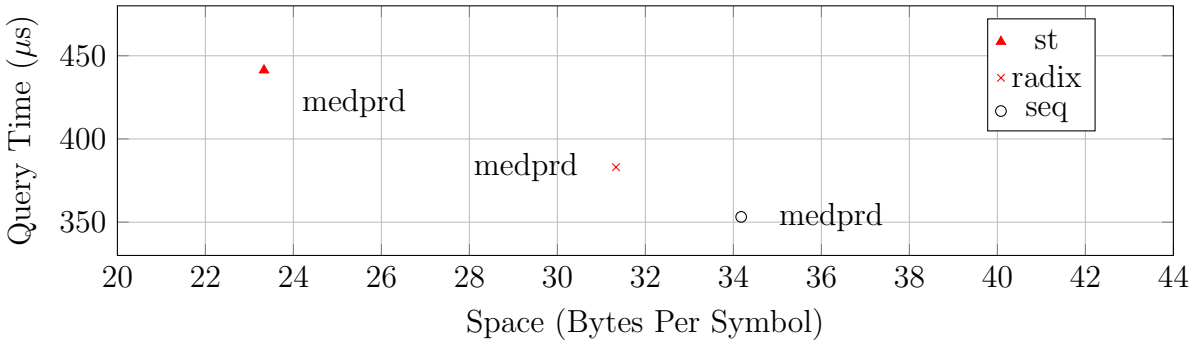
Figure 4.2: Space and time performances on random



(a) cov

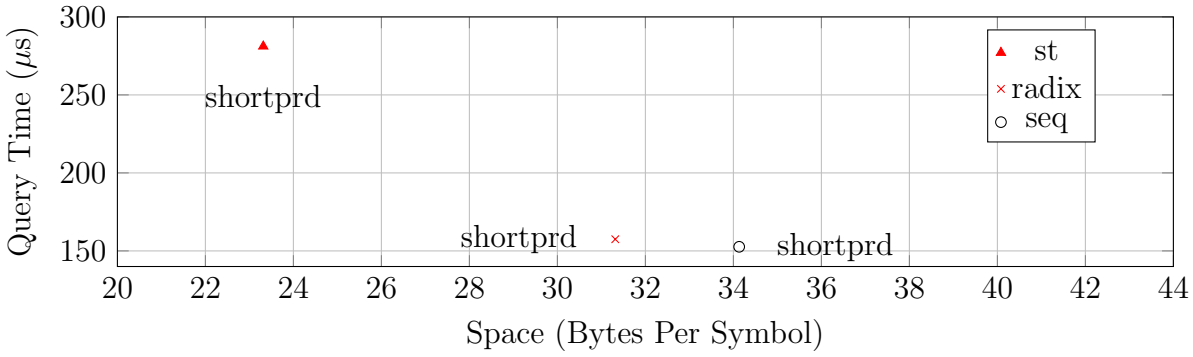


(b) eco

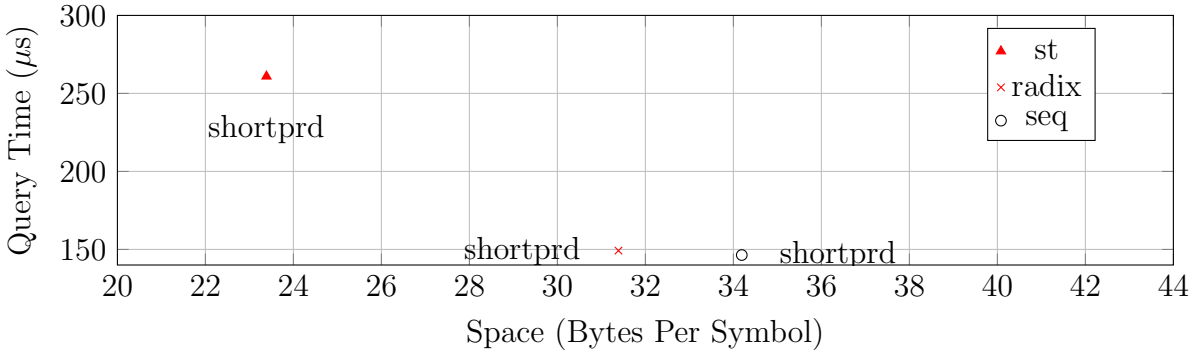


(c) sal

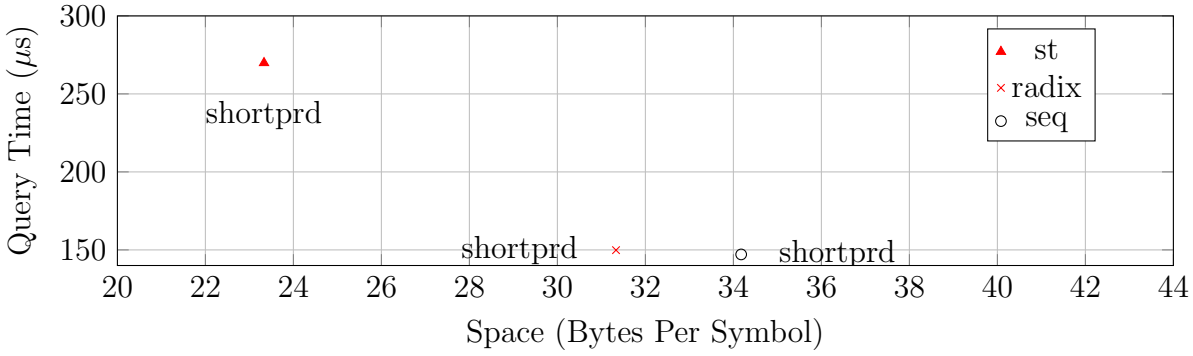
Figure 4.3: Space and time performances on medprd



(a) cov



(b) eco



(c) sal

Figure 4.4: Space and time performances on shortprd

Chapter 5

Conclusions

We set out the purpose of implementing a time and space optimal solution for non-overlapping indexing by Cohen and Porat [14]. The advanced solutions do indeed show a considerable improvement over the plain suffix tree solution st. Evaluating the advanced solutions against each other, it is seen that, for the datasets we are able to obtain, the extra space cost and huge increase of preprocessing time of seq when compared against radix renders seq unattractive since the query time of seq is not much better. Hence, it is expected that radix is the most preferable solution in most experimental cases, offering fast query time at reasonable space and preprocessing time usages.

One future work includes finding large practical text with a higher *occ* to *nocc* ratio to see if seq can outperform radix greatly over them. Having implementations or experiments done on the succinct solution of Ganguly et al. [21] is also considered as related future work. Finally, it is interesting to conduct experimental studies on related problems such as range non-overlapping indexing and string statistics.

Bibliography

[1] (2024). Experimental data. https://github.com/lzhouGit2023/ThesisData. Accessed: 2026-07-19.

[2] (2024). National Center For Biotechnology Information. NCBI. Available from https://www.ncbi.nlm.nih.gov/nuccore.

[3] Abeliuk, A., Cánovas, R., and Navarro, G. (2013). Practical compressed suffix trees. *Algorithms*, 6(2):319–351.

[4] Alexandrescu, A. (2017). Fast Deterministic Selection. In Iliopoulos, C. S., Pissis, S. P., Puglisi, S. J., and Raman, R., editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Amirzhanov, A., Makhmutova, A., and Turan, C. (2025). Plagiarism types and detection methods: a systematic survey of algorithms in text analysis. *Frontiers in Computer Science*, 7.

[6] Arimura, H., Kasai, T., Lee, G., Park, K., and Setsuo, A. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching*, 2089:181–192.

[7] Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., and Tsuruta, K. (2017). The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514.

[8] Barroso, R. A. C. (2010). *Compressed Data Structures for Suffix Trees*. PhD thesis.

[9] Beller, T., Gog, S., Moffat, A., and Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. pages 326–337.

[10] Bentley, J. L. and Friedman, J. H. (1979). Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409.

[11] Biswas, S., Ganguly, A., Shah, R., and Thankachan, S. V. (2018). Ranked document retrieval for multiple patterns. volume 746, pages 98–111.

[12] Brodal, G. S., Lyngsø, R. B., Östlin, A., and Pedersen, C. N. S. (2002). Solving the string statistics problem in time o(nlogn). *International Colloquium on Automata, Languages, and Programming*, 9:728–739.

[13] Chairungsee, S. (2018). Efficient approaches to compute longest previous non-overlapping factor array. *Fundamenta Informaticae*, 163(3):291–304.

[14] Cohen, H. and Porat, E. (2009). Range non-overlapping indexing. In Dong, Y., Du, D.-Z., and Ibarra, O., editors, *Algorithms and Computation*, pages 1044–1053, Berlin, Heidelberg. Springer Berlin Heidelberg.

[15] Crochemore, M., Ilie, L., Iliopoulos, C. S., Kubica, M., Rytter, W., and Waleń, T. (2013). Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26.

[16] Crochemore, M., Ilie, L., and Smyth, W. (2008). A simple algorithm for computing the lempel ziv factorization. *Proceedings of the Data Compression Conference*, pages 482 – 488.

[17] Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, USA. IEEE Computer Society.

[18] Ferrada, H. and Navarro, G. (2019). Lempel–ziv compressed structures for document retrieval. *Inf. Comput.*, 265(C):1–25.

[19] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. *Proc FOCS 2000*, 2000:390–398.

[20] Ferragina, P. and Manzini, G. (2001). An experimental study of an opportunistic index. In Kosaraju, S. R., editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, pages 269–278. ACM/SIAM.

[21] Ganguly, A., Shah, R., and Thankachan, S. V. (2020). Succinct non-overlapping indexing. *Algorithmica*, 82(1):107–117.

[22] Gao, Y., He, M., and Nekrich, Y. (2020). Fast Preprocessing for Optimal Orthogonal Range Reporting and Range Successor with Applications to Text Indexing. In Grandoni, F., Herman, G., and Sanders, P., editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54:1–54:18, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[23] Gusfield, D. (1997). Algorithms on strings, trees, and sequences - computer science and computational biology. New York, NY, USA. Cambridge University Press.

[24] Gusfield, D. and Stoye, J. (2002). Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1):843–856.

[25] Hon, W.-K., Shah, R., Thankachan, S. V., and Vitter, J. S. (2014). Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2).

[26] Huo, H., Chen, L., Zhao, H., Vitter, J. S., Nekrich, Y., and Yu, Q. (2015). A data-aware fm-index. ALENEX '15, page 10–23, USA. Society for Industrial and Applied Mathematics.

[27] Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. pages 943–955.

[28] Keller, O., Kopelowitz, T., and Lewenstein, M. (2007). Range non-overlapping indexing and successive list indexing. 4619:625–636.

[29] Knuth, D. E., Morris, Jr., J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350.

[30] Kolpakov, R. and Kucherov, G. (1999). Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 596, USA. IEEE Computer Society.

[31] Lothaire, M. (1997). Combinatorics on words: Bibliography.

[32] McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272.

[33] Pavesi, A., Vianelli, A., Chirico, N., Bao, Y., Blinkova, O., Belshaw, R., Firth, A., and Karlin, D. (2018). Overlapping genes and the proteins they encode differ significantly in their sequence composition from non-overlapping genes. *PLOS ONE*, 13:e0202513.

[34] Stoye, J. (2008). Suffix tree construction. In Kao, M.-Y., editor, *Encyclopedia of Algorithms*, pages 1–6, Boston, MA. Springer US.

[35] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

[36] Välimäki, N., Mäkinen, V., Gerlach, W., and Dixit, K. (2009). Engineering a compressed suffix tree implementation. *ACM J. Exp. Algorithmics*, 14.

[37] Weiner, P. (1973). Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society.