Graduate Theses, Dissertations, and Problem Reports

2011

Suffix Structures and Circular Pattern Problems

Jie Lin
West Virginia University

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

Recommended Citation

Lin, Jie, "Suffix Structures and Circular Pattern Problems" (2011). *Graduate Theses, Dissertations, and Problem Reports*. 3402.
<https://researchrepository.wvu.edu/etd/3402>

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Suffix Structures and Circular Pattern Problems

Jie Lin

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer and Information Sciences

Dr. Donald Adjero, Ph.D., Chair

Dr. Elaine M Eschen , Ph.D.

Dr. Arun Ross, Ph.D.

Dr. James Harner, Ph.D.

Dr. Cun-Quan Zhang, Ph.D

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia, 2011

Keywords: Suffix Array, Suffix Tree, Pattern Matching, Text Mining, Probabilistic Suffix Trees, Probabilistic Suffix Arrays, Markov Models, Space Efficiency, Circular Patterns, Multidomain Proteins, Circular Pattern Discovery

The suffix tree is a data structure used to represent all the suffixes in a string. However, a major problem with the suffix tree is its practical space requirement. In this dissertation, we propose an efficient data structure – the *virtual suffix tree* (VST) – which requires less space than other recently proposed data structures for suffix trees and suffix arrays. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form, where n is the length of the sequence.

Markov models are very popular for modeling complex sequences. In this dissertation, we present the *probabilistic suffix array* (PSA), a space-efficient alternative to the *probabilistic suffix tree* (PST) used to represent Markov models. The PSA provides all the capabilities of the PST, such as learning and prediction, and maintains the same linear time construction (linearity with respect to sequence length). The PSA, however, has a significantly smaller memory requirement than the PST, for both the construction stage, and at the time of usage.

Using the proposed suffix data structures, we study the circular pattern matching (CPM) problem. We provide a linear time, linear space algorithm to solve the exact circular pattern matching problem. We then present four algorithms to address the approximate circular pattern matching (ACPM) problem. Our bidirectional ACPM algorithm provides the best time complexity when compared with other algorithms proposed in the literature. Further, we define the circular pattern discovery (CPD) problem and present algorithms to solve this problem. Using the proposed circular pattern matching algorithms, we perform experiments on computational analysis and function prediction for multidomain proteins.

Acknowledgement

I would like to thank my advisor, Dr. Don Adjero, for his guidance, advice, and continued encouragement. It has been a pleasure to work under his supervision. Without him, this dissertation could not have come about.

I would also like to thank my other committee members: Dr. Elaine Eschen, Dr. Arun Ross, Dr. James Harner, and Dr. Cun-Quan Zhang for their help during my studies.

And finally, I thank my family members for their constant support, encouragement, and help.

The work reported in this thesis was partly supported by a DOE CAREER award (No: DE-FG02-02ER25541), an NSF ITR award (No: 0312484), and a WV-EPSCoR RCG grant.

Contents

1	Introduction	1
1.1	Overview	1
1.2	The Problem	2
1.2.1	Suffix Tree	2
1.2.2	Markov Models and Probabilistic Suffix Tree	3
1.2.3	Circular Pattern Matching and Circular Pattern Discovery	5
1.3	Contribution	6
1.4	Organization	7
2	Related Work	9
2.1	Suffix Tree and Suffix Array	10
2.1.1	Basic Notations and Definitions	10
2.1.2	Suffix Tree	10
2.1.3	Suffix Links	11

2.1.4	Implementation and Problems with the Suffix Tree	12
2.1.5	Suffix Array	12
2.2	Space-Efficient Suffix Trees	12
2.2.1	ESA	13
2.2.2	LST	14
2.3	Markov Models and Probabilistic Suffix Tree	15
2.3.1	Variable Length Markov Models	15
2.3.2	Probabilistic Suffix Tree	16
2.3.3	Computing TF and DF via Suffix Arrays	16
2.4	Circular Pattern Matching	21
2.4.1	String Pattern Matching	21
2.4.2	Circular Pattern Matching Problems	23
2.4.3	Exact Circular Pattern Matching (ECPM)	25
2.4.4	Approximate Circular Pattern Matching (ACPM)	26
2.4.5	ACPM Problem in Protein Sequences	29
2.5	Pattern Discovery Problem	30
3	The Virtual Suffix Tree	31
3.1	Introduction	31
3.2	Basic Data Structure	33
3.2.1	Example VST	34

3.2.2	Properties of the Virtual Suffix Tree	34
3.2.3	Pattern Matching on VST	36
3.3	Improved Virtual Suffix Tree	38
3.3.1	Adjusting Edge Lengths	38
3.3.2	Construction Algorithm	40
3.3.3	Further Space Reduction	41
3.3.4	Complexity Analysis	41
3.4	Computing Suffix Links	43
3.5	From SA to VST	45
3.6	Summary	50
4	The Probabilistic Suffix Array	59
4.1	Introduction	59
4.2	Probabilistic Suffix Tree	60
4.3	Proposed Data Structure	61
4.3.1	Internal Node Attributes	62
4.3.2	Measurement Attributes	63
4.3.3	Example PSA	64
4.3.4	Interval Array and Document Frequency in Linear Time	65
4.4	Constructing the PSA	66
4.4.1	Building the Interval Tree	67

4.4.2	Building the Suffix Link	68
4.4.3	Sorting the PSA Structure	68
4.4.4	Computing Conditional Probabilities Using the PSA	70
4.4.5	Prediction with VLMM via the PSA	71
4.5	Space Analysis	74
4.5.1	Storage Space	74
4.5.2	Construction Space	75
4.6	Experiments	77
4.6.1	Predicting Protein Families	77
4.6.2	Space Consideration	79
4.6.3	Computational Time Requirement	80
4.6.4	PSA in Phylogenetic Tree Construction	81
4.7	Summary	82
5	Circular Pattern Matching	87
5.1	Introduction	87
5.2	Exact Circular Pattern Matching Problem	89
5.2.1	Linear Time ECPM Algorithm	89
5.2.2	Comparison of ECPM algorithms	92
5.3	Approximate Circular Pattern Matching Problem	93
5.3.1	Greedy ACPM Algorithm	93

5.3.2	ACPM with LIS	94
5.3.3	ACPM with q-grams and Suffix Array	96
5.3.4	Improved Algorithm: ACPM with Bidirectional Edit Distance	98
5.3.5	Comparison with Other ACPM Algorithms	104
5.4	Experiments	105
5.4.1	Data Set	106
5.4.2	CPM Experimental Design	107
5.4.3	Multidomain Protein Networks using Circular Patterns	110
5.5	Summary	115
6	Circular Pattern Discovery	133
6.1	Introduction	133
6.2	The Circular Pattern Discovery Problem	134
6.3	The ECPD Algorithm	136
6.4	The ACPD Algorithm	137
6.4.1	ACPD using Maes' Algorithm	137
6.4.2	Proposed ACPD Algorithm	139
6.4.3	Comparison	140
6.5	Experiments	140
6.6	Summary	146

7	Conclusion and Future Work	147
7.1	Conclusion	147
7.2	Future Work	149
7.2.1	Circular Pattern Discovery	149
7.2.2	Network Analysis for Circular Multidomain Proteins	149
7.2.3	From PSA to PFA	150
7.2.4	Approximate Pattern Matching Using PSA	150
7.2.5	Prediction with PSA using Inexact Matching	150
7.3	Publications from the Dissertation	152

List of Tables

2.1	Interval Array	18
3.1	VST node attributes for the example sequence $T = \text{missississipp}\$$ used in Figure 3.1.	35
3.2	Node attributes in the improved VST for the example sequence, $T = \text{missississipp}\$$	40
3.3	Branching factor and maximum space requirement for various sample files.	54
3.4	Storage requirement for the VST, including suffix links	55
3.5	Detailed attributes for nodes in the TA data structure using the sample sequence, $T = \text{missississipp}\$$	55
3.6	Node mapping table from TA to VST	56
4.1	Attributes of PSA	63
4.2	Example PSA internal nodes, using the PSA of the sequence $T = \text{accactact}\$$	65
4.3	Example PSA leaf nodes, using the PSA of the sequence $T = \text{accactact}\$$	65

4.4	Performance of the PSA in modeling and prediction of protein families. Families correspond to the first 51 protein families with 12 or more members in the Pfam database, ordered alphabetically based on their abbreviated names in Pfam. For comparison, we have included the results obtained using the PST [17] on the same data set. (TP stands for true positive, while MD stands for missed detection). **The family apple was not in the dataset used in [17].	84
4.5	Summary performance in protein family classification using the PSA and PST .	85
4.6	Summary data on the first 51 families in Pfam, as described in Table 4.4.	85
4.7	Construction memory needed for the PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4.	85
4.8	Construction time comparison for PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4. Recorded time is time needed per family (in seconds). Speedup is computed as the ratio with respect to PSA time.	85
4.9	Prediction time comparison between PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4. Recorded time (in seconds) is prediction time per family – i.e. total time needed to predict all members in the family against all the other families.	86
5.1	Comparison of ECPM algorithms	93
5.2	Comparison with other proposed ACPM Algorithms	105
5.3	Top 15 highest degree proteins with GO function	109
5.4	The predicted protein functions using union for In-edge and Out-edge	111
5.5	The predicted protein functions using intersection for In-edge and Out-edge . .	112
5.6	Performance in Protein Function Prediction using the Top-500 Proteins	112

5.7	Network statistics for multidomain protein networks	114
5.8	Top 25 proteins with the highest node degree differences between protein networks using the circular and non-circular patterns.	116
5.9	The longest path in the Protein network	117
5.10	The longest path in the Family network	118
6.1	The number of distinct patterns with pattern length	144
6.2	Sample discovered circular patterns with length five.	144
6.3	Sample discovered circular patterns with length thirteen.	145

List of Figures

1.1	Summary of work in this dissertation.	8
2.1	Algorithm for computing document frequency [131]	20
2.2	Edit Graph of T and PP [87].	27
2.3	Maes' Algorithm	28
3.1	Suffix tree and virtual suffix tree for the string $T = \text{missississipp}\$$	51
3.2	Example VST (solid nodes) showing left SA index (ISA) and right SA index (rSA) for sample nodes.	51
3.3	Edge-length adjustment procedure.	52
3.4	Improved VST for the string $T = \text{missississipp}\$$	52
3.5	Suffix links on the VST for the sample string $T = \text{missississipp}\$$	55
3.6	Constructing VST from the suffix array.	56
4.1	State diagram and transition matrix for a first order Markov model for an example sequence $T = \text{accactact}\$$	61
4.2	Example suffix tree and probabilistic suffix tree for the string $T = \text{accactact}\$$	62

4.3	Top- k classification rate for sample protein families using the PSA.	79
4.4	Memory consumption factor (MC Factor) needed to construct the PSA and PST data structures for the first 51 protein families in Pfam.	80
4.5	Phylogenetic tree for 20 species constructed using the predicted probabilities obtained using the PSA.	82
5.1	Suffix tree for the string $T = \text{missississippi}$ with some suffix links. . .	118
5.2	The number of hypotheses with q -gram	121
5.3	Dynamic Programing in q -gram matching	122
5.4	Three cases in computing the circular edit distance in the ACPM algorithm using the bidirectional edit distance . The numbered double-header show the symbol positions involved in each case.	123
5.5	The time cost of the CPM algorithms	124
5.6	Degree distributions in the network of multidomain proteins constructed based on the circular patterns they contain.	125
5.7	Number of directly connected pairs in Top-K highest degree proteins	126
5.8	The Protein network (using both CPs and non-CPs)	127
5.9	The Protein network using only non-circular patterns	128
5.10	The Protein network using only circular patterns	129
5.11	The Family network (using both CPs and non-CPs)	130
5.12	The Family network using only non-circular patterns	131
5.13	The Family network using only circular patterns	132

6.1	Variation of the number of distinct patterns (including non-circular and circular patterns) with pattern length.	143
6.2	Variation of number of circular patterns with pattern length.	143
6.3	Variation of maximum number of occurrences with pattern length.	145

Chapter 1

Introduction

1.1 Overview

The suffix tree is an important data structure used to represent sequences (for example, text, DNA sequence, video, etc.). However, its space requirement is huge for most practical applications. Most methods for suffix tree (ST) and suffix array (SA) have focused on the theoretical time and space complexity. Markov models are popular for modeling complex sequences whose sources are unknown, or whose underlying statistical characteristics are not well understood. A major problem, however, is its space complexity, which grows exponentially with the order of the Markov model. The probabilistic suffix tree (PST) was proposed to address with the space problem of the Markov model. This reduced the space complexity theoretically. However, in practice, the space requirement for the PST is still relatively large, and often impractical for most real-life problems. Circular permutations and circular pattern matching are interesting problems in computer science and biology. There are several algorithms and methods to solve this problem. But in practice, these algorithms and methods have huge time and space costs.

In the first part of this dissertation, we develop efficient suffix data structures for analysis of huge sequences. In the second part, we use these data structures to study the problem of circular permutations and their applications in computational biology. Based on our circular permutation work, we define and study the circular pattern discovery problem.

First we propose a space efficient data structure called the *virtual suffix tree* (VST). The VST supports the same functions as the suffix tree, but with much less space practical requirement. The average space requirement is significantly smaller than other data structures for suffix tree and suffix array. Secondly, we propose an efficient data structure, the *probabilistic suffix array* (PSA) to represent the Markov model. PSA takes a much smaller space than probabilistic suffix tree which is implemented on a regular suffix tree. We show the experiment in the biology applications. Lastly, using suffix trees and suffix arrays we propose algorithms to solve the circular pattern matching problem and use these to study circular permutations and function prediction for multidomain proteins. We also introduce the circular pattern discovery problem and present algorithms to solve the problem.

1.2 The Problem

1.2.1 Suffix Tree

The suffix tree is an important data structure used to represent the set of all suffixes of a string. The suffix tree is efficient in both time and space, and has been used in a variety of applications, such as pattern matching, sequence alignment, identification of repetitions in genome-scale biological sequences, and in data compression. Various algorithms have been developed for efficient construction of suffix trees [38, 89, 122, 128]. However, one major problem with the suffix tree is its practical space requirement. The suffix array is a related data structure, which was originally introduced in [84] as a space-efficient alternative to the suffix tree. The suffix array simply provides a listing of all the suffixes of a given string in lexicographic order. The suffix array can be used in most (though, not all) situations where a suffix tree is used.

Although the theoretical space complexity is linear for both data structures, typically, for a given string T of length n , the suffix array requires about three to five times less space than the suffix tree. The construction time for both algorithms is also $O(n)$ on average. For suffix arrays, construction algorithms that run in $O(n \log n)$ worst case are relatively easy to develop, but $O(n)$ worst case algorithms are much harder to come by. Recent suffix sorting algorithms with worst-case linear time complexity have been reported in [57, 63, 65]. Gusfield [46] provides a comprehensive treatment of suffix trees and its applications. Puglisi et al. [104] provide a

recent survey on suffix arrays. An extensive discussion on the connection between the Burrows-Wheeler Transform [23] and suffix trees and suffix arrays is provided in [3].

For small alphabet sizes, the suffix tree and the suffix array have about the same complexity in pattern matching. For pattern matching, the suffix array requires time in $O(m \log n)$ to locate one occurrence of a pattern P of length m in T . However, with additional data structures, such as the lcp array, this time can be reduced to $O(m + \log n)$. With the suffix tree, the same search can be performed in $O(m)$ time. The problem, however, is for sequences with large alphabets where $|\Sigma| \rightarrow n$. Here, $|\Sigma|$, the alphabet size is no longer negligible. Using the array representation of nodes in the suffix tree will require $O(n|\Sigma|)$ space for the suffix tree, and $O(m)$ time for pattern matching. For linear space, the linked list or binary search tree can be used, but the search time becomes $O(m|\Sigma|)$ or $O(m \log |\Sigma|)$ respectively.

The Challenge. A key problem then is to develop space-efficient data structures that can support pattern matching using the same time complexity as suffix trees, but at a practical space requirement that approaches that of the suffix array. Such a data structure should also support the complete functionality of the suffix tree, such as support for suffix links, as may be required in certain applications. Two recent data structures that have attempted to address this problem are the ESA – *enhanced suffix array* [2], and the LST – *linearized suffix tree* [61, 62]. Both methods are based on the notion of lcp -intervals [60], constructed using the suffix array and the lcp array.

1.2.2 Markov Models and Probabilistic Suffix Tree

Markov models are very popular for modeling complex sequences whose sources are unknown, or whose underlying statistical characteristics are not well understood. This is especially the case when the sequences exhibit some memory. For a short term memory of length, say L , this means that the conditional distribution of the next symbol given the last L symbols does not change significantly if we condition on L or more previous symbols. Thus, such sequences are often modeled using Markov models of order L , or using the Hidden Markov Models (HMM) [37]. The models provide efficient mechanisms to compute the required conditional probabilities, and also for generating sequences from the models. The problem is that the size of Markov models increases exponentially with increasing memory length L . Thus, they are

practical only for low order models with short memory lengths. This leads to the second challenge: such low-order Markov models often provide a poor approximation of the true sequence being modeled. It is known that learning with Hidden Markov Models is computationally very challenging. Hardness results on the learnability of HMM are discussed in [1], while similar results on inferencing using HMM are reported in [42].

Probabilistic suffix models such as probabilistic suffix trees (PST) have been proposed by Ron et al. [108] to address some of the key problems with Markov models. They showed the equivalence between PSTs and a subclass of probabilistic finite automata (PFAs) called probabilistic suffix automata (PSF/PSA): for a given PST, there is an algorithm to construct a PSF/PSA whose size is the same as that of the PST within a constant factor. Further, the distribution generated by a PST is guaranteed to be within a small distance from that generated using the PSF/PSA, as measured by the Kullback-Leibler divergence.

Probabilistic suffix trees and probabilistic suffix automata are related to context-based models which are extensively used in sequence prediction and data compression. Typical examples of such context models include context tree weighting (CTW) [129], prediction by partial matching (PPM) [28], and the Lempel-Ziv decomposition [133, 134]. See also [106, 107]. The use of these context models as a surrogate for variable length Markov models with applications in sequence prediction are reviewed in [16]. Other applications have been found in modeling DNA sequences [109], protein sequence classification [17, 71], and in modeling API call sequences for malicious codes in Windows XP [88].

The Challenge. The probabilistic suffix models however require $O(Ln^2)$ time and space to construct. The algorithm to construct the PFA/PSA from the PST also runs in $O(Ln^2)$ time. Later Apostolico and Bejerano [9] showed how the PST can be constructed in $O(n)$ time, independent of the order L , using traditional suffix links used in constructing suffix trees, and the notion of reverse suffix links. They did not consider the problem of constructing the PSF/PSA from the PST. Their use of the suffix tree also implies that the space requirements for constructing the PST will be very high, although it is still linear in terms of the sequence length.

1.2.3 Circular Pattern Matching and Circular Pattern Discovery

Computing similarity (or dissimilarity) between two strings is an important problem in general sequence analysis [44, 44, 46, 51, 81, 115], pattern recognition [113, 120] and biology [40, 47, 55]. The *circular edit distance* is an extension of traditional edit distance which seeks to determine similarity between strings in a circular shift. A circular shift is a mapping $f : \Sigma^* \rightarrow \Sigma^*$, $f^t(c_1 \dots c_r) = c_{t+1} \dots c_r c_1 \dots c_{t-1} c_t$, where $0 \leq t \leq r - 1$ and r is the length of string $c_1 c_2 \dots c_r$. The *circular edit distance* between two strings s_1 and s_2 is defined as $ED_c(s_1, s_2) = \min\{ED[f^i(s_1), f^j(s_2)] \mid 0 \leq i \leq |s_1| - 1, 0 \leq j \leq |s_2| - 1\}$, where $|s_1|$ is the length of string s_1 and $|s_2|$ is the length of string s_2 , $ED[A, B]$ is the standard edit distance between A and B . Thus, the dissimilarity between two strings in a circular shift is a function of the circular distance between them. Computational methods have also been proposed to study circular patterns in biological problems. [123, 124, 126, 127]

In biology, circular proteins and circular permutations in proteins are becoming of increasing interest, especially given their role in the structure, function, folding, and stability of proteins [40, 47, 55]. In a circular (or cyclic) protein, the traditional N- and C-termini are joined, resulting in a protein sequence with no termini [123]. The cyclotides is a typical example of a naturally-occurring family of cyclic proteins in the Plant Kingdom. Cyclotides are known to play a major role and provide important functions in terms of plant defense against insects and other pathogens [35]. Their cyclic structure is known to be an important factor in their unusual stability [35]. Other common examples of cyclic proteins are the bacteriocins, small antimicrobial peptides with 30-70 residues produced by bacteria [33], cyclosporins found in fungi [66], and the primate rhesus θ -defensin -1 [119] with antibacterial properties for the immune system of macaques monkeys.

The Challenge. There are several algorithms for calculating the circular distance between text T and circular pattern P , but they did not consider the problem of finding circular pattern P and its circular shifts inside substrings of text T . In our work, we define variants of the CPM problem and propose algorithms to solve them.

Pattern discovery is a fundamental analysis method used to identify possibly hidden relations within or between the sequences. Pattern discovery problems are well studied in many

applications. In biology, motif discovery is often performed as a kind of pattern discovery application. To our knowledge, there is no existing work that explicitly studied the problem of pattern discovery involving circular patterns. In this work, we introduce the Circular Pattern Discovery problem and propose algorithms for its solution.

1.3 Contribution

We introduce the VST (virtual suffix tree), an efficient data structure for suffix trees and suffix arrays. Starting from the suffix array, we construct the suffix tree, from which we derive the virtual suffix tree. Later, we remove the intermediate step of suffix tree construction, and build the VST directly from the suffix array. The VST provides the same functionality as the suffix tree, including suffix links, but at a much smaller space requirement. It has the same linear time construction even for large alphabets, Σ , requires $O(n)$ space to store (n is the string length), and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. Given the VST, we show an algorithm that computes all the suffix links in linear time, independent of Σ . The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the enhanced suffix array [2], and the linearized suffix tree [62]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form.

We present the probabilistic suffix array (PSA), a data structure for representing information in variable length Markov chains. The PSA essentially encodes information in a Markov model by providing a space-efficient representation of the probabilistic suffix tree (PST). Our PSA provides the same functionality as the PST, but at a significantly reduced space requirement. Given a sequence of length n , construction and learning in the PSA is done in $O(n)$ time and space, independent of the Markov order. Prediction using the PSA is performed in $O(m \log \frac{n}{|\Sigma|})$ time, where m is the pattern length, and Σ is the symbol alphabet. The specific memory requirement is $33n$ bytes in the worst case, and $26n$ bytes on average, including space for the suffix array and the input sequence. This can be compared with the $41n$ bytes needed using the PST.

We propose an exact circular pattern matching (ECPM) algorithm that runs in linear time

and linear space. We also propose algorithms to solve the approximate circular pattern matching (ACPM) problem. In our work, we solved a harder version of the ACPM problem when compared to other previous work on ACPM [44, 81, 123, 124, 126, 127]. We present an experiment on finding circular relations in multi-domain proteins. Our experiments show that the methods based on circular permutations can produce very good results in predicting protein functions.

We propose two algorithms for the Circular Pattern Discovery (CPD) problem. The first algorithm uses suffix trees and suffix links to solve the exact circular pattern discovery problem in $O(m_2^2 N)$ time. The second algorithm uses suffix arrays to solve the more challenging approximate circular pattern discovery (ACPD) problem in $O(km_2^2 N^2)$ worst case, and $O(km_2^2 N)$ on average. By exploiting the nature of the ACPD problem, the complexity can be reduced to $O(m_2^2 N^2)$ worst case, and $O(m_2^2 N)$ on average.

Aspects of the work from this dissertation are reported in the following papers: [74–79].

1.4 Organization

The dissertation is organized as follows. In Chapter 2, we introduce related work, including basic notations and definitions. Chapter 3 presents the Virtual Suffix Tree (VST) including its construction algorithm, searching algorithm and support for suffix links. We also analyze its time and space complexity and compare with related algorithms. Chapter 4 introduces the *Probabilistic Suffix Array* (PSA) including detailed explanation and implementation. The practical space requirement of the data structure is also examined. In this chapter, we present the experiments of PSA using protein sequences and miDNA sequences and protein sequences. Chapter 5 discusses the circular pattern matching (CPM) problem. Algorithms are then proposed for the exact and inexact variants of the problem using suffix data structures. We implement the algorithms and show examples of using these algorithms to search for circular patterns in multidomain proteins. In Chapter 6, we define the circular pattern discovery (CPD) problem and present algorithms to solve the CPD problem. Chapter 7 draws some conclusions and also describes possible directions for future work. Figure 1.1 provides a summary of the work reported in this dissertation.

Figure 1.1. Summary of work in this dissertation.

Chapter 2

Related Work

In this chapter, first we introduce *the suffix data structures*, namely suffix trees and suffix arrays, which form the basis for the majority of the work proposed in this dissertation. *The suffix data structures* are important in computer science and bioinformatics. In Section 2, we introduce the *enhanced suffix array* (ESA), the *Linearized Suffix Tree* (LST), and other space-efficient suffix trees. These two structures use the suffix array and LCP array to simulate the suffix tree. They are related to our new structure, the virtual suffix tree (VST).

In Section 3 we describe *variable length Markov models* (VLMM) and *the probabilistic suffix tree* (PST) which is used to implement VLMM. They are related to a new data structure, the *probabilistic suffix array* (PSA) proposed in this work. We also present previous work on calculating the term frequency (TF) and document frequency (DF), which are used in our proposed PSA representation of VLMM.

In Section 4, we discuss the *circular pattern matching* (CPM) problem. We first review the general string pattern matching problem. We define two CPM problems and introduce previous work. We will provide our solutions to the different CPM problems and experimental results in Chapter 5. In Section 5, we discuss the related work on the pattern discovery problem.

2.1 Suffix Tree and Suffix Array

2.1.1 Basic Notations and Definitions

Let $T = T[1..n]$ be the input string of length n , over an alphabet Σ . Let $T = \alpha\beta\gamma$, for some strings α , β , and γ (α and γ could be empty). The string β is called a *substring* of T , α is called a *prefix* of T , while γ is called a *suffix* of T . The prefix α is called a proper prefix of T if $\alpha \neq T$. Similarly, the suffix γ is called a proper suffix of T if $\gamma \neq T$. We will also use $t_i = T[i]$ to denote the i -th symbol in T — both notations are used interchangeably. We use $T_i = T[i..n] = t_i t_{i+1} \dots t_n$ to denote the i -th suffix of T . For simplicity in constructing suffix trees, we ensure that no suffix of the string is a proper prefix of another suffix by appending a special symbol, $\$$ to T , such that $\$ \notin \Sigma$, and $\$ < \sigma, \forall \sigma \in \Sigma$. We let $P = P[1..m]$ to be the pattern string that needs to be found in T .

In our work, the size of alphabet $|\Sigma|$ is not fixed. It may be small, example $|\Sigma|=4$ for DNA sequences, or it may be large, example $|\Sigma| \approx 10^6$ for multidomain protein sequences.

2.1.2 Suffix Tree

Given a string T , its suffix tree (ST) is a rooted tree with n leaves, where the i -th leaf node corresponds to the i -th suffix T_i of T . Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the suffix tree represents a substring of T , and no two edges out of a node start with the same character. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. We use l_i to denote the i -th leaf node. Then, l_i corresponds to T_i , the i -th suffix of T . When the edges from each node are sorted alphabetically, then l_i will correspond to $T_{SA[i]}$, the i -th suffix of T in lexicographic order, where SA denotes the suffix array.

For edge (u, v) between nodes u and v in ST, the edge label (denoted $label(u, v)$) is a non-empty substring of T . The edge length is simply the length of the edge label. The edge label is usually represented compactly using two pointers to the beginning and end of its corresponding substring in T . For a given node u in the suffix tree, its *path label*, $L(u)$ is defined as the label of

the path from the root node to u . Since each edge represents a substring in T , $L(u)$ is essentially the string formed by the concatenation of the labels of the edges traversed in going from the root node to the given node, u . The *string depth* of node u , (also called its string length or path length) is simply $|L(u)|$, the number of characters in $L(u)$. The *node depth* (also called node level) of node u is the number of nodes encountered in following the path from the root to u . The root is assumed to be a node at depth 0.

Given the string $T = T[1..n]$, of length n , but with the end of string symbol appended to give a sequence $T\$$ with length $n + 1$, the suffix tree of the resulting string $T\$$ will have the following properties:

1. Exactly $n + 1$ leaf nodes.
2. At most n internal (or branching) nodes (the root node is considered an internal node).
3. Every *distinct* substring of T is encoded exactly once in the suffix tree. Each distinct substring is spelled out exactly once by traveling from the root node to some node u , such that $L(u)$ is the required substring. Note that the node u may be an implicit node, i.e. ending at a position between two (explicit) nodes.
4. No two edges out of a given node in the suffix tree start with the same symbol.
5. Every internal node has at least two outgoing edges. Properties (1), (2), (4), and (5) imply that a suffix tree will have at most $2n + 1$ total nodes, and at most $2n$ edges.

2.1.3 Suffix Links

Some suffix tree construction algorithms make use of *suffix links*. The notion of suffix links is based on a well-known fact about suffix trees [89, 128], namely, if there is an internal node u in ST such that its path label $L(u) = a\alpha$ for some single character $a \in \Sigma$, and a (possibly empty) string $\alpha \in \Sigma^*$, then there is a node v in ST such that $L(v) = \alpha$. A pointer from node u to node v is called a *suffix link*. If α is an empty string, then the pointer goes from u to the root node. Suffix links are important in certain applications, such as in computing matching statistics needed in approximate pattern matching, regular expression matching, or in certain types of traversal of the suffix tree.

2.1.4 Implementation and Problems with the Suffix Tree

A predominant factor in the space cost for suffix trees is the number of interior nodes in the tree, which depends on the tree topology. Thus, a major consideration is how the outgoing edges from a node in the suffix tree are represented. The three major representations used for outgoing edges are arrays, linked lists, and binary search trees. While the array is simple to implement, it could require a large memory for large alphabets. However, independent of the specific method adopted, a simple implementation of the suffix tree, for example, using Ukkonen's algorithm [122], can require as large as $33n$ bytes of storage with suffix links, or $25n$ bytes without suffix links [3].

2.1.5 Suffix Array

The suffix array (SA) is another data structure, closely related to the suffix tree. The suffix array simply provides a lexicographically ordered list of all the suffixes of a string. If $SA[i] = j$, it means that the i -th smallest suffix of T is T_j , the suffix starting at position j in T . A related structure, the LCP array contains the length of the longest common prefixes between adjacent positions in the suffix array. Combining the suffix array with the LCP information provides a powerful data structure for pattern matching. With this combination, decisions on the occurrence (or otherwise) of a pattern P of length m in the string T of length n can be made in $O(m + \log n)$ time. Given the new worst-case linear-time direct SA construction algorithms, and the small memory footprint of suffix arrays, it is becoming more attractive to construct the suffix tree from the suffix array. A linear-time algorithm for constructing ST from SA is presented in [3].

2.2 Space-Efficient Suffix Trees

The problem of practical space needed in using suffix trees have been recognized, and methods have been proposed to provide space-efficient data structures [8,45,82,83,94,110,111]. Andersson et al. [8] presented a level-compressed suffix tree in $O(n)$ bytes. Munro et al. [94] proposed some space efficient suffix structures in $O(n \log n)$ bits of space and $O(m|\Sigma|)$ searching

time. The structures include a suffix array and other auxiliary structures to represent the suffix tree. But these structures did not include the suffix link which is an important part of the suffix tree. Grossi et al. [45] proposed compressed suffix structures, an indexing structure in $O(n)$ bits with $O(m|\Sigma|)$ searching time. Compact Suffix Array [82] uses at most $9n$ bytes of space (or $13\frac{1}{8}n$ for including the LCP array), but the time of construction is $O(n \log n)$ and time of searching is $O(m \log \log n + (\log n)^2 \log \log n + n_{occ} \log n \log \log n)$, where n_{occ} is the number of occurrences.

While the suffix array reduces the problem of space, the suffix tree still provides a simpler way in certain analysis problems, such as in computing matching statistics [27]. Thus, methods have been proposed to improve the suffix array with extra information to provide the full functionality of the suffix tree. The enhanced suffix array [2] and the linearized suffix tree [61, 62] are two example data structures recently proposed for full-text indexing with the functionalities of both the suffix tree and suffix array.

2.2.1 ESA

The enhanced suffix array (ESA) [2] is composed of the suffix array, and extra data structures, namely, the `lcp` array and a child table that contains branching information between parent and child nodes in the suffix tree. The key idea used in the ESA is the concept of *lcp-intervals* (originally used in [60]). Given the suffix array, SA an interval $[i..j]$ in SA, $1 \leq i < j \leq n+1$ is called an *lcp-interval* with *lcp-value* l if the following conditions hold:

1. $\text{lcp}[i] < l$;
2. $\text{lcp}[k] \geq l \ \forall \ k \text{ s.t. } i+1 \leq k \leq j$;
3. $\text{lcp}[k] = l$, for some k , s.t. $i+1 \leq k \leq j$;
4. $\text{lcp}[j+1] < l$.

Thus, rather than the traditional suffix tree, the ESA constructs an *lcp-interval tree*. Nodes in a suffix tree are now replaced with *lcp-intervals*, such that the parent-child relationships

in a traditional suffix tree are now captured by equivalent parent-child relationships between lcp-intervals. The root node corresponds to the interval $[1..n]$ in the suffix array, essentially the entire suffix array. In the ESA, the basic structure used to represent the child-nodes from a given parent node is a linked list. Essentially, the child table is composed of three arrays, namely `up`, `down`, and `nextIndex`. The `up` and `down` arrays store information about the edges in the tree, while array `nextIndex` records information about the linked list used to represent the sibling relationship between nodes with the same parent. These three arrays would ordinarily require $3n$ elements to store. Interestingly, only n of these elements are required, and hence the child table requires only n integers to store. The ESA assumes that $|\Sigma|$ is small relative to n . Thus, for large alphabets, pattern matching could take longer on the ESA. For instance, with the binary tree representation of the nodes in the suffix tree, pattern matching will take $O(m \log |\Sigma|)$ time for a pattern of length m ; doing the same on the enhanced suffix array (which uses a linked list representation of the nodes) will require $O(m|\Sigma|)$ time, a significant difference for large alphabets.

2.2.2 LST

The linearized suffix tree (LST) [61, 62] is an improvement on the ESA. It uses the same `up` and `down` arrays as the ESA, but replaces the `nextIndex` array with two other arrays: `lchild` and `rchild`. Thus, the two arrays store information about siblings at a given node in the interval tree, such that the intervals can be represented by a complete binary tree. Specifically, let the interval $[i..j]$ denote the lcp-interval for a given node in the lcp-interval tree (equivalently a node in the traditional suffix tree). Then, the two new arrays are defined as follows [61]: `lchild[i]` records the first index of the left child node of the longest interval starting at i in the complete binary tree; `rchild[i]` records the corresponding value for the right child. Like the ESA, the nature of the four arrays in the new child table used by the LST makes it possible to represent the relevant information in the table using only n integers rather than the $4n$ integers that ordinarily would be required. However, unlike the ESA, the LST uses a complete binary tree as the basic structure to represent information about sibling nodes at a given node. This important difference makes it possible for the LST to support pattern matches in $O(m \log |\Sigma|)$ time, the same time bound for suffix trees.

2.3 Markov Models and Probabilistic Suffix Tree

2.3.1 Variable Length Markov Models

A Markov model is a sequence of stochastic events $\{X_n, n = 0, 1, 2, 3, \dots\}$ with state space S that satisfies the Markov property:

$$P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = P(X_{n+1} = j | X_n = i)$$

A Markov model (or Markov chain) of order L , where L is finite, is a sequence of events satisfying:

$$P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \dots, X_{n-L} = i_{n-L})$$

Thus, in an order- L Markov chain, the current state is dependent on the past L states. Fixed length Markov models (FLMM) represent a probabilistic finite state machine which can be used to model arbitrarily complex sequential data. Such models aim at learning the probability, $P(\sigma | C)$, the conditional probability distribution of a symbol σ , given its context C , where $\sigma \in \Sigma$, and $C \in \Sigma^L$, L is the order or memory length of the model, and is fixed. The FLMM of order L is represented as a $\Sigma^L \times \Sigma$ matrix. The space requirement is thus in $O(\Sigma^{L+1})$.

Variable length Markov Models (VLMM) differ from FLMM in an important way. Variable length Markov models attempt to learn the conditional distribution of a symbol whereby the context length or model order could be varying, depending on the data being modeled. Essentially, for VLMM, $C \in \bigcup_{i=1}^L \Sigma^i$. This property of varying memory length implies that with the VLMM, Markov dependencies of varying order in the training data – both large and small – could be captured with ease. This flexibility of the VLMM however comes at a huge cost in terms of space requirement. To represent a VLMM of order L , we have to store L matrices, one for each order, from 1 to L . The total space requirement will be in $O(\Sigma^2 + \Sigma^3 + \dots + \Sigma^{L+1})$, or $O(\Sigma^{L+2})$. The space is huge, even when L is small. Thus, the space requirement for Markov models is exponential in L , whether we consider fixed length or variable length models.

An important observation that could point to a potential reduction in the space requirement for Markov models is that for a given sequence of length n , there are $n(n+1)/2$ possible sub-

strings in the sequence. Thus, there are at most $n(n+1)/2$ states that can be represented in the Markov model for the sequence, for any given order. For the length- n sequence, the maximum order of a Markov model will be $n-1$. Thus, with knowledge of the sequence, we can have a limit on the possible number of states in its Markov model.

2.3.2 Probabilistic Suffix Tree

The probabilistic suffix tree (PST) is a probabilistic suffix model which is based on the traditional suffix tree. Like the suffix tree, the PST represents all the $n(n+1)/2$ substrings from the root to the leaf nodes. The PST models variable length Markov models, which means that the string depth is not fixed for every node. For an FLMM, its corresponding PST can be obtained from the PST of the VLMM by constraining each leaf node to be of the same string depth. The transition probability of a symbol on a given path is computed as the relative frequency of the symbol in the observed data, given the preceding substring on the path. The length of the substring used to determine such conditional probabilities is simply given by the memory length or order of the model.

Example PST and ST for a sample sequence $T = \text{accactact\$}$ are given in Chapter 4 (see Figure 4.2).

The original algorithm [109] used an $O(Ln^2)$ time complexity to construct and prune the PST from a suffix tree. The improved algorithm [88] used balanced red-black trees [31] to construct the PST in $O(Ln \log n)$ time complexity. Apostolico and Bejerano [9] presented an algorithm having an $O(n)$ time complexity using suffix links and reverse suffix links, independent of L .

2.3.3 Computing TF and DF via Suffix Arrays

In [9, 109], the conditional probabilities required for the PST were computed as relative counts, using the notion of *empirical probabilities*, based on symbol frequencies from the observed data. To compute the empirical probabilities, we use the notions of *term frequency* and *document frequency* as used in information retrieval. The term frequency, TF is simply the

number of times a given term (or substring in our case) occurred in a given text. Here, the text could contain many documents. Therefore, the document frequency for a given term is the number of times the term occurred in the document. Thus, the term frequency is easily obtained from the document frequency as a simple sum. When the text contains only one document, the TF and DF will be the same.

There are $n(n+1)/2$ substrings in a sequence of size n . Using a naïve algorithm, we will need to compute TF s for all the $n(n+1)/2$ terms. However, with the suffix tree for this sequence, we have n leaf nodes and at most n internal nodes. These $2n$ nodes represent the $n(n+1)/2$ substrings in the sequence. Therefore, some multiple substrings at different positions in the sequence will be represented by the same node. These substrings represented by the same node must have the same frequency count. Since the node labels are unique in the suffix tree, this means that the multiple substrings in the same node are essentially the same substring, that were repeated multiple times in the sequence. Thus, while there are $O(n^2)$ possible substrings in T , there are at most $O(n)$ unique substrings. Hence, we need to compute the TF for only the $2n$ unique substrings.

Yamamoto and Church [131] presented a data structure to represent nodes in the suffix tree using a suffix array. They called this structure the *interval array*. Using the interval array, their algorithm calculated all the required TF s in $O(n)$ time, but needed $O(n \log n)$ time to compute the DF s. Our data structure is based on the interval array, and we will show how we can improve the algorithm for computing DF to $O(n)$ time.

Interval Array Structure

From the suffix tree, we know we can cluster the potential $n(n+1)/2$ substrings into at most $2n$ "groups". In [131], the interval array was proposed as a data structure to represent these groups. Table 2.1 shows the interval array for sample sequence $T = \text{accactact}\$$. From the table, we can describe some important properties of the interval array as follows.

For all substrings, we have the following:

1. There are at most $2n$ groups in the document of size n . Substrings in the same group have

the same statistics (for example: TF and DF) and the same derivative measurements from these statistics.

2. An lcp-delimited interval $\langle i, j \rangle$ is constructed using the LCP array, where $\langle i, j \rangle$ is an interval on the suffix array. An lcp-delimited interval $\langle i, j \rangle$ must meet the condition:

$\max(LCP[i], LCP[j+1]) < Length_{group(i,j)} \leq \min(LCP[i+1], LCP[i+2], \dots, LCP[j])$, where, $Length_{group(i,j)}$ is the LCP of the suffixes that belong to the same group with the $\langle i, j \rangle$ interval.

The lcp-delimited intervals have the following properties:

1. Each lcp-delimited interval represents one unique group of substrings.
2. The maximum length of an lcp-delimited interval $\langle i, j \rangle$ is given by:
 $\min(LCP[i+1], LCP[i+2], \dots, LCP[j])$.
3. A non-trivial lcp-delimited interval is one with a start position that is less than the end position. That is, the lcp-delimited interval $\langle i, j \rangle$ is non-trivial if $i < j$. Otherwise, if $i \geq j$, the interval is said to be trivial. There are n trivial lcp-delimited intervals with $TF=1$. There are at most $n - 1$ non-trivial lcp-delimited intervals with $TF > 1$
4. The lcp-delimited intervals for a document can form a nested structure of intervals, but no two lcp-delimited intervals can overlap. Thus, the lcp-delimited intervals can be represented in a tree-like structure.
5. Let α and β be two substrings in the same lcp-delimited interval $\langle i, j \rangle$. Then, the following two conditions hold:

Table 2.1. Interval Array

Interval	LCP	Term Frequency
$\langle 2, 3 \rangle$	3	2
$\langle 1, 3 \rangle$	2	3
$\langle 6, 7 \rangle$	2	2
$\langle 4, 7 \rangle$	1	4
$\langle 8, 9 \rangle$	1	2

- $TF(\alpha) = TF(\beta) = j - i + 1$
- $DF(\alpha) = DF(\beta)$

Below we consider algorithms for computing TF and DF .

Determining the lcp-delimited intervals

Computing the term frequency (TF) is almost analogous to determining the lcp-delimited intervals. Given the lcp-delimited interval $\langle i, j \rangle$, the TF algorithm in [131] uses the relation $TF(\langle i, j \rangle) = j - i + 1$ to determine the term frequency for the interval. The time complexity of the algorithm is $O(n)$, using $O(n)$ space.

We observe that there are at most n neighboring LCP pairs (i.e. $LCP[i]$ and $LCP[i + 1]$) in a suffix array of size n . Thus, there are at most n increasing orders in such neighboring pairs. Similarly, there are at most n decreasing orders between the neighboring pairs. A decreasing order between neighboring pairs $LCP[i]$ and $LCP[i + 1]$ implies that there exists at least one lcp-delimited interval $\langle k, i \rangle$, where $k \leq i$. Using the foregoing and the fact that we have at most $2n$ lcp-delimited intervals in a document of size n , Yamamoto and Church [131] computed the term frequency. Given that the lcp-delimited intervals could be nested, a stack can be used to hold the information on the intervals.

Assuming we have m decreasing neighboring pairs in an LCP array of size n . Let the interval between each pair be: $n_1, n_2, \dots, n_m$. Hence, there are $n_1 + 1 + n_2 + 1 + n_3 + 1, \dots, n_i + 1 + \dots, n_m + 1 = m + \sum_{i=1}^m n_i \leq m + n \leq 2n$ lcp-delimited interval. When we find the i_{th} decreasing neighboring pair, we will output the n_i intervals. Thus, determining the lcp-delimited intervals can be done in $O(n)$ time. This linear time complexity can be compared with the work reported in [2], where they constructed a similar structure (the LCP-interval tree) in $O(n \log |\Sigma|)$ time.

Computing the Document Frequency (DF)

Determining the document frequency (DF) is more difficult than computing the term frequency (TF). In [131], an algorithm was given to calculate DF in $O(n \log n)$ time and $O(n)$ space. The

algorithm (reproduced below in Figure 2.1) uses the procedure described above to get the lcp-delimited intervals. It uses a new array to map the first symbol of the substring to a document *id* based on the order of the suffix array index. Using a simple algorithm, we could search in each interval to determine the document frequency. This will however be too expensive in time cost, leading to time in $O(n^2)$. Given that lcp-delimited intervals could be nested, we only need to check in the extra range over the calculated range. So the core of the algorithm is to check whether the document *id* of the current position has been previously computed. We reproduce the *DF* algorithm proposed in [131] below (see Figure 2.1). In Chapter 4, we modify this algorithm for an improved time complexity.

i, the left edge of an interval,
k, the representative ($SIL = lcp[k]$),
df, partial results for *df*, counting documents seen thus far, minus duplicates.

```

print_LDIs_with_df ← function(N){
  stack_i ← an integer array for the stack of the left edges, i.
  stack_k ← an integer array for the stack of the representatives, k.
  stack_df ← an integer array for the stack of the df counter.
  doclink[0..D] : an integer array for the document link initialized with -1.
  D = the number of documents.
  stack_i[0] ← 0.
  stack_k[0] ← 0.
  stack_df[0] ← 1.
  sp ← 1 (a stack pointer).
  (1) For j ← 0, 1, 2, ..., N - 1 do
  (2)   (Output a trivial lcp-delimited interval  $\langle j, j \rangle$  with  $tf = 1$  and  $df = 1$ .)
  (3)   doc ← get_docnum(s[j])
  (4)   if doclink[doc] ≠ -1, do
  (5)     let x be the largest x such that doclink[doc] ≥ stack_i[x].
  (6)     stack_df[x] ← stack_df[x] - 1.
  (7)     doclink[doc] ← j.
  (8)     df ← 1.
  (9)     While lcp[j + 1] < lcp[stack_k[sp - 1]] do
  (10)      df ← stack_df[sp - 1] + df.
  (11)      Output a nontrivial interval  $\langle i, j \rangle$  with  $tf = j - i + 1$  and df,
           if it is lcp-delimited.
  (12)      sp ← sp - 1.
  (13)      stack_i[sp] ← stack_k[sp - 1].
  (14)      stack_k[sp] ← j + 1.
  (15)      stack_df[sp] ← df.
  (16)      sp ← sp + 1. }

```

Figure 2.1. Algorithm for computing document frequency [131]

2.4 Circular Pattern Matching

2.4.1 String Pattern Matching

The pattern matching problem is to find the occurrences of a given pattern in a given text string. This is an old problem, which has been approached from different fronts, motivated by both its practical significance and its algorithmic importance. Matches between strings are determined based on the *string edit distance*. Given two strings $T : t_1 \dots t_n$ and $P : p_1 \dots p_m$, over an alphabet Σ , the edit distance indicates the minimum member of edit operations which transform one string into other string. There are three basic edit operations: *insertion* of a symbol, *deletion* of a symbol, and *substitution* of a symbol with another symbol. We assume that the cost of each edit operation is unity. If two characters are identical, the cost of the *match* operation is zero. The substitution operation can also considered as a *mismatch* operation. The *exact string matching problem* is to look for all occurrences of a pattern matching a substring of the text with zero edit operations. Various algorithms have been proposed for both exact and approximate pattern matching [3, 11, 20, 34, 46, 48, 64, 117].

Grossi and Vitter [45] pointed out that a full-text indexing system is expected to be able to support three basic types of queries, *existential query*: returns a binary value (true or false) indicating whether a pattern, P occurs in the text T ; *counting query*: returns n_{occ} , ($0 \leq n_{occ} \leq n$), the number of occurrences of P in T ; and *enumerative query*: returns n_{occ} numbers, each indicating the starting position in T , of an occurrence of P [3].

Exact String Matching

Three well-know efficient string matching algorithms with linear time complexity are the Knuth-Morris-Pratt(KMP) algorithm [64], Karp-Rabin algorithm [59] and the Boyer-Moore(BM) algorithm [20]. Like KMP, the BM algorithm matches the pattern and the text by skipping characters that are not likely to result in exact matching with the pattern. Unlike the other methods, it compares the strings from right to left of the pattern. These algorithms need an $O(m)$ preprocessing for the pattern and search in $O(n)$ or sometimes even sublinear in practice. The total time will be $O(n + m)$.

A different approach to pattern matching based on bitwise operations was introduced by R. Baeza-Yates and G. Gonnet [13]. Here, the pattern is represented by a binary mask. Bit-wise SHIFT and AND operations that are considered constant time are used to find the patterns. Under this framework, SHIFT and AND correspond to the pattern movement and matching respectively. The algorithm is effective for small patterns, when the pattern length is less than a computer word (say, 64 characters), which is usual for the text searching problem.

When multiple patterns need to be searched, alternative algorithms are used, such as the Aho-Corasick algorithm [5] which used a *keyword* tree for the set of patterns. In addition to multiple pattern matching, the suffix tree algorithm [46] is efficient when the pattern will be searched multiple times. There are linear time suffix tree construction algorithms available [89, 122, 128]. The search time for each pattern will be $O(m)$. The total time for searching s patterns will thus be $O(n + sm)$. This can be compared with $O(m + sn)$ that would be needed by algorithms such as KMP and BM.

Approximate String Matching

Algorithms for the approximate pattern matching problem can be grouped into three major categories: methods based on *dynamic programming* [114]; methods based on bit-wise operations [130]; and methods based on the longest common subsequence (*LCS*) [68]. The edit distance can be computed using dynamic programming. The path that leads to the minimal distance can easily be identified by adding trace-back pointers. The time complexity for computing the edit distance is generally in $O(mn)$. When the number of allowed errors k is known, the edit distance can be computed in $O(kn)$, for example using Ukkonen's deterministic finite state automaton (DFA) approach [121]. Typical approximate matching algorithms based on bit-wise operations are AGREP [130] and NRGREP [95]. These are obtained by extending the bit-wise exact matching algorithms. Another variation of pattern matching with errors is the k -mismatch problem, where only substitution operations are considered during the edit distance computation. In [68], the k -mismatch problem is solved using the suffix tree so that the *LCS* query can be answered in constant time. To find all the k -mismatch patterns of P in text T , we perform k -mismatch checks for every alignment of P , each time starting at different character position in T .

Dynamic programming is the most popular algorithm in the approximate string matching (ASM). From the classical searching for longest common subsequence (LCS) [31] to multiple strings alignment [25, 125]. It uses divide and conquer method to break a complex problem into subproblems by solving the subproblems first and then integrating the solutions to get the optimal answer. The search time is $O(mn)$ and its space requirement is $O(m)$. This algorithm can be improved to achieve $O(\frac{kn}{\sqrt{|\Sigma|}})$ [26] on average. Smith-Waterman algorithm is one of the most famous algorithms in this category which was first proposed by Temple F. Smith and Michael S. Waterman in 1981 [116] for local alignment. Needleman-Wunsch [96] algorithm is a general global alignment technique used in bioinformatics. Both of these algorithms are based on dynamic programming. Smith-Waterman [116] algorithm guarantees finding the optimal solution for local alignment and Needleman-Wunsch [96] algorithm finds the optimal solution for global alignment.

2.4.2 Circular Pattern Matching Problems

A circular shift is a mapping $f : \Sigma^* \rightarrow \Sigma^*, f^t(c_1 \dots c_r) = c_{t+1} \dots c_r c_1 \dots c_t$, where $0 \leq t \leq r - 1$ and r is the length of string $c_1 c_2 \dots c_r$. Thus, $f^0(c_1 \dots c_r)$ corresponds to the original string. Let $[s]$ be a set of circular shifts of string s , then $[s] = \{f^i(s) | 0 \leq i \leq |s| - 1\}$. Given two circular strings s_1 and s_2 , the edit distance between s_1 and s_2 , $ED(s_1, s_2)$, is the minimum number of edit operations needed to transform one member of $[s_1]$ to one member of $[s_2]$. This is defined as $ED_c(s_1, s_2) = \min\{ED[f^i(s_1), f^j(s_2)] | 0 \leq i \leq |s_1| - 1 \text{ and } 0 \leq j \leq |s_2| - 1\}$, where $|s_1|$ is the length of string s_1 and $|s_2|$ is the length of string s_2 , $ED[A, B]$ is the standard edit distance. We consider two major problems related to circular pattern matching (CPM) defined as follows.

Problem 1: Exact circular pattern matching (ECPM). Given one circular pattern $P = P[1 \dots m]$ and the text $T = T[1 \dots n]$, return all occurrences of circular string $[P]$ and its circular shifts inside text T without any error. $[P]$ is a match to text T at position $j \in [1 \dots n - m + 1]$ $\Leftrightarrow f^t(P) = T[j \dots j + m - 1]$, for some t , $0 \leq t \leq m - 1$.

Problem 2: Approximate circular pattern matching (ACPM1). Given text $T = T[1 \dots n]$, circular pattern $P = P[1 \dots m]$ and maximum error k , return “Matching” when the edit distance between text T and circular pattern P is less or equal to k . Thus, the result will be the matching pair $g = \{(T, P) | ED_c(P, T) \leq k, s.t. -k \leq n - m \leq k\}$.

This problem uses the existential query to look for circular pattern matches between text T and circular pattern P , where $-k \leq n - m \leq k$. This problem compares two sequences with circular edit distance less than k .

We also consider a harder variation of the ACPM problem with the extension $-k \leq n - m$. This variation is to find circular permutations of P **inside** T . This problem compares circular pattern P and substring of T with circular edit distance less than k . We define this variation more formally as Problem 3 below:

Problem 3: Approximate circular pattern matching problem 2 (ACPM2)

Given text $T = T[1..n]$, circular pattern $P = P[1..m]$ and maximum error k , return all positions where the circular string $[P]$ matched text T with at most k errors. $[P]$ is said to be a k -approximate match with text T at position $j \in [1..n - m - k + 1]$ if $ED_c(P, T[j..j+m]) \leq k$, where $0 \leq t \leq m - 1, -k \leq n - m$.

Comparing with the ECPM problem, the ACPM2 problem looks for all approximate matches within the given maximum error k . The ACPM1 problem looks for an approximate match between two whole strings text T and circular pattern P , but the ACPM2 problem looks for all approximate matches between every substring of text T and the circular pattern P and its circular shifts. The ACPM2 problem is the hardest of these three problems.

An extension of the CPM problem is the All-Against-All variant.

Problem 4: All-Against-All CPM problem

Given SeqDB, the database of sequences, the All-Against-All CPM problem is to compare each sequence in the database with every other sequence in the database for possible circular matches.

Similar to the standard CPM problem, we can also consider the ECPM and ACPM versions for the All-Against-All CPM problem.

2.4.3 Exact Circular Pattern Matching (ECPM)

Given a pattern P and a text T , the exact circular pattern matching problem is to find the position in T which matches a circular permutation of P . The exact circular pattern matching problem was first studied by Booth [19] in 1980. He proposed an algorithm to detect the lexicographically smallest conjugate of a word. Improved methods were proposed in [10, 36]. However, the focus of the algorithms was on the canonical rotation(s) of a word. ECPM was a particular case in that problem. The swap pattern matching problem [7] is related to the ECPM problem. It looks for an exact match of a swapped pattern P in text T . We can see that the ECPM problem is a particular case of the swap pattern matching problem. Gusfield [46] discussed the ECPM problem as an end-of-chapter exercise but did not provide an explicit solution. Shiloach [115] provided an algorithm for the ECPM problem. However both only solve the online version of the ECPM problem which does not use indexing to preprocess the text T .

Our work is perhaps more closely related Iliopoulos et al. [51] who proposed two algorithms that solve the ECPM problem using indexing. These two algorithms are based on suffix structures and the time complexity are $O(m \log \log n + n_{occ})$ and $O(m \log n + n_{occ})$ respectively, where n_{occ} is the number of occurs.

The first algorithm builds a new data structure CPI-I to index circular patterns. Two steps are used to find the circular pattern. First, the algorithm will index the text T in two suffix trees ST_T and $ST_{\bar{T}}$ where $\bar{T}$ is the reverse order of T . It also maintains two list $LL(R)$ and $LL(\bar{R})$ which are linked lists of all the leaf nodes from left to right in ST_T and $ST_{\bar{T}}$ respectively. Secondly, the algorithm will search two parts Q_1, Q_2 of each permutation of pattern P in ST_T and $ST_{\bar{T}}$ respectively. ST_T returns the occurrences of Q_1 and $ST_{\bar{T}}$ returns the occurrences of Q_2 . The algorithm finds the intersection of these two sets by using the two linked list $LL(R)$ and $LL(\bar{R})$.

The construction time and space complexity for this algorithm is $O(n \log^{1+\varepsilon} n)$, where $0 < \varepsilon < 1$ and n is the length of text T . The query time complexity is $O(m \log \log n + n_{occ})$.

The second algorithm used another new structure CPI-II to address the ECPM problem. The data structure CPI-II is constructed by the suffix array SA , inverse suffix array SA^{-1} and arrays Pre and Suf , where Pre is an array for the prefix of pattern P and Suf is an array for the suffix of pattern P . There are three steps involved. First, compute the interval of prefix

of pattern P into array Pre and the interval of suffix of pattern P into array Suf . There are m prefixes and suffixes in P . For each prefix or suffix, it is calculated using the previous prefix or suffix, so the time complexity is $O(\log n)$ using the suffix array SA and inverse suffix array SA^{-1} . Secondly, for each circular permutation pattern which can be constructed by $P[m-i]P[i]$ where $1 \leq i \leq m$, the algorithm finds the intervals of $P[m-i]$ and $P[i]$ in $Suf[m-i]$ and $Pre[i]$ separately. Thirdly, output the intersection of the intervals of $P[m-i]$ and $P[i]$.

The space complexity are $O(n)$ bytes for this algorithm implemented using in the suffix array. The time complexity for answering a query is $O(m \log n + n_{occ})$. When implemented using the compress suffix array [3,45], its space complexity will be $O(n \log n)$ bits, but the time complexity for queries increases to $O(m \log^2 n + n_{occ})$.

2.4.4 Approximate Circular Pattern Matching (ACPM)

The ACPM problem is to find k -approximate matches between circular pattern $[P]$ and text T . The naïve method for the ACPM problem is to use each of circular strings $f^t(P)$ to calculate the edit distance between T and $f^t(P)$, where m is the length of pattern and $0 \leq t \leq m-1$. Thus the dynamic programming procedure will be run m times. The time complexity of a naïve algorithm to compute $ED([P], T)$ is $O(m^2n)$.

Maes [81] published a “divide and conquer” algorithm to compute $ED([P], T)$ in $O(mn \log m)$. Up to now, this is the best theoretical result for computing the edit distance between a circular pattern and a text. Given the significance of Maes’ algorithm, we present the details below.

In theory, Maes [81] algorithm is the best algorithm. It uses “divide and conquer” to calculate the edit distance by using a dynamic program table. This algorithm constructs an edit graph between text T and string PP (Figure 2.2 [87]), where PP is a concatenation of pattern P to itself. In this edit graph, let $path_{(x_1, y_1) - (x_2, y_2)}$ be a path from vertex (x_1, y_1) to vertex (x_2, y_2) . For each vertex (x, y) on this path, we have $x_1 \leq x \leq x_2, y_1 \leq y \leq y_2$. The edit distance between T and $f^i[P]$ in the subgraph can be computed by following a path from vertex $(0, i)$ to vertex $(n, i+m-1)$, where $0 \leq i \leq m-1$. Let $Path^i$ be an optimal edit path between $f^i(P)$ and T in the edit subgraph. That is a path of minimum cost from vertex $(0, i)$ to vertex $(n, i+m-1)$. Maes’ algorithm is based on an important observation on the edit graph: if $Path^i$ and $Path^j$ are each

an optimal edit path, then $Path^i$ and $Path^j$ can not cross each other, where $0 \leq i < j \leq m-1$. We can see that when $Path^i$ and $Path^j$ has a crossing point say at (x, y) , then edit distance of $path_{(0,i)-(x,y)}$ is less than edit distance of $path_{(0,j)-(x,y)}$, whenever $i < j$. Thus $Path^j$ is no longer an optimal path. The set of paths $\{Path^t | i < t \leq j\}$ do not have optimal path, because each $Path^t$ has to cross $Path^i$ at the point (x, y) .

Figure 2.2. Edit Graph of T and PP [87].

Based on the above observations, the algorithm calculates $Path^i$ and $Path^j$ first, where $i < j$. If two paths do not cross each other, that means there is an optimal edit path between $Path^i$ and $Path^j$. Next, they calculate the path $Path^t$ in-between $Path^i$ and $Path^j$, where $i < t < j$. In this case, the time for calculating $Path^t$ is $O((j-i) \times n)$. When $Path^i$ and $Path^j$ cross each other, we do not need to calculate the path set $\{Path^t | i < t \leq j\}$ anymore, because these is no optimal path between them.

Following this idea, the algorithm calculates all optimal paths starting from $Path^0$ and $Path^m$, where $Path^m$ is the optimal path from $(0, m)$ to $(n, 2m-1)$, and $Path^0$ and $Path^m$ are parallel paths. Figure 2.3 illustrates this algorithm. The step (1) of Figure 2.3 shows this with time cost of $O(mn)$. In the second step(step (2) of Figure 2.3), the algorithm computes the optimal path $Path^{\frac{m}{2}}$ between $Path^0$ and $Path^m$ with time complexity of $O(mn)$. In the third step (step (3) of Figure 2.3), two optimal paths $Path^{\frac{m}{4}}$ and $Path^{\frac{3m}{4}}$ are computed. Time cost for computing each path is $O(\frac{mn}{2})$, hence time complexity of this step is $O(mn)$ too. The step 4 (step (4) of Figure 2.3) calculates four paths $Path^{\frac{m}{8}}$, $Path^{\frac{3m}{8}}$, $Path^{\frac{5m}{8}}$ and $Path^{\frac{7m}{8}}$. Time cost of

computing each path is $O(\frac{mn}{4})$, so time complexity of this step is $O(mn)$ too. And so on and so forth, there are $O(\log m)$ steps and time cost of each step is $O(mn)$. Even when there are some crossing points, the time for calculating each step is still $O(mn)$. Thus the total time complexity is $O(mn \log m)$. After getting all optimal paths, the minimum edit distance between text T and circular pattern P can be computed.

Figure 2.3. Maes' Algorithm

Gregor et al. [44] gave a $O(m^2n)$ algorithm, however, this is a data-dependent algorithm. In practice, the algorithm may reach to $O(mn)$ time complexity on average. Oncina [98] presented an algorithm which has the same time complexity as Gregor et al. [44]. Marzal et al. [87] provide a branch and bound algorithm which is based on Maes [81] algorithm. The worst case time complexity is the same as Maes algorithm, but with more efficient time complexity on average.

The above methods all produce complete results in calculation of the circular edit distance. Some studies [22, 90–92] also present suboptimal algorithms with reduced time complexity that runs on $O(mn)$, but with the possibility of missing some results. Bunke and Buhler [22] presented a suboptimal algorithm whose time complexity is $O(mn)$. Mollineda [90–92] published two algorithms based on Bunke algorithm [22] and showed in an experiment that the suboptimal solution is almost as good as the optimal counterparts.

We note that all the above methods on the ACPM problem have only considered the ACPM1 variant. To our knowledge, there has been no published work addressing the more challenging ACPM2 problem.

2.4.5 ACPM Problem in Protein Sequences

A number of studies have been reported on algorithms for detecting circular permutations for protein sequences [40, 47, 55]. The first method [54] used the dot matrix and human visualization to identify circular relationships between protein sequence pairs. The work in [6] used a dictionary method to find short fragments common to the protein sequence pairs and used human visualization to report the best local matches.

Needleman et. al [96] proposed a method for global alignment between two protein sequences. The global alignment algorithm measures the number of edit operations (insertion, deletion, and substitution) for transforming one sequence to another sequence. Uliel et al. [123, 124] introduced a method to detect circular permutations in protein sequences using global alignment [96]. They gave an $O(m^3)$ time complexity algorithm to find the complete set of matching circular permutations. They also proposed a greedy algorithm in $O(m^2)$ time complexity, but which could miss some valid circular permutations in the text T . Weiner et al. [126, 127] proposed another greedy method that runs in $O(m^2)$ time complexity. They focused on circular multidomain proteins, where the alphabet are now the protein domain blocks, rather than traditional protein symbols. Thus, $|\Sigma|$ could be quite large, of the order of 20^q , where q is the length of the domain blocks. This is the first application of the CPM problem in studying multidomain proteins. However, they did not consider the problems posed by the expanded alphabet.

The algorithm of Uliel et al. [123, 124] used a simple method that calculated the edit distance [72] between text and one of the circular permutations using the Needleman and Wunsch algorithm [96]. This was repeated m times for all the circular permutations, with $O(m^3)$ time complexity and $O(m^2)$ space complexity for each sequence as a pattern P against on the other protein sequences. The greedy algorithm of Uliel et al. [123, 124] modified the local alignment algorithm to find the best local alignment in a $2m \times n$ matrix. This algorithm is similar to the Smith-Waterman local alignment algorithm [116]. It is not guaranteed to find all circular

matches, and thus may miss some valid matches. The algorithm of Weiner et al. [126, 127] is also a greedy algorithm and thus could miss some valid matches. They concatenated the text T as TT and the pattern P as PP , and thus constructed a $2n \times 2m$ matrix using the Needleman and Wunsch algorithm [96]. At the verification phase, the circular matching condition must satisfy certain conditions defined on the $2n \times 2m$ matrix [126, 127].

More fundamentally, both groups [123, 124, 126, 127] that have studied CPM in protein sequences have focused on whole sequence comparison with another whole sequence. In their experiments, they have to group the protein sequences based on their specified lengths, and used the dissimilarity in lengths for initial pruning. These methods ignored the fact that a shorter circular protein sequence could be part of the functional region of a much larger multidomain protein. This, however, could be a key consideration in function prediction for multidomain proteins. Further, as with the more theoretical algorithms for the ACPM problem, the methods for protein sequences [123, 124, 126, 127] also only considered the ACPM1 problem.

2.5 Pattern Discovery Problem

Pattern discovery is a well studied problem in computational biology and data mining, and various methods have been proposed. The basic method is to identify short sequences that tend to be over-represented within a given set of sequences. Mining sequential patterns was studied in [30, 132]. Motif discovery methods in bioinformatics are surveyed in [112]. Algorithms for discovery of proximity patterns were proposed in [12]. Proximity pattern discovery is closely related to the more recent notion of "complex motif", which is defined as a composite motif whereby the individual components are constrained to be within a specified separation distance. Perhaps, a more closely related work is the method of pattern discovery using mutable permutation patterns [49]. However, although permutation patterns offer a lot of flexibility in the match, ignoring the order of the patterns still does not handle the problem of possible cyclic relations between patterns. Most efforts in pattern discovery have been invested in studying the statistical significance of the patterns (see for e.g. [85, 86]), and the biological relevance of the discovered patterns, in the case of biological applications [112]. There has not been much attention on the pattern matching problem involved, which forms the basis of pattern discovery.

Chapter 3

The Virtual Suffix Tree

3.1 Introduction

Our proposed data structure is most closely related to the ESA and LST. The virtual suffix tree can be constructed in the same time and space bounds as the suffix tree. It also supports basic search operations in the same time and space bound as the suffix tree. However, the VST requires a much smaller practical space than the suffix tree. The space requirement ($12.05n$ bytes using the compact form) is generally smaller than that of ESA and LST, the other closely related data structures (each requires $20n$ bytes). Other related data structures that have been proposed include the suffix cactus [56], suffix vectors [93, 103], compact suffix trees [82], the lazy suffix trees [41], level-compressed suffix trees [8], compressed suffix trees [94], and compressed suffix arrays [45]. See also [3].

Main results¹ We introduce a new data structure, the *virtual suffix tree* (VST), an efficient data structure for suffix trees and suffix arrays. The VST neither stores the `lcp` array nor the `lcp`-intervals, but rather exploits the inherent nature of the suffix tree topology. We state our main results in the form of two theorems about the VST.

¹Part of the work reported in this chapter has been published in the following papers: [78, 79]

Theorem 3.1: *Given a string $T = T[1..n]$, with symbols from an alphabet Σ , and the virtual suffix tree for T , we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Theorem 3.2: *Given a string $T = T[1..n]$, with symbols from an alphabet Σ , the virtual suffix tree, including the suffix links, can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ .*

Essentially, the VST provides the same functionality as the suffix tree, but at a much smaller space requirement. It has the same linear time construction for large $|\Sigma|$, requires $O(n)$ space to store, and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. To provide the complete functionality of the suffix tree, we describe a simple linear time algorithm that computes the suffix links based on the VST. We present two algorithms for VST construction. The first algorithm builds the VST from the suffix tree, which in turn is generated from the suffix array. The second algorithm eliminates the need for the suffix tree construction step, and thus builds the VST directly from the suffix array. Although the space needed for the VST is linear (as in suffix tree implementations using linked lists or binary trees), the practical space requirement is much smaller than that of a suffix tree. The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the ESA [2], and the LST [62]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form. This can be compared with the $20n$ bytes needed by the LST or the ESA.

Organization In Section 2, we introduce the basic data structure and discuss the properties of the VST. Section 3 presents an improved data structure, along with algorithms for its construction. A complexity analysis on the construction and use of the VST is also presented in this section. Section 4 shows how the suffix link can be constructed on the VST. In Section 5, we eliminate the need to construct the suffix tree, and show how the VST can be constructed directly from the suffix array. We make the summary in Section 6.

3.2 Basic Data Structure

Starting from the suffix array, we construct an efficient data structure to simulate the suffix tree (ST). We call this structure a virtual suffix tree (VST). The VST stores information about the basic topology of the suffix tree, the suffix array, and the suffix links. Thus, the VST is represented as a set of arrays that maintains information on the internal nodes of the suffix tree. The leaf nodes are not stored directly. However, whenever needed, information about any leaf node can be obtained via the suffix array. Unlike the ESA and LST, the VST neither uses the `lcp`-interval tree nor stores the `lcp` array. We call the data structure a virtual suffix tree in the sense that it provides all the functionalities of the suffix tree using the same space and time complexity as a suffix tree, but without storing the actual suffix tree. Later, we show that the VST leads to a more compact representation of suffix trees and suffix arrays. (We mention that [60] also used the term "virtual suffix tree", but for a limited form of the enhanced suffix array).

Below, we present the basic VST. This structure will require 14 bytes for each node in the VST and supports pattern matching in $O(m \log |\Sigma|)$ time, for an m -length pattern. In the next section, we present an improved data structure that reduces the space cost by eliminating the need to store edge lengths, while still maintaining $O(m \log |\Sigma|)$ time for pattern matching. We also describe a more compact structure for the VST that uses only 10 bytes for each internal node of the VST, and 5 bytes for each leaf node. Pattern matching on this compact representation will, however, be in $O(m|\Sigma|)$ time.

Each node in the VST corresponds to a distinct internal node in the suffix tree. In its basic form, each node in the VST is characterized by five attributes. For a given node in the VST (say node u), with a corresponding internal node in ST (say node u_{ST}), the five attributes are defined as follows.

- `sa_index`: index in the suffix array (SA index) of the leftmost leaf node under the internal node u_{ST} of the suffix tree.
- `fchild`: the node ID of the first child node of u_{ST} that is also an internal node. (Scanning is done left to right; edges at a node are also sorted left to right in ascending lexicographic

order). If node u is a leaf node in the VST, the value will be negative. The absolute value will point to the first child node of the next internal node in the VST.

- `elength`: The edge length of the edge (v, u) in the VST, or equivalently (v_{ST}, u_{ST}) in the suffix tree, where v is the parent node of u and v_{ST} is the parent node of u_{ST} .
- `nleaf`: the number of child leaf nodes *before* the first child of u_{ST} that is also an internal node.
- `nnleaf`: the number of sibling leaf nodes *after* u_{ST} , the current internal node of the suffix tree, but before the next sibling internal node.

In terms of storage, the `sa_index`, `fchild` and `elength` each requires one integer (4 bytes), while `nleaf` and `nnleaf` each requires one byte of storage (assuming $|\Sigma| \leq 256$).

3.2.1 Example VST

We use an example sequence to explain the above definitions. The suffix tree and VST for the string `missississippis` are shown in Figure 3.1. Note that the string `missississippis` is made intentionally different from `mississippi`, to capture some of the cases involved in a VST. Only the internal nodes (dark nodes) are explicitly stored in the VST. The leaf nodes (empty circles) are not stored. The order of storage is based on the node-depths, from top to bottom. Table 3.1 shows the corresponding values of the VST node attributes for each VST node in the example.

3.2.2 Properties of the Virtual Suffix Tree

We can trace the properties of the VST based on the standard properties of a suffix tree.

1. The VST only stores the internal nodes of the suffix tree. No leaf nodes in the ST are represented in the VST. Information about the leaf nodes can be obtained from the SA when needed. Then the space requirement of the VST depends on the topology of the the suffix tree, or more specifically, on the number of internal nodes.

2. The number of leaf nodes in a suffix tree is n . The number of internal nodes in the suffix tree (and hence number of nodes in the VST) is at most n .
3. The VST stores only the SA index of the leftmost leaf nodes and information about the child nodes.
4. For a given node in the VST, the number of child nodes will be no larger than $|\Sigma|$. Thus, the time needed to match a symbol is at most $O(\log |\Sigma|)$.
5. The nodes in the VST are ordered based on the internal nodes of the suffix tree using the hierarchy sequential access method (HSAM). The child nodes from any given node will be stored sequentially. The child nodes of two nearby nodes will therefore be stored in nearby locations. This is an important property for addressing problems involving locality of reference.

We introduce further definitions needed in the description below. For a given node u in the VST, we use the term *prior* node to denote the node that appears before the current node u in the HSAM ordering. Similarly, *next* node denotes the node that appears after the current node u in this ordering. We use `lsa_index` (left sa_index) to denote the SA index of the leftmost leaf node that is a descendant of u . Similarly, `rsa_index` (right sa_index) denotes the rightmost leaf node that has u as its ancestor. Figure 3.2 shows an example.

It is simple to determine the `lsa_index` and the leftmost child node of any given node. The properties of the VST and the organization of the VST lead to the following lemma about the VST:

Table 3.1. VST node attributes for the example sequence $T = \text{missississipp}\$$ used in Figure 3.1.

node	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
sa_index	0	1	7	9	3	9	12	4	10	13
fchild	N_1	N_4	$-N_5$	N_5	N_7	N_8	N_9			
elength	0	1	1	1	3	1	2	3	3	3
nleaf	1	2	2	0	1	1	1	2	2	2
nnleaf	0	1	0	0	0	0	0	0	0	0

Lemma 1: *For a given node in the VST, its rightmost child node, and the right `sa_index` can each be determined in constant time.*

Proof: Let u be the current node in the VST, with parent node v . Let w be the next node in the HSAM ordering. By property 5, if w is an internal node in the VST, the rightmost child (`rchild`) node of u will be the prior node to the leftmost child node of w . If w is a leaf node in the VST, then $w.fchild$ will point to the next node after u 's rightmost child node. Then the time to determine the rightmost child node will be $O(1)$.

For the right `sa_index`, if u has a next sibling node, say w , the right `sa_index` of u will be the left `sa_index` of this sibling node w minus the `nnleaf` of u . If u does not have a next sibling node, the right `sa_index` of u will be the right `sa_index` of node v (u 's parent) minus the `nnleaf` of u . That is,

$$u.rsa_index = \begin{cases} w.sa_index - u.nnleaf - 1 & : u \text{ has a next sibling, } w \\ v.rsa_index - u.nnleaf & : \text{otherwise} \end{cases} \quad (3.1)$$

Thus the time required to determine the right `sa_index` is $O(1)$. $\square$

3.2.3 Pattern Matching on VST

Lemma 1 provides an indication of how pattern matching can be performed on the VST. For pattern matching using the suffix tree, an important issue is how to quickly locate all the child nodes for a given internal node. In the VST, each node points to its leftmost leaf node using the `sa_index`. During pattern matching, at any given node in the VST, we will need to determine four parameters, namely the leftmost child node (`lchild`), the rightmost child node (`rchild`), the left `sa_index` (`lsa_index`) and the right `sa_index` (`rsa_index`). These parameters define the boundaries of the search at the given node. To search in a leaf node of the VST, we will need only the left `sa_index` and right `sa_index` of the node. When we search in an internal node, we will need all the four parameters to match a pattern. Lemma 1 shows that for any given node, we can determine each of these parameters in constant time. The following examples illustrate the two cases involved in computing the `rsa_index`, and how

pattern matching can be performed on the VST.

Example: *Determining the right boundary from a next sibling node.* Consider node N_5 in Figure 3.2. The left `sa_index` of N_5 is 9 and the right `sa_index` is 11, since $N_5.\text{sa_index}=9$ and $N_{5+1}.\text{sa_index}=12$, and hence the right `sa_index` of $N_5=12-1=11$. The leftmost child node is the `fchild` of the current node, thus the leftmost child of N_5 is N_8 . The next node of the rightmost child node is $N_{5+1}.\text{fchild}=N_9$. Then the rightmost child node is $N_{9-1}=N_8$, since the child node will be stored side by side between sibling nodes.

Example *Determining the right boundary from the right boundary of the parent node.* Consider node N_1 in Figure 3.2. The left `sa_index` of N_1 is $N_1.\text{sa_index}=1$. The right `sa_index` of N_1 is $N_2.\text{sa_index} - (N_1.\text{nnleaf} - 1) = 7 - 1 - 1 = 5$. The leftmost child node of N_1 is $N_1.\text{fchild}=N_4$. The next node of N_1 is N_2 . Since $N_2.\text{fchild}=-N_5$ is negative, N_2 must be a leaf node in the VST. We therefore know that the next node of the rightmost child node of N_1 will be N_5 . Finally, the rightmost child node of N_1 can be determined as $N_{5-N_1.\text{nnleaf}} = N_{5-1} = N_4$.

We summarize the foregoing discussion as the first main result of this work:

Theorem 3.1: *Given a string $T = T[1..n]$ of length n , with symbols from an alphabet Σ , and the virtual suffix tree for T , we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Proof: The theorem is a consequence of Lemma 1. First consider the cost of one single symbol-by-symbol comparison at a node in the VST. The number of child nodes at any internal node can be no larger than $|\Sigma|$, and we can find the boundaries of the search in constant time. Since the edges are ordered lexically at each internal node, and given the HSAM ordering, matching a single symbol can be done in $O(\log |\Sigma|)$ time steps using binary search. To find the first match, we need to consider the m symbols in the pattern. We perform the above symbol-by-symbol comparisons at most m times to decide whether there is a match or not. After a match is found, we can again use binary search (using `lsa_index` and `rsa_index` as bounds) to determine all the η_{occ} occurrences of the pattern. Reporting each occurrence can be done in constant time, or an additional η_{occ} time for all the occurrences. $\square$

3.3 Improved Virtual Suffix Tree

The basic data structure introduced above stores the length of each edge in the VST. We can improve the structure to reduce the space requirement by avoiding the need to store information about the edge lengths directly. The improved data structure has only four attributes rather than five. The attributes `sa_index` and `elength` in the basic structure are now combined into one attribute called the adjusted SA index (`asa_index`). This requires a key modification to the suffix tree, leading to an important distinction between the suffix tree and the virtual suffix tree.

3.3.1 Adjusting Edge Lengths

A well-known property of the suffix tree is that no two edges out of a node in the tree can start with the same symbol. For efficient representation of the VST, this characteristic of the ST is modified such that, for a given node, every edge that leads to an internal node in the VST has an equal length. This modification is done as follows: Start from the root node and progress towards the leaf nodes in the VST. For a given internal node, say u , adjust the edge label from u to each of its children such that all edges that lead to an internal node will have the same edge length. The major criteria is that, for two sibling internal nodes, their edge labels differ only in the last symbol. If for some edge, say (u, w) , the original edge length (or edge label) is longer than the new length, prepend the extraneous part of old $label(u, w)$ to each outgoing edge from w . The edge length for edges that lead to leaf nodes are left unchanged. Then repeat the adjustment at each child node of u . Figure 3.3 shows an example of this procedure. Observe that this adjustment only affects the edge lengths, and does not change the general topology of the suffix tree.

The above adjustment procedure leads to an important property of the VST:

Property: *In the improved VST, all internal sibling nodes occur at the same node-depth, and same string-depth, and the edge labels for the edges from the parent to each sibling differ only in the last symbol. This means that, in the VST, two branches from the same node can start with the same symbol, but their edge labels will differ.*

This property provides an important difference between the suffix tree and the VST. The

suffix tree mandates that no two edges from the same node have the same starting symbol. Further, the suffix tree only guarantees that the node-depth of two sibling nodes are the same, but not their string depth. This property of equal-length sibling edge labels is the key to more efficient representation of the VST, without explicit edge labels. Figure 3.4 shows an example of the modified suffix tree with equal-length edges for sibling nodes that are also internal nodes, and the corresponding improved virtual suffix tree. Table 3.2 shows the corresponding values of the attributes for each node in the improved VST. What remains is how we compute `asa_index`, the adjusted SA index. This is done by combining the original `sa_index` with `elength`.

Lemma 2 : *Given a node in the VST say u , and its parent node (say v), we can compute the adjusted SA index in constant time. Further, when required, the edge length can be determined in constant time.*

Proof: Computing the adjusted edge length (`new_elength`) and the adjusted SA index (`asa_index`) can be done using the following relations:

$$u.asa_index = \begin{cases} u.new_elength + u.sa_index & : \quad u = v.fchild \text{ and} \\ & u.new_elength \neq 1 \\ u.sa_index & : \quad \text{otherwise} \end{cases} \quad (3.2)$$

$$u.sa_index = u.fchild.sa_index - u.nfleaf \quad (3.3)$$

At time of VST construction, we calculate `asa_index` from bottom to top. For leaf nodes in the VST, we already know the `sa_index` and `new_elength`, then we can calculate the `asa_index` from Eqn (3.2). When the node u is an internal node in the VST, we first obtain `u.sa_index` from Eqn (3.3) since we know `u.nfleaf` and `u.fchild.sa_index`. Then we determine `u.asa_index` from Eqn (3.2). The new edge length is not stored explicitly in the VST nodes, but can be computed in constant time whenever needed (for instance, during pattern matching) by simply changing the subjects in Eqns (3.2) and (3.3). This is possible since at this time we already know `u.asa_index` for each node in the VST. $\square$

Thus while we store only the `asa_index`, our calculations will still use the original `sa_index`. However, this can be derived from the `asa_index` in constant time. In fact,

we can observe that in practice, we need to compute the `asa_index` for only the leftmost child node at each node-level, while keeping the original `sa_index` for all other nodes. To determine the `new_elen` for these other nodes, we simply make a constant time access to their leftmost (sibling) node (at the same node-level), and then use this to compute the length. For searching with the VST, we will calculate the length of the common string at each level. If the length is greater than 0, then we know there is a common string in the edge labels for the child nodes and only the last character is different. Thus, we do not need to store the edge lengths explicitly, leading to a reduction of one integer per node over the basic VST.

Table 3.2. Node attributes in the improved VST for the example sequence, $T = \text{missississipi\$}$.

NodeName	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
<code>sa_index</code>	0	1	7	9	3	9	12	4	10	13
<code>fchild</code>	N_1	N_4	N_5	N_5	N_7	N_8	N_9			
<code>new_elen</code>	0	1	1	1	1	1	1	3	1	2
<code>nleaf</code>	1	2	2	0	1	1	1	2	2	2
<code>nnleaf</code>	0	1	0	0	0	0	0	0	0	0
<code>asa_index</code>	0	1	7	9	3	9	12	4+3=7	10	13+2=15

We have included `new_elen`, so one can compare with `elen` in Table 3.1. However, in practice this will not be stored in the VST.

3.3.2 Construction Algorithm

Construction of the VST makes use of an array Q which records the internal nodes of the suffix tree. This array maps the internal nodes of the suffix tree to nodes in the VST. Thus, elements in the array are in the same ordering as the corresponding nodes in the VST.

Given an input string T , the first step is to construct the suffix array for T . This can be done in worst case linear time and linear space using any of the existing algorithms [57,63,65]. Using the SA, we construct the suffix tree as described in [3]. While the suffix tree can be constructed directly in linear time, working from the SA to the ST will require less space for the construction. The suffix tree is then preprocessed in linear time to adjust the edges from a given parent node that lead to internal child nodes to equal-length edges. Using the adjusted suffix tree, the algorithm will process the internal nodes in the suffix tree in a top-down manner

to determine the attributes (`fchild`, `nleaf` and `nnleaf`) for the corresponding nodes in the VST. Next, we process the VST from the VST leaf nodes to the root, using the Q array to update the `asa_index` at each node. The adjusted `asa_index` field includes information on the `sa_index` and edge length.

The steps for constructing the VST for a given input string are summarized in Algorithm 3.1.

3.3.3 Further Space Reduction

We can further reduce the space needed by the VST, at the cost of an increased time for pattern matching. In the pattern matching phase, if the algorithm is to compare symbols one-by-one, rather than using binary search on the branches from a given node in the VST, we will only need to compute the `lsa_index` and `rsa_index` of the node.

Consider an arbitrary node (say node u) in the VST. The number of children from u or the number of u 's leaf nodes cannot be larger than $|\Sigma|$. Thus, the `sa_index` of any child node of u will lie between node u 's `lsa_index` and `rsa_index`. Then comparing one symbol from the pattern against the first symbol on each edge from u to its children will require at most $O(|\Sigma|)$ time steps. The left child node and the right child node will not need to be used again. Thus, the attributes `fchild` and `nleaf` in the leaf nodes of the VST are no longer required. We make the `asa_index` to be negative for the leaf nodes. Thus, during pattern matching, this serves as a flag for the VST leaf nodes. This compact structure will reduce the space requirement at each leaf node of the VST by 5 bytes. Time for pattern matching, however, will increase to $O(|\Sigma|)$ for each symbol in the pattern P , or $O(m|\Sigma|)$ overall, where $m = |P|$.

3.3.4 Complexity Analysis

Time and space complexity.

The time cost for lines 1-3 in the construction algorithm CONSTRUCT-VST (Algorithm 3.1) is $O(n)+O(n)+O(n)=O(n)$. Lines 5-17 in the algorithm perform a one time traversal of the nodes

in the suffix tree. The respective values of $pTop$ and $pBottom$ range from 1 to $2n$. Thus the cost for the traversals is $O(n)$. Lines 18-27 in the algorithm run at most $pBottom$ times. The time for lines 18-27 in the algorithm is thus $O(n)$, since each iteration of the loop requires constant time. Therefore, for the regular VST, the overall construction time is $O(n)$. The time for pattern matching is in $O(m \log |\Sigma|)$. For the compact structure, the construction time is the same as the regular structure, but the VST is no longer stored linearly. Here we use an array to store the relation between the Q array and the compact VST. The searching time is now $O(m|\Sigma|)$.

The space requirement clearly depends on the number of nodes in the VST, which is at most n for a sequence of length n . Each node requires a fixed amount of memory to store, leading to an $O(n)$ space requirement.

Number of nodes and practical space requirement.

The actual space needed for the VST depends on the topology of the suffix tree. This topology can be captured by the number of internal nodes in the suffix tree, or alternatively, by the quantity R_{IL} , the ratio between the number of internal nodes and the number of leaf nodes. We call R_{IL} the *density* or *branching factor* for the suffix tree. We conducted an experiment to evaluate the effect of this branching factor on the storage requirement of the VST. The suffix tree was constructed and the branching factors computed for a set of files taken from [104]. For each file, we used the first 2^{24} symbols as the text, and computed the branching factor. Table 3.3 shows the results. The maximum ratio of 0.76 was observed for the file `Jdk13c`. On average, however, the maximum ratio was around 0.63. The worst case occurs for a sequence with $|\Sigma| = 1$, (that is, $T = a^n$), leading to a branching factor of 1. The table shows that, for a given sequence, the branching factor depends on a complex relationship between n , $|\Sigma|$, and the mean LCP.

The space requirement for the VST, for both the compact and regular structures depends directly on the branching factor. The last two columns in Table 3.3 show the maximum space requirement for each file.

The foregoing discussion leads to the following lemma on VST construction:

Lemma 3: *Given a string $T = T[1..n]$, with symbols from an alphabet Σ , the virtual suffix tree (without suffix links) can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ .*

3.4 Computing Suffix Links

Constructing the suffix tree from the suffix array as described in [3] does not include the suffix link. There are also a number of other suffix tree construction algorithms that build the suffix tree without the suffix link. See for example, Farach et al. [38], and Cole and Hariharan [29]. The suffix link, however, is a significant component of the suffix tree, and is important in certain applications, such as approximate pattern matching using matching statistics, and other forms of traversal on the suffix tree. Thus, a data structure to support the complete functionality of the suffix tree requires an inclusion of the suffix link. Recent efficient data structures for suffix trees have thus provided mechanisms for constructing the suffix link. The ESA [2] provided suffix links using complicated RMQ preprocessing [18]. The LST [62] also supported suffix links using the `lcp`-interval tree and intervals defined on the inverse suffix array. A recent work by Maaß [80] focused exclusively on suffix link construction from suffix arrays, or from suffix trees that do not have such links.

The virtual suffix tree provides a natural mechanism for constructing suffix links. The key idea is that suffix links in the VST can be computed bottom-up, from the nodes with the highest node-depth (leaf nodes) in the VST to those with the least (the root). This is based on the following two observations about suffix links.

1. Consider a leaf node u_{ST} in the suffix tree corresponding to suffix T_i in the original sequence. The suffix link from u_{ST} will point to the leaf node corresponding to the suffix T_{i+1} (that is, the suffix that starts at the next position in the sequence).
2. The suffix link from a node u in the VST will point to some node w with a smaller string-depth in the VST, such that $|L(u)| = |L(w)| + 1$ (or equivalently $|L(u_{ST})| = |L(w_{ST})| + 1$).

The following lemma establishes how we can build suffix links on the VST.

Lemma 4 *Given the VST for a string $T = T[1..n]$ of length n , the suffix links can be constructed in $O(n)$ time using an additional $O(n)$ space.*

Proof. Let u and w be two arbitrary nodes in the VST. Let v be the parent node of u . Let $u.\text{slink}$ be the node to which the suffix link from node u points. We consider two cases:

Case A: *u is a leaf node in the VST.* Then, using the above observations, the suffix link from node u will point to node w in the VST (that is, $u.\text{slink} = w$) such that $SA[w.\text{sa_index}] = SA[u.\text{sa_index}] + 1$. Clearly, $|L(w)| = |L(u)| - 1$, where $L(x)$ is the path label of node x . Note that this path label is not explicitly stored in the VST, but for each node, the length can be computed in constant time. This computation can be performed in constant time by maintaining two arrays and observing that $n - |L(w)| = n - |L(u)| + 1$. One array is the inverse suffix array (ISA) for the given string, defined as follows: $ISA[i] = j$ if $SA[j] = i$, ($i, j = 1, 2, \dots, n$). The second is an array M that maps the SA values to the corresponding parent nodes in the VST, defined as follows: $M[i] = u$, if u_{ST} in ST is the parent node of the leaf node corresponding to the suffix $T_{SA[i]}$. Clearly, both arrays can be computed in linear time, and require linear space.

Case B: *u is not a leaf node in the VST.* This is a simpler case. When u is an internal node in the VST, the suffix link from u will point to some node w , such that w is an ancestor of node $u.\text{fchild}.\text{slink}$, such that $|label(u, u.\text{fchild})| = |label(w, u.\text{fchild}.\text{slink})|$. The $O(n)$ time result then follows by using the skip/count trick [46], by observing that a VST has at most n nodes, a node depth of at most n , and that each upward traversal on the suffix link decreases the node depth by at least 1. $\square$

Although the above description is from the viewpoint of a VST already constructed, the suffix links can be constructed as the VST is being built, by some modification of the VST construction algorithm. Algorithm CONSTRUCT-VST-WITH-SUFFIXLINKS (Algorithm 3.2) shows a modification of Algorithm algorithm CONSTRUCT-VST (Algorithm 3.1) to incorporate sections to compute the suffix links. The suffix link construction algorithm is based on the Q array used during the VST construction. We observe that the additional work required to construct the suffix links on the VST is independent of the alphabet size.

Figure 3.4 shows the result of the suffix link algorithm when applied to the VST of our example string $T = \text{missississipi}\$$. Essentially, given the VST, the suffix link is con-

structed right to left, node-depth by node-depth, starting with the rightmost node at the deepest node-depth, and moving up the VST until we reach the root. Thus, the order of suffix link construction in the example will be $SL_1, SL_2, \dots, SL_9$.

Algorithm 3.2 shows that the additional work required to compute all the suffix links is linear in the length of the string. After construction, the suffix link on the VST will require one additional integer per internal node in the VST. This can be compared with the 2 integers per node required to store the suffix link using the ESA, or LST. In a typical VST, where the maximum branching factor is usually less than 0.7, the suffix link will require a maximum extra space of $0.7n * 4 = 2.8n$ bytes. Table 3.4 shows the space required for the VST (including the suffix array and suffix links) for both the compact structure and the regular VST, at varying values of the branching factor.

We summarize the above discussion in the following theorem which captures the second main result of the work:

Theorem 3.2: *Given a string $T = T[1..n]$, with symbols from an alphabet Σ , the virtual suffix tree, including the suffix links, can be constructed in $O(n)$ time and $O(n)$ space, independent of Σ .*

Proof. The theorem follows directly from Lemma 3 and Lemma 4. $\square$

3.5 From SA to VST

So far, we have constructed the VST by first building the suffix tree from the suffix array, and then converting the suffix tree to a VST. The major problem with this approach is the relatively large memory requirement for suffix tree construction (for instance, compared to its storage). In this section, we eliminate this problem by constructing the VST directly from the suffix array, without a need to first construct the suffix tree.

The VST mainly encodes the structural information in a suffix tree, while avoiding the need to store some information that could be computed from the encoded structure. Thus, the key to going from SA to VST directly is to observe how the SA encodes the structural information

in a suffix tree. The observation is that, given a sequence, the branching information in its suffix tree can be determined by making use of the corresponding suffix array and `lcp` array of the sequence. The edge labels, and hence edge lengths can be determined by analyzing the differences between adjacent `lcp` values. In a sense, it was this same observation that was used in constructing the suffix tree from the suffix array in [3] which was exploited in Algorithm 3.1.

Like the suffix tree, the VST has two types of nodes, leaf nodes and non-leaf nodes. The VST encodes only the non-leaf nodes in the suffix tree. Each leaf node in the VST corresponds to an internal node in the suffix tree whose child nodes are all leaf nodes in the suffix tree. The suffix tree leaf nodes in turn point to positions in the suffix array. Thus, to determine the VST nodes and their respective attributes from the suffix array, we consider whether the node is a VST leaf node, or a non-leaf node. We call the former `Type 0` nodes, and the later `Type 1` nodes. The problem then is to determine how the VST node attributes are derived from the SA and `lcp` for each type of node. We take a two step approach. First, we scan the SA and `lcp` from left to right, and use a temporary data structure to record pertinent information about each node in the VST. The temporary data structure (denoted TA) will be an array of structures, (similar to a VST node structure), but a TA node will contain more information than a VST node. Each node in the VST has a corresponding entry in TA. In the second stage, we construct a mapping function (denoted MAP) that provides a one-to-one map from the elements in TA to the VST nodes. At this stage, some attributes in TA are renamed, and non-required fields in the TA structure are removed to give the required virtual suffix tree.

The first stage makes use of two structures – the TA structure and a stack data structure. The stack structure has two elements, the stack value (an integer) and the stack type (one bit). The stack type shows the VST node types described above. Type 0 indicates an unmerged leaf node, while Type 1 indicates an unmerged internal node. The TA structure contains the same attributes as a VST node (`sa_index`, `fchild`, `nflleaf`, and `nnleaf`), in addition to two pointers, namely, `next` which points to the next sibling node of the current node, and `rsa_index`, the `rsa_index` of the current node.

The algorithm scans the suffix array (SA) and `lcp` array (LCP) from left to right, (assumes the suffixes are sorted left to right in ascending order), and determines whether to create a new node based on the `lcp` values. The condition for starting the procedure to create a new node is when `LCP[Stack.top.index]` is larger than `lcp` of current index. We exit from the

procedure when $LCP[Stack.top.index]$ is less than lcp of current index. When entering the node creation procedure, we use $curNode$ to denote the current index, and the $curLCP$ to denote the lcp of current index. Whenever we exit the procedure, we run a special exiting function for housekeeping, which could also create a new node. We make use of the following definitions:

$$Stack.top.index = \begin{cases} Stack.top.value & : \text{Stack.top.type} = 0 \\ TA[Stack.top.value].rsa_index & : \text{Stack.top.type} = 1 \end{cases} \quad (3.4)$$

$$curNode.sa_index = \begin{cases} curNode.value & : \text{curNode.type} = 0 \\ TA[curNode.value].sa_index & : \text{curNode.type} = 1 \end{cases} \quad (3.5)$$

Essentially, a node is created by merging an existing internal node with another internal node, or with a leaf node, or by merging two leaf nodes to form an internal node. The algorithm makes use of several auxiliary routines, depending on the type of node. There are two cases, corresponding to the two node types:

1. CASE 1: VST LEAF NODES.

Here, $Stack.top.type = 0$. We consider two sub-cases.

Case 1A: $LCP[Stack.top.index] > curLCP$

- **Case 1A1:** $LCP[Stack.top.index] \neq LCP[curNode.sa_index]$

We merge $Stack.top.index$ and $curNode$ to a new element of TA , say $T[k]$.

Let $Stack.top.index$ be the leftmost leaf node and $curNode$ be the rightmost child. The required update is performed using the **merge1A1()** routine

described as follows: If $curNode$ is an index for SA , then $T[k]$ has two leaf

nodes: if $(curNode.type = 0)$, then update $T[k]$ as follows: $T[k].sa_index =$

$Stack.top.index$, $T[k].nleaf = 2$ (since there are 2 leaf nodes), $T[k].rsa_index$

$= curNode.value$. If $curNode$ is an element of TA , then $T[k]$ has one leaf node,

and one child node; then, update $T[k]$ as follows: $T[k].sa_index = Stack.top.index$, $T[k].fchild = curNode.value$, $T[k].nleaf = 1$ (since there is one leaf node), $T[k].rsa_index = TA[curNode.value].rsa_index$.

- **Case 1A2:** $LCP[Stack.top.index] = LCP[curNode.sa_index]$.

In this case, $curNode$ must be an element of TA . We update the node as follows:

$TA[curNode.value].sa_index = Stack.top.index$,

$TA[curNode.value].nleaf = TA[curNode.value].nleaf + 1$, and pop the stack.

Case 1B: $LCP[Stack.top.index] = curLCP$.

Again, in this case, $curNode$ must be an element of TA . We simply update the number of leafs, namely, $numleaf = numleaf + 1$ and pop the stack

2. CASE 2: VST INTERNAL NODES.

Here $Stack.top.type = 1$. We also consider two sub-cases.

Case 2A: $LCP[Stack.top.index] > curLCP$.

We merge $TA[Stack.top.value]$ and $curNode$ to a new element of TA , say $T[k]$.

The first child node will be $TA[Stack.top.value]$, and the next (sibling) node will be $curNode$. The required update is performed using the **merge2A()** routine described as follows: If $curNode$ is an index for SA , then $T[k]$ has one leaf node.

Update $T[k]$ as follows: $T[k].sa_index = TA[Stack.top.value].sa_index$, $T[k].fchild = TA[Stack.top.value]$, $T[k].nleaf = 0$, $T[k].rsa_index = curNode.value$,

$TA[Stack.top.value].nleaf = 1$ (this leaf is $curNode$). If $curNode$ is an element of TA , then $T[k]$ has two child nodes. If $LCP[Stack.top.index] = LCP[Stack.(top-1).index]$ then pop the stack. (These two must be an element of TA). Then, update $T[k]$ as follows:


```

T[k].sa_index = TA[Stack.top.value].sa_index,
T[k].fchild = TA[Stack.top.value], T[k].nleaf=0,
T[k].rsa_index = TA[curNode.value].rsa_index,
TA[Stack.top.value].next = curNode.value.

```

Case 2B: $\text{LCP}[\text{Stack.top.index}] = \text{curLCP}$.

The update here is performed using the **merge2B()** routine, described as follows: If curNode is an index for SA, then, update as follows: $\text{TA}[\text{Stack.top.value}].\text{nleaf} = \text{TA}[\text{Stack.top.value}].\text{nleaf} + \text{numleaf} + 1$. If curNode is an element of TA, then update as follows: $\text{TA}[\text{Stack.top.value}].\text{nleaf} = \text{TA}[\text{Stack.top.value}].\text{nleaf} + \text{numleaf}$, $\text{TA}[\text{Stack.top.value}].\text{next} = \text{curNode.value}$.

The special exit housekeeping procedure is performed using **exitFunction()**. The procedure is described as follows: If $\text{numleaf} = 0$, then push curNode into stack. Otherwise, (so we must have $\text{numleaf} \neq 0$), then let $T[k]$ be the node resulting from merging curNode with the leaf which has the same LCP value as curLCP . Note that curNode is an element of TA. Update $T[k]$ as follows: $T[k].\text{sa_index} = \text{TA}[\text{curNode.value}].\text{sa_index} - \text{numleaf}$, $T[k].\text{nleaf} = \text{numleaf}$, $T[k].\text{fchild} = \text{curNode.value}$, $T[k].\text{rsa_index} = \text{TA}[\text{curNode.value}].\text{rsa_index}$. Push $T[k]$ into the stack, (equivalently, push (k) and set stack type to 1), and increment k by 1. If $\text{curLCP} \leq \text{lcp}$ of next index then push curNode into the stack).

Algorithm 3.3 shows the steps for constructing the TA structure, given the SA and LCP array. Figure 3.5 shows the VST nodes (nodes in TA) created using the algorithm on our running example string $T = \text{missississipi}\$$. Table 3.5 shows the attributes of each node in the TA structure. Notice that some nodes, such as $\text{TA}[2]$ and $\text{TA}[4]$ were updated at later steps of the algorithm, after their initial creation.

Algorithm 3.4 shows how the TA node labels are mapped to VST node labels. The algorithm computes elength for the TA nodes, in order to compute asa_index , the adjusted sa_index which is used in the VST to avoid storing the edge lengths. Table 3.5 shows the result of the mapping for the TA structure shown in Fig. 3.5 and Table 3.5. The algorithm uses a simple auxiliary function **computeChildren.elengths()** to determine the edge lengths.

Building the suffix links on the VST structure above can be done as was done earlier. The two algorithms still maintain the linear time construction for the VST.

3.6 Summary

In this work, we have presented the virtual suffix tree (VST), an efficient data structure for suffix trees and suffix arrays. The searching performance is the same as the suffix tree, that is, $O(m \log |\Sigma|)$ for a pattern of length m , with symbol alphabet Σ . We also showed how suffix links can be constructed on the VST in linear time, independent of the alphabet size. The VST does not store the edge lengths explicitly. This is achieved by modifying a key property of the suffix tree - the requirement that no two edges from a given node in the suffix tree can start with the same symbol. This key modification leads to a major distinction between the VST and the suffix tree, and results in extra space saving. However, whenever needed, the length for any arbitrary edge in the VST can be obtained in constant time using a simple computation. A further space reduction leads to a more compact representation of the VST, but at the expense of an increased search time, from $O(m \log |\Sigma|)$ to $O(m|\Sigma|)$.

The space requirement depends on the topology of the suffix tree, in particular, the branching factor. For the compact structure, the worst case space requirement (including the suffix array) is $11.5n$ bytes without suffix links, and $15.5n$ bytes with suffix links, where n is the length of the string. However, in practice, the branching factor is typically less than 0.7. For the compact structure, this gives less than $9.25n$ bytes on average without the suffix links, or $12.05n$ bytes with suffix links.

In this work, we started from efficient storage of the suffix tree and suffix array after they have been constructed. Thus, we constructed the VST from the suffix tree, which in turn was constructed from the suffix array. To reduce the space requirement at construction time, we introduced another algorithm that constructs the VST directly from the suffix array. An interesting question is whether one can construct compressed versions of the VST, in a way that is analogous to compressed suffix trees and compressed suffix arrays. This could lead to further space saving for the VST.

Figure 3.1. Suffix tree and virtual suffix tree for the string $T = \text{mississippi}\$$.

(a) suffix tree ; (b) virtual suffix tree. The number at each leaf node indicates the position in SA. The number at each internal node indicates the node ID in the VST.

Figure 3.2. Example VST (solid nodes) showing left SA index (ISA) and right SA index (rSA) for sample nodes.

Figure 3.3. Edge-length adjustment procedure.

Figure 3.4. Improved VST for the string $T = \text{mississippi}\$$

Algorithm 3.1: VST Construction Algorithm

```

CONSTRUCT-VST( $T, n$ )
1   $SA \leftarrow \text{COMPUTE-SUFFIXARRAY}(T, n)$ 
2   $ST \leftarrow \text{SUFFIXTREE-FROM-SUFFIXARRAY}(SA)$ 
3   $ST \leftarrow \text{ADJUST-EDGELNGTHS}(ST)$ 
4  Initialize  $VST[], Q[], pTop=0, pBottom=0, curNode=root, Q[pTop]=root$ 
5  while ( $pBottom \geq pTop$ )
6    for ( each  $childnode$  in  $curNode$ ) do
7      if ( $childnode$  is internal node in  $ST$ ) then
8         $pBottom \leftarrow pBottom + 1; Q[pBottom] \leftarrow childNode$ 
9        if  $childnode$  is first internal node then
10          $VST[pTop].fchild \leftarrow pBottom$ 
11        end if
12      else
13        Update  $VST[pTop].nleaf$  and  $VST[pBottom].nleaf$ 
14      end if
15    end for
16     $pTop \leftarrow pTop + 1; curNode \leftarrow Q[pTop]$ 
17  end while
18  for ( $pb \leftarrow pBottom$  down to 0) do
19    if ( $VST[pb]$  is leaf node) then
20       $VST[pb].asa\_index \leftarrow Q[pb].fchild$ 
21    else if ( $Q[pb].elength=1$ ) then
22       $VST[pb].asa\_index \leftarrow VST[pb].fchild.asa\_index$ 
23         $+ VST[pb].nleaf - Q[pb].elength$ 
24    else
25       $VST[pb].asa\_index \leftarrow VST[pb].fchild.asa\_index$ 
26         $+ VST[pb].nleaf - Q[pb].elength + Q[pb].elength$ 
27    end if
28  end for

```

Table 3.3. Branching factor and maximum space requirement for various sample files.

File	$ \Sigma $	Max Ratio	Compact	Regular	Description
Bible	63	0.61	8.60n	10.13n	King James bible
Chr22	5	0.73	9.50n	11.33n	Human chromosome 22
E.coli	4	0.65	8.89n	10.52n	<i>Escherichia coli</i> genome
Etext	146	0.54	8.02n	9.36n	Texts from Gutenberg project
Howto	197	0.55	8.13n	9.51n	Linux Howto files
Jdk13c	113	0.76	9.69n	11.59n	JDK 1.3 documentation
Rctail	93	0.66	8.95n	10.60n	Reuters news in XML format
Rfc	120	0.64	8.77n	10.36n	Concatenated IETF RFC files
Sprot	94	0.61	8.54n	10.05n	
World	94	0.54	8.06n	9.41n	CIA world fact book
Average		0.63	8.71n	10.29n	

Algorithm 3.2: VST construction with suffix links

 CONSTRUCT-VST-WITH-SUFFIXLINKS(T, n)

```

4   Initialize VST[], Q[], ISA[], M[], pTop ← 0, pBottom ← 0, curNode ← root, Q[pTop] ← root
    ⋮
18  for (pb ← pBottom down to 0) do
19    if (VST[pb] is leaf node) then
20      Update array  $M$  to map SA index and node VST [pb]
    ⋮
26  end if
27  end for
28  for (pb ← pBottom down to 0) do
29    if (VST[pb] is leaf node) then
30      VST[pb].slink ← M[ISA[VST[pb].sa_index+1]]
31    else
32      Find ancestor  $w$  of VST[pb].fchild.slink s.t.
         $|label(w, VST[pb].fchild.slink)| = |label(VST[pb], VST[pb].fchild)|$ 
33      Set VST[pb].slink ←  $w$ 
34    end if
35  end for
```

Figure 3.5. Suffix links on the VST for the sample string $T = \text{missississippis\$}$.

(a) suffix links on the VST, but showing the leaf nodes of the suffix tree; (b) suffix links on VST (no ST leaf nodes).

The suffix links are labeled $SL_1, SL_2, \dots, SL_9$, indicating the order in which they were constructed

Table 3.4. Storage requirement for the VST, including suffix links

	Ratio	Compact	Regular
Worst Case	1	15.50n	18.00n
Average Case	0.75	12.63n	14.50n
	0.7	12.05n	13.80n
	0.65	11.48n	13.10n
	0.6	10.90n	12.40n

Table 3.5. Detailed attributes for nodes in the TA data structure using the sample sequence,

$T = \text{missississippis\$}$.

	TA[0]	TA[1]	TA[2]	TA[3]	TA[4]	TA[5]	TA[6]	TA[7]	TA[8]	TA[9]
label	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
sa_index	4	3	1	0	7	10	9	13	12	9
fchild	null	TA[0]	TA[1]	TA[2]	null	null	TA[5]	null	TA[7]	TA[6]
next	null	null	TA[4]	null	TA[9]	null	TA[8]	null	null	null
rsa_index	5	5	5	5	8	11	11	14	14	14
nleaf	2	1	2	1	2	2	1	2	1	0
nnleaf	0	0	1	0	0	0	0	0	0	0

Figure 3.6. Constructing VST from the suffix array.

Nodes are labeled based on their labels in the temporary array, TA. The mapping of the TA node labels to the corresponding VST node labels is shown in Table 3.5.

Table 3.6. Node mapping table from TA to VST

TA nodes	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
VST nodes	N_7	N_4	N_1	$root$	N_2	N_8	N_5	N_9	N_6	N_3

Algorithm 3.3: Constructing VST From Suffix Array

```

CONSTRUCT-VST-FROM-SA( $LCP[]$ ,  $SA[]$ )
1   Stack ← buildStack();  $TA[n]$ ;  $k \leftarrow 0$ ;  $st.value=0$ ;  $st.type=0$ ; push(st)
2   for ( $i \leftarrow 1$  to  $n$ ) do
3       if ( $LCP[i] \geq LCP[Stack.top.index]$ ) then
4            $st.value=i$ ;  $st.type=0$ ; push(st)
5       else
6            $curLCP \leftarrow LCP[i]$ ;  $curNode.value \leftarrow i$ ;  $curNode.type \leftarrow 0$ ;  $numleaf \leftarrow 0$ 
7           do while (Stack is not empty &  $curLCP \leq LCP[Stack.top.index]$ )
8               if ( $Stack.top.type=0$  &  $LCP[Stack.top.index] > curLCP$ ) then
9                   if ( $LCP[Stack.top.index] \neq LCP[curNode.sa\_index]$ ) then
10                        $TA[k] \leftarrow merge1A1(Stack.top.index, curNode)$ 
11                        $curNode.value \leftarrow k$ ;  $curNode.type \leftarrow 1$ ;  $k \leftarrow k+1$ 
12                   else
13                        $TA[curNode.value].sa\_index \leftarrow Stack.top.index$ ;
14                        $TA[curNode.value].nleaf \leftarrow TA[curNode.value].nleaf+1$ 
15                   end if
16               else if ( $Stack.top.type=0$  &  $LCP[Stack.top.index]=curLCP$ ) then
17                    $numleaf \leftarrow numleaf+1$ 
18               else if ( $Stack.top.type=1$  &  $LCP[Stack.top.index] > curLCP$ ) then
19                    $TA[k] \leftarrow merge2A(TA[Stack.top.value], curNode)$ 
20                    $curNode.value \leftarrow k$ ;  $curNode.type \leftarrow 1$ ;  $k \leftarrow k+1$ 
21               else
22                    $merge2B(TA[Stack.top.value], curNode)$ 
23                   pop(Stack); break
24               end if
25           pop(Stack)
26       end while
27        $exitFunction()$ 
28   end for
29    $root \leftarrow$  the value of last element of Stack
30   return MAP-TA-TO-VST( $TA[]$ ,  $k$ ,  $ROOT$ )

```

Algorithm 3.4: Mapping TA nodes to VST nodesMAP-TA-TO-VST($TA[], k, \text{root}$)

```

1  MAP[0..k-1]; W[0..k-1]; pTop ← -1; curNode ← root
2  TA[curNode].fchild.length ← 1
3  for ( $j \leftarrow 0$  to  $k-1$ ) do
4      MAP[j] ← curNode
5      computeChildren_lengths(curNode)
6      if ( $\text{curNode.next} \neq \text{NULL}$ ) then
7          curNode ← curNode.next
8      else
9          pTop ← pTop + 1
10         do while ( $\text{TA}[\text{MAP}[\text{pTop}]].\text{fchild} = \text{NULL}$ )
11             pTop ← pTop + 1
12         end while
13         curNode ←  $\text{TA}[\text{MAP}[\text{pTop}]].\text{fchild}$ 
14         if ( $\text{TA}[\text{curNode}].\text{length} > 1$ ) then
15              $\text{TA}[\text{curNode}].\text{sa\_index} \leftarrow \text{TA}[\text{curNode}].\text{sa\_index} + \text{TA}[\text{curNode}].\text{length}$ 
16         end if
17          $\text{TA}[\text{MAP}[\text{pTop}]].\text{fchild} \leftarrow j+1$ 
18     end if
19 end for
20 for ( $j \leftarrow 0$  to  $k-1$ ) do
21      $W[\text{MAP}[j]] \leftarrow j$ 
22 end for
23 for ( $j \leftarrow 0$  to  $k-1$ ) do
24     if ( $\text{MAP}[j] \geq 0$ ) then
25         index ← j
26         SWAP1 ←  $\text{TA}[\text{MAP}[\text{index}]]$ 
27         do while ( $\text{MAP}[\text{index}] \geq 0$ )
28             SWAP2 ←  $\text{TA}[\text{index}]$ 
29              $\text{TA}[\text{index}] \leftarrow \text{SWAP1}$ 
30             SWAP1 ← SWAP2
31              $\text{MAP}[\text{index}] \leftarrow -1$ 
32             index ←  $W[\text{index}]$ 
33         end while
34     end if
35 end for
36 Remove next, sa_index, length from TA
37 Return TA as VST

```

Chapter 4

The Probabilistic Suffix Array

4.1 Introduction

It has earlier been shown [109] that the *probabilistic suffix tree* (PST) is equivalent to the probabilistic suffix automata which is a type of probabilistic finite automata (PFA). The PFA on the other hand can be viewed as a variable length Markov model (VLMM), whose memory length constraint is determined by the observed data. We present the probabilistic suffix array (PSA), a new data structure for representing information in variable length Markov chains. The PSA essentially encodes information in a VLMM by providing a space-efficient representation of the probabilistic suffix tree (PST). Our PSA provides the same functionality as the PST, but at a reduced space requirement. The equivalence between the PST and a class of PFAs implies that our PSA is also equivalent to this class of PFAs.

Main Resultes. We present algorithms to construct the PSA and for sequence prediction based on the constructed PSA. Our algorithms are based on the notion of empirical probabilities, modeled using information retrieval notions of term frequency (TF) and document frequency (DF). We present a linear time algorithm for computing the document frequency. We state our main results in the form of two theorems about the PSA.

Theorem 4.1: *Given a sequence $T = T[1...n]$, with symbols from an alphabet Σ , and the memory constraint L on the variable length Markov model, the probabilistic suffix array (PSA) for T can be constructed in $O(n)$ time, and $O(n)$ space, independent of the Markov order, L .*

Theorem 4.2: *Given a sequence $T = T[1...n]$, with symbols from an alphabet σ , where $\sigma = \sigma_1\sigma_2...\sigma_{|\Sigma|}$, and the probabilistic suffix array (PSA) for T , we can decide on whether a pattern $P = P[1..m]$ is generated by the same variable length Markov chain that generated T in $O(m \log \frac{n}{|\Sigma|})$ time.*

In previous work by Ron et al. [109] and Apostolico and Bejerano [9], the probabilistic finite automata (PFA) was represented using the probabilistic suffix tree (PST). Here, we present a space-efficient data structure to simulate such finite state machines when represented as a PST. Since a variable length Markov model is a finite state machine, and can be represented as a PST, our proposed structure can be used to capture the information in a variable length Markov model. We call our data structure the *probabilistic suffix array* (PSA), since it is built on suffix arrays rather than the suffix tree data structure. The PSA uses an array of nodes to capture the branching structure in the suffix tree, and other auxiliary arrays to maintain information needed for learning from the observed data. Learning in the PSA is performed by computing conditional probabilities at each node in the PSA using empirical probabilities computed via the TF and DF .

Organization. In the next section, we briefly describe the PST using an example sequence. In Section 3, we present our PSA data structure. We also give an example for the PSA to explain how it works. The construction algorithm is presented in Section 4. We analyze its practical space requirement in Section 5. Section 6 presents experimental results on protein family classification and phylogenetic tree construction. The last section provides a summary and conclusion.

4.2 Probabilistic Suffix Tree

Given a sequence T with n observations, the Markov model needed to represent T will require space that is exponential in L , the order of the Markov model. In practice, the transition

matrix for the Markov model to represent such a sequence will be sparse as the order L increases. Suffix tree data structures represent all substrings of a sequence seq in $O(n)$ internal nodes and leaf nodes. Ron et al. [109] presented a space-efficient data structure called the *probabilistic suffix tree* (PST) to represent the order L transition matrix. The PST encodes only the non-zero transition probabilities. Therefore, the PST is a suffix tree which contains the transition probabilities for the Markov model with any order L , where $0 < L \leq n$. However, in practice, the space requirement for the suffix tree is still a problem.

Consider the sample sequence $T = \text{accactact}\$$. Its first order transition matrix and the corresponding state diagram are shown in Figure 4.1. Figure 4.2 shows an example suffix tree and the corresponding PST, for this example sequence. The PST is shown for the case of order $L = 3$. In this PST, we label the transition probability in each symbol. The transition probability for a given symbol is calculated using the conditioning context.

Figure 4.1. State diagram and transition matrix for a first order Markov model for an example sequence $T = \text{accactact}\$$.

4.3 Proposed Data Structure

We propose the *probabilistic suffix array* (PSA) as a way to simulate the probabilistic suffix tree. Each node in the PSA has a corresponding node in the PST. The basic PSA structure has three types of attributes. The first category of attributes are the *foundation attributes*, which consists of the original text and its suffix array. Construction of the suffix array is in $O(n)$ time

and space, using any of the various linear-time linear-space algorithms. See for example [4, 57, 97, 104]. The second category of attributes are the *internal node attributes*. These are derived from the interval array, which is determined following [131]. These are used to represent the internal nodes in the suffix tree including the suffix links. The suffix link is the link from an internal node to its suffix node. The third type of attributes are *measurement attributes*. These record measurement information, such as term frequency, document frequency, and conditional probabilities, which are needed to compute probabilities in the Markov model. Table 4.1 shows the three categories of attributes used in the PSA. We use the term *length of PSA* to refer to the number of nodes in the PSA. The term *length* emphasizes the fact that our PSA nodes are stored as arrays. In this work, we use M to denote the PSA length.

The internal node attributes are derived from the interval array. The pair $\langle Start, End \rangle$ represents the interval position of this node in the suffix array. *Length* denotes that length

of longest common prefix between the substrings represented by the node. This essentially corresponds to the length of the path from root to the current node. The PSA internal node attributes are used to simulate the internal nodes in a suffix tree. The attribute *Suffix Link* is a regular suffix link from the current internal node to its suffix node. We use this link to continue the searching process when a mismatch occurs at the time of prediction. The internal node attributes including the suffix link are constructed using Algorithm BUILDPSA.

4.3.2 Measurement Attributes

These attributes are dependent on the application. For example, for applications in document feature selection, or in protein sequence classification, we only compute and store *TF* and *DF*, and the conditional probabilities. In other applications, such as document clustering, we may need to compute document frequency for the classes, rather than just the document frequency. The attributes are also dependent on the methods used in calculating the probabilities in the Markov model. Thus, we focus on the method of computing the conditional probabilities, and the probability of a node in the VLMM. For a given node with node label, say $(S_1 \dots S_n)$, its probability, P_{VLMM} is given by:

$$P_{VLMM} = P(S_1 \dots S_n) = P(S_n | S_1 \dots S_{n-1}) \dots P(S_2 | S_1) P(S_1) \quad (4.1)$$

If $S = S_1 \dots S_{t-1} S_t$ (where $1 \leq t \leq n$) does not occur in the training data, we find the longest suffix of S which occurred in the training data. Assume the $S_k \dots S_{t-1} S_t$ (where $1 \leq k \leq t$) is the

Table 4.1. Attributes of PSA

Type	Attribute	Space
Fundamental Attributes	Original Text(text)	char
	Suffix Array(SA)	integer
Internal Node Attributes	Start	integer
	End	integer
	Length	integer
	Suffix Link	integer
Measurement Attributes	Term Frequency(TF)	integer
	Document Frequency(DF)	integer
	cProbability $P(S_t S_k \dots S_{t-1})$ (CP)	float

longest suffix of S , then $P(S_t|S_1...S_{t-1}) = P(S_t|S_k...S_{t-1})$. Thus,

$$P(S_t|S_1...S_{t-1}) = P(S_t|S_k...S_{t-1}) = \begin{cases} \frac{TF_{S_t}}{n} & : k = t \\ \frac{TF_{S_k...S_t}}{TF_{S_k...S_{t-1}}} & : k < t \end{cases} \quad (4.2)$$

Here, TF_u is the term frequency of the node with node label u . We make two observations: First, if the terminal symbol S_t of a path $S_k...S_{t-1}S_t$ is a first symbol in an edge of the suffix tree, then the conditional probability $P(S_t|S_k...S_{t-1})$ can be computed as the frequency of the current node divided by the frequency of the parent node. Secondly, if the terminal symbol S_t of a path $S_k...S_{t-1}S_t$ is a non-first symbol in an edge of the suffix tree, then the conditional probability $P(S_t|S_k...S_{t-1})$ is 1. We call this a *trivial conditional probability*. Thus, we need to store the conditional probabilities for only the first symbols in each edge. When we determine that the terminal symbol of a path is a non-first symbol in an edge, we simply return 1 for the conditional probability of the symbol.

4.3.3 Example PSA

Table 4.2 and Table 4.3 show the nature of the PSA nodes and the order-3 conditional probabilities in a PSA, using the sequence $T = \text{accactact}\$$ used in Figures 4.2 and 4.1. The entries in Table 4.2 are directly calculated from the interval array. We notice that $P(c|a)$ is 1. This indicates the substring “ac” is represented on one edge and that the terminal symbol “c” is the second symbol on this edge. This is an example of a *trivial conditional probability*. $P(c|ta)$ also is an example whose probability is 1.

Entries in Table 4.3 are computed from the PSA leaf nodes. We only showed the non-trivial conditional probabilities on the leaf nodes. These conditional probabilities are calculated based on the first observation described in Section 4.3.2. The numerator is 1 since this is a leaf node which must have a frequency of 1. The denominator is the frequency of the substring corresponding to the node label of the parent of the current leaf node. This is easily obtained as $(End - Start + 1)$ using the elements in the interval array. The PSA nodes can be compared with nodes in the example PST shown in Figure 4.2. The transition matrix in Figure 4.1(b) can be constructed in the PSA.

PSA node Index	Interval	Length	Suffix Link	Probability Expression	Conditional Probability
1	$\langle 2, 3 \rangle$	3	2	$P(t ac)$	$\frac{2}{3}$
2	$\langle 1, 3 \rangle$	2	3	$P(a)$	$\frac{1}{3}$
2	$\langle 1, 3 \rangle$	2	3	$P(c a)$	1
3	$\langle 6, 7 \rangle$	2	4	$P(t c)$	$\frac{1}{2}$
4	$\langle 4, 7 \rangle$	1	-1	$P(c)$	$\frac{4}{9}$
5	$\langle 8, 9 \rangle$	1	-1	$P(t)$	$\frac{2}{9}$

Table 4.2. Example PSA internal nodes, using the PSA of the sequence $T = \text{accactact}\$$

SA index	Probability Expression	Conditional Probability
1	$P(c ac)$	$\frac{1}{3}$
3	$P(a act)$	1
4	$P(a c)$	$\frac{1}{4}$
5	$P(c c)$	$\frac{1}{4}$
7	$P(a ct)$	$\frac{1}{2}$
9	$P(a t)$	1

Table 4.3. Example PSA leaf nodes, using the PSA of the sequence $T = \text{accactact}\$$

4.3.4 Interval Array and Document Frequency in Linear Time

The interval nodes represent the basic structure of the PSA. Since other attributes are based on the structure of these interval nodes, we will need to compute these interval nodes first. The term frequency (TF), and document frequency (DF) are computed as by products, as we build the interval nodes. In this section, we give a new algorithm to compute DF in linear time. Compared with the original algorithm [131] which runs in $O(n \log n)$ time, our algorithm is more efficient and applicable to large document collections. The interval nodes are stored as an array using the interval array data structure. To build the interval array, we use Yamamoto and Church's TF algorithm [131]. In our structure, the non-trivial lcp-delimited intervals represent the internal nodes of the suffix tree. Therefore, we modify the algorithm to output only the non-trivial lcp-delimited intervals. After this procedure, we obtain the interval array which includes interval attributes $\langle \text{Start}, \text{End} \rangle$ and Length for each node. Length is simply the LCP of the interval represented by $\langle \text{Start}, \text{End} \rangle$. The attribute TF is also computed at this stage.

For applications that involve multiple sequences or documents, for instance, in text clustering or in protein sequence classification, we may require the document frequency. In this work, the input sequence in such applications will be a concatenation of all the sequences, with a special end of document symbol (\$) delimiting each individual sequence.

The original algorithm proposed in [131] for computing document frequency runs on $O(n \log n)$ time. Here, we modify the algorithm to improve its running time to $O(n)$. The original algorithm was reproduced in Figure 2.1 for easy reference.

In line 4-6, the algorithm searches for the largest x . The worst case for this search will be $O(\log n)$. We add a new array *docsp* that maps the document *id* to a stack. The length of this array is the number of documents, Z . When we calculate a new interval $\langle \text{Start}, \text{End} \rangle$, the algorithm will check whether the element has been observed previously. The algorithm searches *docsp* with the document *id* of the new element. If the document *id* is found, the document frequency (*DF*) is changed. The process performs a simple look-up using *docsp* in constant time, and hence the modified algorithm runs in $O(n)$ time. Algorithm COMPUTEDOCUMENT-FREQUENCY implements the proposed modifications.

Algorithm 4.1: Computing Document Frequency in Linear Time

COMPUTEDOCUMENTFREQUENCY

- (3) $\text{doc} \leftarrow \text{getdocnum}(s[j]), \text{docsp}[\text{doc}] \leftarrow \text{sp}$
 - (4) **if** $\text{doclink}[\text{doc}] \neq -1$, **do**
 - (5) **if** $\text{docsp}[\text{doc}] > \text{sp}$ **or** $\text{stack}[\text{docsp}[\text{doc}]].i > \text{doclink}[\text{doc}]$ **do**
 - (5.5) $\text{stack_df}[\text{docsp}[\text{doc}]] \leftarrow \text{stack_df}[\text{docsp}[\text{doc}]] - 1$
 - (6) $\text{doclink}[\text{doc}] \leftarrow j, \text{docsp}[\text{doc}] \leftarrow \text{sp}$
-

4.4 Constructing the PSA

Having described the building blocks for the probabilistic suffix array (PSA), we are now ready to describe how we put them together to construct the PSA. Algorithm BUILDPSA (Algorithm 4.2) uses five major steps or procedures in constructing the PSA data structure for a given input sequence. The first step is the construction of the suffix array from the original sequence. We use standard linear-time linear-space algorithms for this step. Using Yamamoto

and Church's `PRINT_LDIS_STACK()` function [131], we construct the interval array. We then simulate the tree-like structure from the interval array in the third step. The third step maps each position in the input sequence to its interval. In the forth step, the routine `BUILDSUFFIXLINK` starts by first constructing the inverse suffix array W . From W , the algorithm determines a link that allows the interval array to point to the next position. Thus, the suffix link is easy to construct.

The final procedure constructs a ranked list of the elements in the interval array (the nodes in the PSA) in a non-decreasing order. This process uses the counting sort in $O(M)$ time and using $2M$ integer space, where M is the number of non-trivial lcp-delimited intervals. Since $M \leq n$, this will take $O(n)$ time.

Algorithm 4.2: Building the Probabilistic Suffix Array

`BUILDPSA(Text)`

```

1   SA ← BUILDSA(Text)
2   IntervalArray ← PRINT_LDIS_STACK(SA)
3   <posInterval,parent> ← BUILDINTERVALTREE(SA,IntervalArray)
4   PSA ← BUILDSUFFIXLINK(posInterval,parent)
5   PSA ← SORTPSA(PSA)
```

4.4.1 Building the Interval Tree

Algorithm `BUILDINTERVALTREE` (Algorithm 4.3) uses the interval pairs $\langle Start, End \rangle$ to construct a tree-like structure that encodes the parent-child relationships between the intervals. The main idea is to set the interval ID into a position between positions $Start$ and End , such that the position has not been previously set. The problem is that the pairs $\langle Start, End \rangle$ can overlap (nesting). A naïve algorithm for this task will require an $O(n^2)$ time.

We use a stack to store the free position which is the current pair $\langle Start, End \rangle$. If the position has not been earlier set to some interval ID, the algorithm sets the interval ID one by one from $Start$ to End . If the position has earlier been set to some interval ID, the algorithm will set the interval ID in the last part of the pair $\langle Start, End \rangle$ which has not earlier been set

to an interval ID. This process guarantees that the position will be set to one interval ID, and that the position is included in only one $\langle Start, End \rangle$ pair.

Some positions which could not be set to an interval ID in the last pass will be set to an interval ID in the next loop. The following step pops positions from the stack and sets interval IDs into these positions. This process guarantees that every position will be set to one interval ID, even the positions that belong to the next higher level interval, or to a lower level interval.

After determining the interval IDs, the algorithm computes the *parent* array. The *parent* array stores the link from one interval to the higher interval position which shares the same value for *Start*. The variable *pN* is a stack. This stack stores the intervals which have been computed, but the parent's interval ID has not yet been set. Line 16 pushes the interval into the stack.

When the current interval includes the interval at the top of the stack *pN*, it means the current interval is the parent of the interval at the top of *pN*. Thus, we set the value for *parent* of the interval at the top of *pN* to the current interval. The loop in lines 13 to 15 repeat the step to find all the child intervals of the current interval.

4.4.2 Building the Suffix Link

Algorithm BUILDSUFFIXLINK (Algorithm 4.4) uses the suffix array (SA) to compute the inverse suffix array *W*. To compute the suffix link, the algorithm sets the position of the parent in *W* to be the current position. Thus we obtain the suffix link for each leaf node. To compute the suffix link of an internal node, we simply use the *parent* array to find the suffix link of the internal node.

4.4.3 Sorting the PSA Structure

After obtaining the suffix link, we resort the PSA structure to make it suitable for efficient searching during the VLMM prediction stage. The prediction procedure performs frequent searches using the interval array. (See Section 4.4.5). After sorting using the $\langle Start, Length \rangle$ attributes of the PSA, searching will be done in $O(\log M)$ time, where *M* is the PSA length, the

Algorithm 4.3: Buliding the Interval Tree

```

BUILDINTERVALTREE( $SA[], Start[], End[], Length[]$ )
1   posInterval[]  $\leftarrow$  -1; pt  $\leftarrow$  0
2   for ( $i \leftarrow 1$  to  $M$ ) do
3       if posInterval[i]=-1 do //first time to observe  $Start[i]$ 
4           for ( $j \leftarrow Start[i]$  to  $End[i]$ ) do
5               if posInterval[j]=-1 do posInterval[j]  $\leftarrow$  i else break end if
6           end for
7           for ( $j \leftarrow End[i]$  down to  $Start[i]$ ) do
8               if posInterval[j]=-1 do posInterval[j]  $\leftarrow$  i else break end if
9           end for
10          end if
11          pop position pos which is between  $Start[i]$  and  $End[i]$  in Stack
12          posInterval[pos]  $\leftarrow$  i; push the position between  $Start[i]$  and  $End[i-1]$ 
13          while pt > 0 and  $Start[i] \leq Start[pN[pt-1]]$ 
14              parent[pN[pt-1]]  $\leftarrow$  i; pt  $\leftarrow$  pt-1
15          end while
16          pN[pt]  $\leftarrow$  i; pt  $\leftarrow$  pt+1
17      end for
18      return < posInterval[], parent[] >

```

Algorithm 4.4: Buliding Suffix Link

```

BUILDSUFFIXLINK( $SA[], Start[], Length[], posInterval[], parent[]$ )
1   for ( $i \leftarrow 1$  to  $n$ ) do  $W[SA[i]-1] \leftarrow i$  end for
2   for ( $i \leftarrow 1$  to  $n$ ) do  $W[i] \leftarrow posInterval[W[i]]$  end for
3   for ( $i \leftarrow 1$  to  $M$ ) do
4       if  $Length[i] \neq 1$  and sLink = NULL do
5           sLink[i]  $\leftarrow$   $W[SA[Start[i]]]$ 
6           k  $\leftarrow$  parent[i]; pi  $\leftarrow$  i
7           do while ( $k \neq \text{root}$  and sLink[k] = NULL)
8               sLink[k]  $\leftarrow$  parent[sLink[pi]]
9               do while ( $Length[sLink[k]] \neq Length[k]-1$ )
10                  sLink[k]  $\leftarrow$  parent[sLink[k]]
11              end while
12              pi  $\leftarrow$  k; k  $\leftarrow$  parent[k]
13          end while
14      end if
15  end for

```

number of the interval pairs in the PSA.

The algorithm SORTPSA (Algorithm 4.5) uses counting sort to sort the PSA structure. In the interval array, the attribute *Length* is already in decreasing order for intervals that share the same *Start*. Therefore, we only need to sort the structure based on the order of the *Start* attribute. The time complexity of this algorithm is linear with respect to the number of nodes (number of intervals). The additional space is at most $2n$ integers, see analysis in Section 4.5.

Algorithm 4.5: Sorting the PSA Structure

SORTPSA(*Start*[], *Length*[], *sLink*[])

```

1   Count[]  $\leftarrow$  0; Order[]  $\leftarrow$  0; wOrder[]  $\leftarrow$  0; NewStart[]  $\leftarrow$  0; NewEnd[]  $\leftarrow$  0
   NewLength[]  $\leftarrow$  0; NewsLink[]  $\leftarrow$  0; sum  $\leftarrow$  0; NewPattern[]  $\leftarrow$  0
2   for (i  $\leftarrow$  1 to M) do Count[Start[i]]  $\leftarrow$  Count[Start[i]]+1 end for
3   for (i  $\leftarrow$  1 to n) do
4       if Count[i]  $\neq$  0 do sum  $\leftarrow$  sum+Count[i]; Count[i]  $\leftarrow$  sum-Count[i]; end if
5   end for
6   for (i  $\leftarrow$  1 to M) do
7       Order[Count[Start[i]]]  $\leftarrow$  i; Count[Start[i]]  $\leftarrow$  Count[Start[i]] + 1
8   end for
9   for (i  $\leftarrow$  1 to M) do wOrder[Order[i]]  $\leftarrow$  i end for
10  for (i  $\leftarrow$  1 to M) do sLink[i]  $\leftarrow$  wOrder[sLink[i]] end for
11  for (i  $\leftarrow$  1 to M) do
12      NewStart[i]  $\leftarrow$  Start[Order[i]]; NewEnd[i]  $\leftarrow$  End[Order[i]]; NewLength[i]  $\leftarrow$  Length[Order[i]]
13      NewsLink[i]  $\leftarrow$  sLink[Order[i]]; NewsPattern[i]  $\leftarrow$  pattern[Order[i]]
14  end for
```

4.4.4 Computing Conditional Probabilities Using the PSA

Algorithm COMPUTEPROBABILITY (Algorithm 4.6) is a simple routine which is based on the PSA structure. It computes the conditional probability at each internal node by using equation (4.2). The algorithm uses the temporary array *parent* which was generated by algorithm BUILDSUFFIXLINK (Algorithm 4.4). The array *parent* contains a record of the parent for each given interval node. The algorithm is linear in time and is in place. After computing the conditional probabilities, the space used by array *pattern* can be released, since the array is no longer needed.

We summarize the above results in Theorem 4.1, our first main result on the PSA:

Theorem 4.1: *Given a sequence $T = T[1\dots n]$, with symbols from an alphabet Σ , and the memory constraint L on the variable length Markov model, the probabilistic suffix array (PSA) for T can be constructed in $O(n)$ time, and $O(n)$ space, independent of the Markov order, L .*

Proof: Algorithm BUILD_SA and PRINT_LDIS_STACK each runs in $O(n)$ time complexity as in [104] and [131] respectively. Algorithm BUILD_INTERVAL_TREE computes two arrays *posInterval* and *parent*. The array *posInterval* stores the relationship between leaf nodes and interval nodes, while *parent* represents the parent nodes for the interval nodes. There are n leaf nodes and M interval nodes. Lines 2-17 in algorithm BUILD_INTERVAL_TREE searches each interval node. Lines 4-6 makes the relationships between current interval node and the leaf nodes before the first child interval node of the current interval node. Lines 7-9 calculates the relationships between current interval node and leaf nodes after the last child interval node of current interval node. Line 11 computes the relationships between current interval node and leaf nodes which is not included in other interval nodes. Each leaf node will be computed one time in algorithm BUILD_INTERVAL_TREE. Lines 13-15 compute the child nodes of the current interval node. The time cost is $O(M)$ over the whole algorithm, even with the loop in Line 2. Algorithm BUILD_SUFFIX_LINK calculates the suffix link via the inverse suffix array W . It calculates suffix links starting at the lowest level interval nodes which does not include other interval nodes. Then it calculates the suffix link for the parent node of current interval node following the *parent* array. So the time is $O(M)$ for computing the suffix links for all interval nodes. There are 6 loops in algorithm SORT_PSA. Each loop runs either n times or M times. So the time complexity is $O(n)$. Therefore, overall, the probabilistic suffix array (PSA) for a sequence T of length n can be constructed in $O(n)$ time.

Linear space requirement follows from the space analysis in Section 4.5. $\square$

4.4.5 Prediction with VLMM via the PSA

An important procedure in Markov models is to compute the probability that a given test pattern is generated by the model. For the variable length Markov model (VLMM), we denote

Algorithm 4.6: Calculating Conditional ProbabilityCOMPUTEPROBABILITY(PSA, M, n)

```

1  for ( $i \leftarrow 1$  to  $M$ ) do
2      if  $PSA.Length[i]=1$  do
3           $PSA.cProbability[i] \leftarrow PSA.TF[i]/n$ 
4      else
5           $PSA.cProbability[i] \leftarrow PSA.TF[i]/PSA.TF[PSA.parent[i]]$ 
6      end if
7  end for

```

this probability as the P_{VLMM} of the input pattern. Algorithm VLMM-PREDICTION (Algorithm 4.7) calculates the P_{VLMM} of a test sequence. This algorithm uses Equation (4.1) to compute $P(S_1), P(S_2|S_1) \dots$ by searching for the sub-patterns $S_1, S_1S_2, \dots$. When there is a mismatch while searching with the sub-pattern $S_k \dots S_t$, the algorithm jumps to the node pointed to by the suffix link attribute of the current node. Thus, matching proceeds from the node representing the suffix $S_{k+1} \dots S_t$, after accounting for the prefix of this suffix, which has already been matched in the previous step.

The algorithm scans positions in the pattern from left to right. While matching the sub-pattern, it uses the function `LeftmostMatchedPosition` which uses standard suffix array search algorithms [3, 84] to find the left most position (*leftPosn*) that matched the sub-pattern. The function `LeftmostMatchedPosition` searches the pattern $Pattern[s \dots i]\#$, where $\#$ is a symbol that never occurred in the alphabet. That is, $\# \notin \Sigma, \# < \sigma, \forall \sigma \in \Sigma$, and $\$ < \#$. The search will result in a mismatch. Since $Pattern[s \dots i]$ has already matched up to position $i - 1$ using the SA, we can easily determine the leftmost position of $Pattern[s \dots i]$ in the suffix array. We use *leftPosn* to denote this leftmost position in the algorithm. The function `SearchPSA` uses the determined leftmost position to search in the PSA. This function also uses standard suffix array search algorithms [3, 84]. The function determines the index of the PSA node such that $PSA.index.Start = leftPosn$ and $PSA.index.Length$ is the minimum value that is larger than $i - s$, the length of matching prefix of the pattern. Since the PSA is already sorted by *Start* and *Length*, the search for the index will be done in $O(\log M)$ time complexity, where M is the length of PSA (i.e. the number of nodes in the PSA).

When a mismatch occurs, the algorithm uses the previously computed index to determine

the suffix link. Thus the search will be redirected to the new branch following the suffix link. The algorithm now uses the longest suffix of the sub-pattern that so far matched as the new sub-pattern, and re-starts matching from the symbol that mismatched. Determining this point where matching should re-start is a constant time operation.

The foregoing implies that, given a sequence $T = T[1..n]$, with symbols from an alphabet Σ , and the PSA for T , we can decide on whether a pattern $P = P[1..m]$ is generated by the same variable length Markov chain that generated T in $O(m \log \frac{n}{|\Sigma|})$ time.

We can use the predicted probability above to perform protein sequence classification. Suppose we have F protein families and we have computed the PSA for each family. Let PSA_k be the model constructed using the k -th protein family. Further, let $P_{VLM}(P, PSA_k)$ be the probability that protein sequence P is generated by PSA_k , as returned by Algorithm VLMM-PREDICTION. Then, we classify P to protein family f , where f is given by:

$$f = \arg \max_{k=1..F} \{P_{VLM}(P, PSA_k)\}. \quad (4.3)$$

We summarize the foregoing in the following Theorem, the second major contribution of this work on the PSA:

Theorem 4.2: *Given a sequence $T = T[1..n]$, with symbols from an alphabet σ , where $\sigma = \sigma_1\sigma_2...\sigma_{|\Sigma|}$, and the probabilistic suffix array (PSA) for T , we can decide on whether a pattern $P = P[1..m]$ is generated by the same variable length Markov chain that generated T in $O(m \log \frac{n}{|\Sigma|})$ time.*

Proof: We observe that whenever a mismatch occurs, the start position of the sub-pattern s will be increased by one. The number of mismatches is at most m . Similarly, the number of matching positions will equally be at most m . Thus, the loop in Lines 2-17 will run at most $2m$ times. Line 12 calls the function `LeftmostMatchedPosition` which uses standard suffix array search algorithms [3, 84], that run in $O(\log n)$ time per call. The function `SearchPSA` in Line 13 searches the PSA data structure with the parameters determined in the previous step in $O(\log M)$ time, where M is the length of the PSA. Thus, the total running time will be in $O(m \log n)$. We improve this time to $O(m \log \frac{n}{|\Sigma|})$ using an extra $|\Sigma|$ space to record the starting

position of each symbol in the suffix array. $\square$

Algorithm 4.7: Prediction with VLMM via the PSA

VLMM-PREDICTION(*Pattern*, *PSA*)

```

1    $s \leftarrow 1, \text{Prob} \leftarrow 1, L \leftarrow 1, R \leftarrow n, \text{index} \leftarrow 0, m \leftarrow \|\text{Pattern}\|$ 
2   for ( $i \leftarrow 1$  to  $m$ ) do
3       Search  $\text{Pattern}[s \dots i]$  in  $\text{PSA.SA}$  with parameter  $L, R$ 
4       if mismatch do
5            $\text{index} \leftarrow \text{PSA.index.sLink}$ 
6            $L \leftarrow \text{PSA.index.Start}, R \leftarrow \text{PSA.index.End}$ 
7            $s \leftarrow s + 1$ 
8           if  $i \leq s - 1$  do
9                $i \leftarrow i + 1$ 
10          end if
11      else
12           $\text{position} \leftarrow \text{LeftmostMatchedPosition}(\text{Pattern}[s \dots i], \text{PSA.SA}, L, R)$ 
13           $\text{index} \leftarrow \text{SearchPSA}(\text{PSA}, \text{position}, i - s + 1)$ 
14           $\text{Prob} \leftarrow \text{Prob} \times \text{PSA.index.Conditional.Probability}$ 
15           $L \leftarrow \text{PSA.index.Start}, R \leftarrow \text{PSA.index.End}$ 
16      end if
17  end for
18  return Prob

```

4.5 Space Analysis

In this section, we analyze the space requirement for the PSA structure. We consider space required during its construction (work space), and for its storage and use.

4.5.1 Storage Space

The basic PSA structure has four types of attributes. The measurement attributes are dependent on the application, and may not be needed every time the PSA is used. We also observe that, though the TF is needed for computing the empirical probabilities required for the later stage of determining the conditional probabilities, we do not need to store the TF s directly.

They can be obtained easily using the $\langle Start, End \rangle$ pairs stored at each node. Thus, we focus on the other attributes. From Table 4.1, we see that the worst case space for the PSA structure is $6n$ integers plus n characters, or $25n$ bytes, assuming $n \leq 2^{32}$, and $|\Sigma| = 256$. On average, we could save at least n integers by storing the $\langle Start, End \rangle$ pair as $\langle Start, (End - Start) \rangle$, and the fact that the maximum value in the *Length* array will be the maximum LCP value for the sequence, which is known to be of length in $O(\log_{|\Sigma|} n)$ [58].

Further, with $M = \text{PSA length (number of PSA nodes or intervals)}$, the space for the internal node attributes and suffix link attributes will be $4M$ integers. In this case, M is a sub-linear function of n . Thus, the ratio $\gamma = \frac{M}{n}$, where $0 \leq \gamma \leq 1$ is an important measure on the node branching structure of the PSA, and hence the complexity of the original sequence. With larger γ , we need more practical space to store the PSA. At $\gamma < 0.7$, which is our observation for example sequences tested (See Table 3.3), this will result in an average space requirement of $(5n + 12n * \gamma)$, or $13.4n$ bytes for the PSA.

4.5.2 Construction Space

There are five steps to build the PSA and compute the VLMM probabilities. We list the work space in each step and analyze the worst case space over all the construction steps.

1. Building Interval Array and Computing TF, DF

The space requirement will depend on the following:

- (a) Input: Text of size n , suffix array (n integers) and LCP Array (n integers).
- (b) Work space (in integers): $Stack(2n)$, $DF(n)$, $doclink(Z)$ and $docsp(Z)$, where Z is the number of documents.
- (c) Output: interval array with $\langle Start, End \rangle$ and *Length*, n integers each.

The total space will be $(7n + 2Z)$ integers plus n characters for the text.

Memory reuse: The interval $\langle Start, End \rangle$ and *Stack* can share the same space. *Stack* is the space which stores the yet-to-be-computed non-trivial lcp-delimited intervals. In the extreme case, the length of *Stack* is at most n , since there are at most n non-trivial

lcp-delimited intervals. So when a non-trivial lcp-delimited interval has been processed, it will be stored in the interval $\langle Start, End \rangle$. The length of the interval $\langle Start, End \rangle$ is at most n . Thus, the interval $\langle Start, End \rangle$ records the calculated non-trivial lcp-delimited intervals. However, the sum of length of the interval $\langle Start, End \rangle$ and $Stack$ is at most n . Thus, the interval $\langle Start, End \rangle$ and $Stack$ can share the same space. The maximum space required will then be $(5n + 2Z)$ integers plus n characters.

2. Building the Interval Tree

The space requirement will depend on the following:

- (a) Input: Text of size n , suffix array, interval array $\langle Start, End \rangle$ and $Length$.
- (b) Work space and output (in integers): $posInterval(n)$, $parent(n)$, $pN(n)$ and $Stack(n)$.

The total space needed for this step is $8n$ integers plus n characters.

Memory reuse: The arrays $Stack$ and pN can share the same n integer space, since pN is a stack that indicates the nodes have been processed, while $Stack$ is the space that indicates those not yet processed. These are mutually exclusive, with a combined length n . The maximum space will be $7n$ integer plus n char.

3. Calculating Suffix Links

The space requirement will depend on the following:

- (a) Input : Text of size n ; $1n$ integer array each for, SA , $Start$, $Length$, $posInterval$, and $parent$.
- (b) Work space and output (in integers): inverse suffix array $W(n)$ and suffix link(n).

The total space for this step is $7n$ integer plus n characters.

Memory reuse: Suffix link and $posInterval$ share the same n integer space. In the algorithm `BUILDSUFFIXLINK`, after line 7, the array $posInterval$ is never used again, and the space could be reused for the suffix link. The maximum space is thus $6n$ integer plus n characters.

4. Sort PSA

The space requirement will depend on the following:

- (a) Input: Text of size n ; $1n$ integer array each for *Start*, *Length*, and *SuffixLink*.
- (b) Work space and output: 8 counting arrays (n integer each), namely: Count, Order, wOrder, NewStart, NewEnd, NewLength, NewsLink, NewPattern.

Memory reuse: At most two additional arrays are need at any given time. Thus, the maximum work space is $5n$ integers plus n characters.

5. Computing the Conditional Probabilities

This is an in place algorithm, requiring no extra space.

From above analysis, the maximum space required during PSA construction will be $7n$ integers plus n characters, where we have assumed that $n \gg Z$. We can use an argument similar to that made for the storage space to save n integers, and applying the γ ratio to get an average case construction space requirement of $5.15n$ integers, or $20.6n$ bytes.

The above PSA space requirement can be compared with the space requirement using the PST. The use of reverse suffix links [9] imply that the PST will require at least $37n$ bytes on average (assuming Ukkonen's suffix tree construction algorithm), without counting other auxiliary structures needed for the PST construction.

4.6 Experiments

We performed experiments on protein sequences to test the proposed data structure. The experiments were performed using a DELL PC, with $4 \times 2.67\text{GHz}$ CPU, and 8G memory, running Ubuntu 10.10 Linux operating system. All programs were compiled using **gcc**.

4.6.1 Predicting Protein Families

Our major objective was to develop a time- and space-efficient alternative to the PST. However, to place our results in the correct context, we must first verify that the proposed PSA pro-

duces an equivalent performance in protein sequence modeling and prediction, when compared with the original PST. Thus, to be able to compare the PSA results with previous approaches using the PST [17, 71], we used the same protein sequence dataset that was used in [17] and in [71]. We downloaded the Pfam database [15, 39], release 1.0. The database contains 175 families originally derived from the SWISSPROT 33. We use family members from the unaligned sequences to generate the PSA, one PSA for each family. To test the performance in modeling the protein sequences and in predicting the family for unknown sequences, we used leave-one-out cross validation. For each sequence in the SWISSPROT 33 database, we compute the P_{VLM} using the PSA structure for each family. We then assign the protein sequence to the family with the maximum probability. When the maximum probability is obtained using the model of the correct family (whose PSA is generated without the test sequence), we say we have a correct classification (true positive), otherwise, there is a classification error. This simple approach avoids the difficult problem of setting thresholds for correct classification.

Table 4.4 shows the classification performance using the PSA. For comparison, we have also included the results obtained using the PST on the same dataset, as reported in [17]. Table 4.5 shows the summary classification performance. As expected, both the PST and the PSA produce comparable results with respect to modeling and classification of protein families. The PST had an average true positive rate of 90.8%, while the PSA had 90.2%. We note the significant differences in the family sizes for the PST and PSA results. Although we used the same general Pfam dataset, there has been various additions to the Pfam database since the original publication of the PST results. On average, the size of the families in our dataset was 128.21, while the size used for the PST was 97.82. The total size of the current dataset used for the PSA was more than 1500 sequences larger than that of PST (6539 versus 4891). As can be seen from Table 4.4, most of the families where the PST performed significantly better than the PSA could be due to this difference in family sizes (see for example, **ank**, **C2**, **efhand**).

One advantage of performing classification using Eqn. (4.3), is that, when there is an error in the classification, we can consider the family with the next highest predicted probability. For instance, we can consider the the top- k classification rate, which shows the probability of finding the correct protein family within the first k families, as ordered based on their generated probabilities, using the test sequence. Fig. 4.3 shows the classification performance of the PSA on some sample families, using the top- k classification rate. We can observe how the

classification performance rapidly approaches 100% after the first few k values.

Figure 4.3. Top- k classification rate for sample protein families using the PSA.

4.6.2 Space Consideration

A major problem with suffix trees is their practical memory space requirement. Although they have the same theoretical linear space requirement as suffix arrays, in practice, suffix trees consume much more space [3, 46]. This was our primary motivation for developing the PSA. Table 4.6 shows the summary data on the protein families in Pfam used in our experiments. Table 4.7 compares the memory space required to construct the PST [17] and SPST [71]) with that required for the PSA.

We have included results for PST-20, PST-FULL, and SPST [71]. PST-FULL corresponds to the complete PST with no pruning, i.e. with the full string depth for each leaf node. PST-20 corresponds to orde-20 PST, i.e. PST with a maximum string depth of 20 symbols. This was the variant used in [17]. The SPST proposed in [71] also involved some pruning of the suffixes. First, we can observe the nature of the protein sequences (Table 4.6). The maximum branching factor ($\gamma = \frac{M}{N}$) observed was 0.75, while the minimum was 0.46. The average was 0.62. As a key performance measure, we used the *memory consumption factor* (MC Factor), defined as the ratio of the required memory to the total sequence length (N) of the family. We compared

the MC Factor for PSA, PST, and SPST. The table shows that the PSA ratio was steady at about $33.16N$ bytes. The maximum memory required by the PSA for any of the families was $53.46N$ bytes. This can be compared with the (mean and maximum) memory needed for PST-20 ($41.67N, 222.12N$), PST-FULL ($167.47N, 1216N$) and SPST ($67.17N, 111.53N$). Fig. 4.4 shows more detailed information on the memory consumption needed to construct the data structures, using the Pfam protein families. Perhaps, more significantly, while the PSA memory is relatively constant independent of the sequence or family, we can observe the huge fluctuation in the memory needed for the PST and SPST, as captured by the range and standard deviation on the memory consumption factor.

Figure 4.4. Memory consumption factor (MC Factor) needed to construct the PSA and PST data structures for the first 51 protein families in Pfam.

4.6.3 Computational Time Requirement

Table 4.8 shows the summary of the time required for constructing the PSA and the PST data structures. Table 4.9 shows the corresponding summary of the time needed for prediction using the models. The tables show that for prediction, on average, the PSA is about 2.5 times faster than PST-20. The PSA was much faster at the construction stage. For instance, while the PSA was about 3 times faster to build than PST-20, and about 250 times faster than constructing

PST-FULL. The speedup could be related to the fact that the PSA requires a much smaller construction space, and thus less time is spent on moving data in memory.

4.6.4 PSA in Phylogenetic Tree Construction

Ron et al [109] and Bejerano and Yona [17] have enumerated various applications of the PST. As earlier indicated, the PSA is simply a more efficient alternative to the PST. Thus, the PSA can be used anywhere the PST is used. To study this versatility of the PSA/PST further, we considered a new application of the PSA – specifically, its use in the problem of phylogenetic tree construction, using mtRNA or mtDNA sequences. We used the mtDNA sequences from 20 species, namely, human (*Homo sapiens*, V00662), common chimpanzee (*Pan troglodytes*, D38116), pigmy chimpanzee (*Pan paniscus*, D38113), gorilla (*Gorilla gorilla*, D38114), orangutan (*Pongo pygmaeus*, D38115), gibbon (*Hylobates lar*, X99256), baboon (*Papio hamadryas*, Y18001), horse (*Equus caballus*, X79547), white rhinoceros (*Ceratotherium simum*, Y07726), harbor seal (*Phoca vitulina*, X63726), gray seal (*Halichoerus grypus*, X72004), cat (*Felis catus*, U20753), fin whale (*Balenoptera physalus*, X61145), blue whale (*Balenoptera musculus*, X72204), cow (*Bos taurus*, V00654), rat (*Rattus norvegicus*, X14848), mouse (*Mus musculus*, V00711), opossum (*Didelphis virginiana*, Z29573), wallaroo (*Macropus robustus*, Y10524) and platypus (*Ornithorhynchus anatinus*, X83427). This is the same dataset previously used in constructing phylogenetic trees by Otu et al [99] and Li et al [73]. This is a challenging dataset, and there has been some debate on the position of some of the species [24, 105].

To construct the phylogenetic tree, we use the PSA to compute a dissimilarity measure between every pair of species in the dataset. First we construct the PSA for the mtDNA sequence for each species. Then for a given species, we compute the P_{VMM} , the probability that the given species is generated by the PSA constructed from each of the other species. After getting the probabilities, we use the quantity $\lambda_{(A,B)} = -\log P_{VMM}(A, PSA_B)$ as the dissimilarity measure between the sequences from two species A and B , where $P_{VMM}(A, PSA_B)$ is the predicted probability that sequence A is generated by the model represented by the PSA of sequence B . We then use the measurements $\lambda_{(A,B)}$ for all pairs of species to construct the phylogenetic tree.

Fig. 4.5 shows the constructed phylogenetic tree using the PSA. The results are generally in agreement with earlier work on this dataset (see [73, 99] for example). The only major difference

is in the placement of cow, which is supposed to be closer to bluewhale and finwhale than to mouse and rat. This is a very encouraging result, especially, given that the PSA approach is completely alignment free. We believe a more detailed study of the use of PSA in phylogentic tree construction (for instance, deriving other features or measures of similarity) could further improve the results.

Figure 4.5. Phylogenetic tree for 20 species constructed using the predicted probabilities obtained using the PSA.

4.7 Summary

We have presented the probabilistic suffix array (PSA), a data structure for representing information in variable length Markov models. The PSA provides the same functionality as the probabilistic suffix tree (PST), but at a significantly reduced time and space requirement. Given a sequence of length N , construction and learning in the PSA is done in $O(N)$ time and space, independent of the Markov order. Prediction using the PSA is performed in $O(m \log \frac{N}{|\Sigma|})$ time, where m is the pattern length, and Σ is the symbol alphabet. The specific memory requirement for PSA construction is $33N$ bytes in the worst case, and $26N$ bytes on average, including space

for the suffix array and the input sequence. This can be compared with the $41N$ bytes needed using the PST.

We have shown experiments in computational biology. The first experiment compares PSA with PST [17] and SPST [71] on the same data set. The space for PSA is more efficient than PST-FULL and SPST. The space of PSA is close to the PSA-20 which only stores $L=20$ depth path. The construction time of PSA is significant fast than PST and SPST and the prediction time of these three methods (PSA, PST and SPST) is similar. The other experiment is Phylogenetic Tree Construction by PSA. This experiment shows a very encouraging result, especially, given that the PSA approach is completely alignment free.

Table 4.4. Performance of the PSA in modeling and prediction of protein families. Families correspond to the first 51 protein families with 12 or more members in the Pfam database, ordered alphabetically based on their abbreviated names in Pfam. For comparison, we have included the results obtained using the PST [17] on the same data set. (TP stands for true positive, while MD stands for missed detection). **The family apple was not in the dataset used in [17].

Family	Size	# MD by PSA	TP (%) by PSA	Size	# MD by PST	TP (%) by PST
7tm_1	530	26	0.951	515	36	0.930
7tm_2	36	2	0.944	36	2	0.944
7tm_3	12	1	0.917	12	2	0.833
AAA	79	9	0.886	66	8	0.879
ABC_tran	330	46	0.861	269	44	0.836
actin	160	22	0.863	142	4	0.972
adh_short	186	48	0.742	180	20	0.889
adh_zinc	129	22	0.829	129	6	0.953
alddedh	69	11	0.841	69	9	0.870
alpha-amylase	114	0	1.000	114	14	0.877
aminotran	63	14	0.778	63	7	0.889
ank	305	60	0.803	83	10	0.880
apple**	16	1	0.938			
arf	43	2	0.953	43	4	0.907
asp	72	5	0.931	72	12	0.833
ATP-synt_A	79	1	0.987	79	6	0.924
ATP-synt_ab	183	1	0.995	180	6	0.967
ATP-synt_C	62	1	0.984	62	5	0.919
beta-lactamase	51	2	0.961	51	7	0.863
bZIP	95	14	0.853	95	10	0.895
C2	101	21	0.792	78	6	0.923
cadherin	168	12	0.929	31	4	0.871
cellulase	40	3	0.925	40	6	0.850
cNMP_binding	69	4	0.942	42	3	0.929
COesterase	62	3	0.952	61	5	0.918
connexin	40	3	0.925	40	1	0.975
copper-bind	61	1	0.984	61	3	0.951
COX1	80	4	0.950	80	13	0.838
COX2	114	10	0.912	109	2	0.982
cpn10	58	1	0.983	57	4	0.930
cpn60	84	1	0.988	84	5	0.940
crystall	103	6	0.942	53	1	0.981
cyclin	80	19	0.763	80	9	0.888
Cys-protease	95	1	0.989	91	11	0.879
cystatin	88	11	0.875	53	4	0.925
Cy_knot	61	6	0.902	61	4	0.934
cytochrome_b_C	133	4	0.970	130	27	0.792
cytochrome_b_N	170	4	0.976	170	3	0.982
cytochrome_c	175	10	0.943	175	11	0.937
DAG_PE-bind	108	5	0.954	68	7	0.897
DNA_methylase	57	2	0.965	48	8	0.833
DNA_pol	51	12	0.765	46	9	0.804
dsrm	22	14	0.364	14	2	0.857
E1-E2_ATPase	117	2	0.983	102	7	0.931
efhand	739	96	0.870	320	25	0.922
EGF	676	33	0.951	169	18	0.893
enolase	41	2	0.951	40	0	1.000

Table 4.5. Summary performance in protein family classification using the PSA and PST

Family	Size	# MD by PSA	TP (%) by PSA	Size	# MD by PST	TP (%) by PST
Mean	128.216	12.373	0.902	97.820	8.720	0.908
Std	145.798	17.711	0.103	84.778	8.690	0.050
Min	12.000	0.000	0.364	12.000	0.000	0.792
Max	739.000	96.000	1.000	515.000	44.000	1.000
Total	6539	631		4891	436	

Table 4.6. Summary data on the first 51 families in Pfam, as described in Table 4.4.

Protein	Family Size	Length (N)	# of Internal nodes (M)	$\gamma = \frac{M}{N}$	Min LCP	Max LCP
Mean	128.216	22173.451	252.353	0.616	190.608	14.659
Std	145.798	23359.451	136.862	0.079	139.531	9.198
Min	12.000	1360.000	68.000	0.460	26.000	3.987
Max	739.000	140744.000	753.000	0.750	719.000	49.912

Table 4.7. Construction memory needed for the PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4.

	PSA	MC Factor	PST-20	MC Factor	PST- FULL	MC Factor	SPST	MC Factor
Mean	616	33.16	425	41.67	1577	167.47	1068	67.17
Std	655	8.23	116	42.68	125	211.22	541	23.77
Min	71	16.74	263	4.33	1455	12.77	79	18.53
Max	4523	53.46	1023	222.12	2235	1216	2547	111.53

Table 4.8. Construction time comparison for PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4. Recorded time is time needed per family (in seconds). Speedup is computed as the ratio with respect to PSA time.

	PSA	PST-20	Speedup	PST-FULL	Speedup	SPST	Speedup
Mean	0.092	0.244	3.150	12.648	250.967	1.825	1.740
Std	0.106	0.231	1.297	9.984	238.806	2.701	2.617
Min	0.005	0.016	0.854	3.780	20.917	0.047	0.036
Max	0.545	1.376	6.895	56.540	1076.000	17.121	16.658

Table 4.9. Prediction time comparison between PSA and PST. Results are based on the first 51 families in Pfam, as described in Table 4.4. Recorded time (in seconds) is prediction time per family – i.e. total time needed to predict all members in the family against all the other families.

	PSA	PST-20	Speedup	PST-FULL	Speedup	SPST	Speedup
Mean	87.71	98.83	2.52	113.25	2.79	207.03	3.4
Std	93.89	66.4	3.28	74.47	3.46	130.93	2.39
Min	5.00	59.23	0.25	21.47	0.32	33.50	1.03
Max	576	528	20.13	579.3	18.86	600	17.6

Chapter 5

Circular Pattern Matching

5.1 Introduction

The CPM problem was first introduced in 1980 [19]. Since then, variations of the circular pattern matching problem has been studied. The first variant [46, 115] is the exact circular pattern matching (ECPM) problem. This problem is to find all occurrences of a circular pattern P in a text T without any error. The second variant [44, 81] is the approximate circular pattern matching problem. This problem allows some error between the circular pattern P and the text T . According to the definitions in Chapter 2 on related work, most approaches have focused on the existential query for the *ACPM1* problem.

The *ACPM2* problem is given text T , circular pattern P and maximum error k , return all positions where the circular string $[P]$ match to text T with at most k errors. $[P]$ is said to be a k -approximate match with text T at position $j \in [1 \dots n - m - k + 1]$ if $ED_c(P, T[j \dots j + m]) \leq k$, where $0 \leq t \leq m - 1, -k \leq n - m$. This is clearly more difficult than the ECPM problem or the *ACPM1* problem for existential and counting queries.

Main Results. Our main goal is to solve the ECPM and *ACPM2* problems for existential queries which were introduced under related work. We then define and solve the ECPD and ACPD problems to find “interesting” circular patterns, as defined using specified constraints.

In this chapter, we present a new algorithm to solve the ECPM problem. This algorithm runs in linear time and space complexity. To date this is the best ECPM algorithm with respect to time and space complexity. We also present four algorithms to solve the ACPM2 problem. Three of the algorithms report complete results and one is a greedy(suboptimal) algorithm reporting an incomplete result. We compare our algorithms with other algorithms in the literature such as those reported by Maes [81], Gregor [44], Uliel [123] which were introduced in related work. On average, our ACPM2 algorithm provides the best result for the ACPM2 problem with respect to time and space complexity.

The following three theorems represent our main contributions on the CPM problem.

Theorem 5.1: *Given a text $T = T[1..n]$ and a circular pattern $P = P[1..m]$, with symbols from an alphabet Σ , the ECPM algorithm can solve the ECPM problem in $O(m \log |\Sigma|)$ worst case time after constructing the suffix tree and suffix links in $O(n)$ time and space complexity.*

Theorem 5.2: *Given a text $T = T[1..n]$ and a circular pattern $P = P[1..m]$, with symbols from an alphabet Σ , Algorithm ACPM2 solves the ACPM2 problem in $O(km^2n)$ time, and $O(n)$ space.*

Theorem 5.3: *Given a database sequences $SeqDB$, with Z sequences and N total symbols ($|\Sigma|$ could be $O(N)$), Algorithm ACPM2 solves the all-against-all ACPM problem in $O(km_a N)$ time cost on average, and $O(km_m N^2)$ time worst case, using $O(N)$ worst case space, where $m_a = \frac{N}{Z}$, m_m is the length of the longest sequence in $SeqDB$.*

Organization. In the next section, we present a linear-time linear-space algorithm for the ECPM problem. Algorithms for the ACPM2 problem are presented and analyzed in Section 3. In Section 4, we show experiments on analyzing circular permutations in multidomain proteins using our algorithms. Based on the results, we perform protein function prediction for multidomain proteins. In Section 5, we summarize our work on circular pattern matching problems.

5.2 Exact Circular Pattern Matching Problem

In this section, we present an algorithm to solve the ECPM problem in linear time and linear space. Our method is index-based and can be built on the space-efficient *virtual suffix tree* proposed earlier in Chapter 3. The key to our method is the suffix link provided by the suffix tree and the VST. To our knowledge, this is the first time that the suffix link has been exploited to solve the circular pattern matching problem.

Notation. Before describing the algorithm, we assume there is a sequence database *SeqDB* with Z sequences. The total number of symbols in *SeqDB* is N . Let $SeqDB[i]$ be the i -th sequence in *SeqDB*, where $0 < i \leq Z$. Let m_i be the length of $SeqDB[i]$. The average number of symbols per sequence in *SeqDB* is $m_a = \frac{N}{Z}$. Let k be the allowed error in a match. In this database *SeqDB*, the alphabet size is $|\Sigma|$.

5.2.1 Linear Time ECPM Algorithm

The index-based approach was first introduced by Iliopoulos and Rahman [51]. However, their methods were quite complex and difficult to implement. The best time complexity of Iliopoulos and Rahman's algorithms is $O(N \log N)$. Our algorithm is relatively simple with lower time and space complexity. First we give an example of ECPM and use this to explain our algorithm. Then we present a formal algorithm for the ECPM problem with input pattern P and ST , the suffix tree constructed from T . We build this algorithm to solve the all-against-all version of the ECPM problem, given a database of sequences.

Examples. In chapter 3, we showed the suffix tree for the string $T = \text{missississippis}$. We utilize this example again here. Figure 5.1 shows the suffix tree for T with a few suffix links. Let circular pattern $P = \text{iss}$, so $[P] = \{\text{iss}, \text{ssi}, \text{sis}\}$, where $f^0(P) = \text{iss}$, $f^1(P) = \text{ssi}$, $f^2(P) = \text{sis}$. $f^{i+1}(P)$ is obtained by removing the first symbol from $f^i(P)$ and then appending this symbol at the end, where $0 \leq i < m - 1$. So in our algorithm, when we match $f^i(P)$, we use the suffix link to find the first $(m - 1)$ symbols of $f^{i+1}(P)$. This operation is done in constant time. Then we only need to compare the last symbol of $f^{i+1}(P)$ to the corresponding position in the suffix tree and if the symbols match, we report a match for $f^{i+1}(P)$.

In our algorithm, we use the suffix link to match the circular pattern in an incremental manner. For example, we search $f^0(P) = iss$ first. In this case, we find iss is in the edge between node N_1 and node N_4 , thus there is a match. Then match for $f^1(P) = ssi$ by following the suffix link from node N_4 to node N_6 . Thus we get the next matching edge in internal node N_6 by using the “skip/count” method [46]. The time cost of this operation is constant time. To look for the matches to $f^2(P) = sis$, we use the suffix link from node N_6 to node N_5 . The prefix si has matched up to node N_5 , thus we only check the outgoing edges from node N_5 to its children, and select the one whose first symbol matches symbol s . We find the edge between node N_5 and N_8 is matched by symbol s . So all of circular matchings of $[P]$ have been found whose positions start from the leaf nodes of N_4, N_6 and N_5 respectively. From position 2 to position 9 in T , there are nine substrings that matched the circular pattern $P = iss$.

We give another more complex example for searching $f^0(P) = ism$. First, we find the prefix is in the edge between node N_1 and node N_4 , but there is a mismatch at the last symbol m . To search for $f^1(P) = smi$, we follow the suffix link from node N_4 to node N_6 . The previous iteration only matched two symbols, so this iteration start from the second symbol of $f^1(P) = smi$ which is m . However, the length of path from *root* to N_6 is 3, which is larger than 1, so this searching starts from node N_3 which is the parent node of N_6 . Then we check the second symbol, but it is still a mismatch. Thus we know that $f^1(P) = smi$ does not occur in T . Now to search for $f^2(P) = mis$, we again follow the suffix link from node N_3 to the *root*, and continue the match from the *root*. Thus, we find a match of $f^2(P) = mis$ on the path to leaf node 6. Therefore, the system will report one occurrence in leaf node 6 of the suffix tree, which corresponds to position 1 in T .

ECPM Algorithm Description

Algorithm ECPM (Algorithm 5.1) shows the pseudo code for our exact circular pattern matching algorithm. In this algorithm, the input ST is the suffix tree for the text T . T denotes a given text (string, or sequence) to be searched. For a given pattern P with length of m , the algorithm derives a new pattern, called PP , PP is derived from Pattern P which repeats the Pattern P and removes the last character of P . Thus is, $PP = P[1..m] \circ P[1..m-1]$. Therefore, the new pattern PP has a length of $2m - 1$.

In the suffix tree, ST , the algorithm searches PP starting from the root of ST from the leftmost branch to the rightmost. The iteration variable i indicates the starting position in PP .

The variable len indicates the position of node label. When len is equal to 1, a new child node is created for the current node. Line 4 in the algorithm represents the process of finding the right child for the current node. This operation requires $O(\log(|\Sigma|))$ time for comparison. If len is larger than 1, the matching operation takes place in the current node.

The variable top is the pointer which points to the current circular pattern. If the length of the matched pattern $PP[top...i]$ is m , it means that a circular pattern occurs. Since the number of possible circular patterns is m , pointer top will never be larger than m . If top pointer is larger than i , it indicates that the symbol $pattern[top]$ character never occurred in the text. Thus, this set of circular patterns cannot be in the text. The pointer top is increased by one in the following two cases.

The first case is when a mismatch occurs in the current node. In this situation, the current node is replaced by the node which is pointed to by its suffix link. At the same time the string depth for the current node decreases by one. Pointer top increases by one. After this, the next iteration of the string matching process starts. The other case is when the path length from the root to the current node is m . It indicates one of the circular patterns occurs inside $Text$. Before matching the next position of PP , pointer top increases by one to keep the length of the pattern equal to m .

Based on the ECPM algorithm, we develop an algorithm to solve the all-against-all problem. That is, we compute $ECPM(SeqDB[i], SeqDB[j]), \forall i, j, i \neq j$. Algorithm ALL-VS-ALL-ECPM (Algorithm 5.2) enumerates each sequence in the database as a pattern to search the circular relation by using the former ECPM algorithm. It constructs the suffix tree ST for the entire database first. Then let each sequence be a pattern to search in ST .

Algorithm Analysis

In the ECPM algorithm (Algorithm 5.1), when the same position in PP is compared again (Line 18), pointer top increases by one. top remains less than or equal to m . Thus, Line 18 runs at

most m times.

The “for” loop from Line 3 to Line 23 runs at most $2m - 1$ times. Inside this loop, the cost of Line 4 is at most $O(\log |\Sigma|)$. The other lines inside this “for” loop have a constant running time. Therefore, the time cost of this algorithm is at most $\log |\Sigma| \times (3m - 1) = O(m \log |\Sigma|)$. Given the relation between the ST and VST as described in Chapter 3, the proposed algorithm can easily be modified to use the VST.

The space cost of implementation using the suffix tree is $32n + 2m - 1$, where $m \leq n$. The space cost if we implement use the VST is $13.8n + 2m - 1$.

We summarize the above in Theorem 5.1.

Theorem 5.1: *Given a text $T = T[1..n]$ and a circular pattern $P = P[1..m]$, with symbols from an alphabet Σ , the ECPM algorithm can solve the ECPM problem in $O(m \log |\Sigma|)$ worst case time after constructing the suffix tree and suffix links in $O(n)$ time and space complexity.*

Algorithm ALL-VS-ALL-ECPM (Algorithm 5.2) builds the ST in $O(N)$ time cost. Time cost of line 4 is $O(m \log |\Sigma|)$, so the total cost of the loop from line 2 to line 5 is $O(N)$. Worst case time complexity of this algorithm is $O(N \log |\Sigma|)$. Space complexity of this algorithm is $O(N)$.

5.2.2 Comparison of ECPM algorithms

Table 5.1 shows the comparison of our ECPM algorithm with Iliopoulos and Rahman’s algorithms [51] CPI-I, CPI-II. The table shows that the ECPM algorithm is the best algorithm with respect to time complexity, even when $|\Sigma| \rightarrow O(N)$. In terms of space, the ECPM algorithm has the same time complexity with CPI-II algorithm which was implemented using the suffix array. The CPI-II algorithm implemented using the compressed suffix array has the best space complexity among these algorithms, but the time complexity is not as good. We note that for practical space complexity, the CPI-I need two suffix trees and various auxiliary arrays, and hence will require a significantly larger practical space.

5.3 Approximate Circular Pattern Matching Problem

In this section, we present our algorithms for the ACPM problem. We start with a simple greedy algorithm and then consider a suffix-array based q -gram algorithm for the ACPM problem. First, we introduce a basic LIS algorithm APM-VIA-LIS (Algorithm 5.3) to find an approximate match of a pattern P in text T . The algorithm does not handle circular pattern matching. Next, we propose our algorithms for the ACPM problem and analyze their complexity. The LIS method for pattern matching will be used in these algorithms. When we use this algorithm to solve the ACPM problem, we have to use all circular shifts $f^t(P)$ to match the text T .

The LIS method utilizes the LIS algorithm [46, 50] to calculate the longest common subsequence (LCS) [31, 46] between two sequences. The verification process is to verify whether the edit distance between these two sequences is less than k . When we calculate LIS and LCS, each matched symbol will occur in the LCS. We are able to get occurring positions in two sequences for the matched symbols. We can use these positions to check the number of edit operations between two matched symbols. Thus the algorithm reports the edit distance between these two sequences. The time complexity for this algorithm is $O(\frac{mn}{|\Sigma|} \log m)$. When $|\Sigma|$ is close to $O(m)$, as in the case for multidomain proteins, the time complexity will be $O(n \log m)$.

5.3.1 Greedy ACPM Algorithm

Algorithm ACPM-GREEDY (Algorithm 5.4) compares any two sequences with one as text and the other one as circular pattern in two main steps. The first step is the generation of LCS. The second step will verify whether the LCS generated in step 1 represents a part of a valid subsequence. These two steps were presented in Algorithm APM-VIA-LIS (Algorithm 5.3).

Table 5.1. Comparison of ECPM algorithms

	ECPM algorithm	CPI-I [51]	CPI-II [51]	CPI-II(Compressed Suffix Array) [51]
Time Complexity	$O(N \log \Sigma)$	$O(N \log^{1+\epsilon} N + N \log \log N)$	$O(N \log N)$	$O(N \log N \log N)$
Space Complexity	$O(N) \text{ bytes}$	$O(N \log^{1+\epsilon} N) \text{ bytes}$	$O(N) \text{ bytes}$	$O(N \log N) \text{ bits}$

First, Algorithm ACPM-GREEDY (Algorithm 5.4) will choose two sequences, one as text and the other one as circular pattern. After getting text T and circular pattern P , the ACPM works on the following two steps. The first step creates a new pattern PP by concatenation of P . And then the second step calculates the LCS between PP and T and returns the LCS string lcs . This procedure is performed in line 5. This step also verifies the approximate pattern matching with parameter k .

This method is greedy(suboptimal): it finds only one occurrence of the pattern, it may not detect all the existing circular patterns in the text. If there is more than one LCS in T , this method may miss some matches.

Time Complexity Analysis

For the time complexity analysis, we need to consider three cases.

1. For the case of using one sequence as pattern P and the other sequence as text T , the time complexity of getting LCS (line 5) is $O(\frac{mm}{|\Sigma|} \log m)$. When $|\Sigma|$ is close to $O(m)$ as in mutildomain proteins, the time complexity will be $O(n \log m)$.
2. For the case of searching for one sequence against a group of sequences (loop from line 2 to line 6), the time complexity is $O(\sum_{i=1}^Z n_i \log m) = O(N \log m)$, where N is the total length of all sequences used, $N = \sum_{i=1}^Z n_i$ Z is the number of sequences, and n_i is the length of the i -th sequence in *SeqDB*.
3. For the case of searching for a CP among a group of sequences (loop from line 1 to line 7), the time complexity is $ZN \log m$, where m is the length of the longest sequences. The final time complexity is $O(N^2 \log m)$, since $Z = O(N)$. In our experiment with multidomain proteins, $N \approx 6Z$

5.3.2 ACPM with LIS

Here we present a second algorithm (Algorithm 5.5) for the ACPM problem. This method also utilizes the LIS algorithm [46, 50] to calculate LCS [31, 46]. However, the method to

construct Pattern P and Text T are changed. More importantly, unlike the greedy algorithm described earlier, this algorithm can detect all the circular patterns. The pseudo code is listed in Algorithm ACPM-LIS . One sequence is used to construct the circular pattern P . Another sequence is used as text T . All possible circular shifts of the pattern are enumerated. The $subT$ is extracted from sequence T by a sliding window with size of $m + k$. Finally, each enumerated circular shift of the pattern is searched against the sliding window separately. Assuming the sequence to be researched using a sliding window has the length n , then, there are $\max\{1, n - (m + k) + 1\} = O(n)$ windows to be constructed.

During the searching process, if there is a common subsequence with length $m-k$ is found, then, there is a circular pattern occurring in Text T . This method reveals all the circular patterns in each sequence. Thus it can find the optimal solution, with respect to completeness of the results.

Time Complexity Analysis

For one sliding window, the time complexity of finding one circular pattern is $O(\frac{m(m+k)}{|\Sigma|} \log m)$ (line 7 to line 9). There are $O(n)$ sliding windows and $O(m)$ circular patterns inside one query pattern and one text (line 5 to line 10). Therefore, the time complexity of detecting a circular pattern between one pattern and one text is $O(\frac{m(m+k)}{|\Sigma|} \log m \times mn)$. Examining these terms, we can find that k is at most $O(m)$ and $|\Sigma| = O(m)$. Thus, the time complexity can be abbreviated as $O(m \log m \times mn) = O(m^2 n \log m)$.

For each pattern, the algorithm compares with the other sequences (line 2 to line 11). The time complexity of each pattern comparing with all other sequences is $O(m^2 \log m \times \sum_{i=1}^Z n_i) = O(m^2 N \log m)$, where N is the total length of all sequences and n_i is the length of i -th sequence in *SeqDB*.

After considering all patterns (line 1 to line 12), the time complexity becomes $\sum_{i=1}^Z m_i^2 N \log m$, where m is the length of the longest sequence. In fact, $\sum_{i=1}^Z m_i^2 \leq (\sum_{i=1}^Z m_i)^2 = N^2$. The final time complexity is therefore $O(N^3 \log m)$. This is the worst case complexity.

5.3.3 ACPM with q -grams and Suffix Array

The q -gram approach [3] is a two-phase method to reveal all approximate patterns. The first phase is the *Hypothesis Phase* which determines all potential matching positions using only q -gram substrings of P and T . In the second phase, the *Verification Phase*, the algorithm verifies each potential matching position to report the correct matches. First we introduce an ACPM algorithm with q -grams, then we present a hybrid algorithm with ECPM and ACPM with q -grams. The latter algorithm is more time efficient in practice, but the theoretical time complexity is the same as the ACPM algorithm with q -grams.

Figure 5.2 shows the number of hypotheses with different q values in the ProDom database of multidomain protein sequences [32]. Here we used $N = 10^6$. We notice that when q increases, the number of hypotheses will decrease fast. So when q is not very small, e.g $q \geq 3$, the number hypotheses will typically reduce to $O(N)$.

Algorithm Description

Algorithm ACPM-QGRAM (Algorithm 5.6) shows the process. Lines 1-7 is the preprocessing stage. This stage constructs a long concatenated sequence, seq , using all the sequences so far encountered in $SeqDB$. It also builds an auxiliary array pos . This array is used to maintain the relationship between position in seq and $SeqDB$. Line 8 constructs the suffix array for the concatenated sequence.

Lines 9-24 is a loop to generate all of the hypotheses for the q -gram method using the LCP array. Line 11-13 determines candidate matching positions that have the same q -gram prefix. Line 14 considers each pair of candidate positions obtained with the current q -gram for verification.

Lines 15-22 is the verification algorithm. We use the LIS algorithm to verify the approximate patterns. Constructing the circular pattern is the same as in the previous algorithm. We enumerate the m circular patterns from a sequence one by one. We construct $subT$ from the second sequence T as follows. Assume the q -gram occurs in position y , so let $subT$ be the substring of T which includes $T[y..y + q - 1]$ and the length is $(m + k)$. So text will be

$T[y - m - k + q \dots y + q - 1]$, $T[y - m - k + q + 1 \dots y + q]$, ... $T[y \dots y + m + k - 1]$. There are $(m + k - q)$ number of such substrings.

Time Complexity Analysis

The time complexity of LIS to verify a pattern vs. substrings of Text which includes one matched q -gram is $O(m \log m \times (m + k - q))$. Since $k \leq O(m)$ and $q \leq O(m)$, the time complexity is $O(m^2 \log m)$. Each pair in the same group has $O(m)$ circular pattern operations, thus the time complexity for verifying each pair is $O(m^2 \log m \times m) = O(m^3 \log m)$. There are r groups and group i has n_i elements and there are $\sum_{i=1}^r n_i^2$ pairs. The total complexity is $O(m^3 \log m \times \sum_{i=1}^r n_i^2)$. The worst case occurs when $r = 1$ with time complexity of $O(N^2 m^3 \log m)$. For the average case, m is the average length of the sequences. Then the time complexity will be in $O(N m_a^3 \log(m_a))$, where is $m_a = \frac{N}{Z}$.

Hybrid Algorithm

We can combine the ECPM algorithm (ECPM) and the ACPM algorithm with q -gram (ACPM-QGRAM) for a possible improvement in practical time. The ECPM algorithm reports the exact circular pattern matches which should not be computed again when we search for approximate matches. First, the hybrid algorithm uses the ECPM algorithm to look for matching exact circular patterns and stores them. Next the hybrid algorithm uses the q -gram method to generate hypothesis. In the verification phase, the algorithm checks whether this hypothesis has occurred before within the ECPM results. If it occurred, then the verification stage is skipped.

Thus this approach will reduce the practical running time, given the reduced verifications. For checking the each hypothesis, it takes $O(\log N)$ time cost. The time complexity is $O(N^2(m^3 \log m + \log N))$. But this algorithm needs $O(N^2)$ space to maintain the pair-wise results from the ECPM algorithm.

5.3.4 Improved Algorithm: ACPM with Bidirectional Edit Distance

In this subsection, we propose an algorithm to solve the all-against-all ACPM2 problem. The algorithm uses a two-stage hypothesis generation – hypothesis verification paradigm. After generating the hypotheses using the q -gram filtration method, the algorithm verifies each hypothesis in $O(km)$ time complexity, where k is the maximum error allowed and m is the length of the pattern. This algorithm follows the same general paradigm as the previous algorithm (Section 5.3.3), however, there are significant differences in both the hypothesis. In practical, this algorithm uses the suffix tree than the suffix array.

Filteration via q -grams

The q -gram approach [53] is a filtration method which is based on the fact that for any two strings that are approximate matches, there must be some exact matching sub-region between them. The problem is how to determine such sub-regions and their length(s). Lemma 5.1 shows this fact and points out how to choose the value q , the minimum length of the matching regions.

Lemma 5.1 [14] : Given a text T , a pattern P of length m , and an integer k , ($0 \leq k < m$), for a k -approximate match of P to occur in T , there must exist at least one q -length block of symbols in P that form an exact match to some q -length substring in T , where $q = \lfloor \frac{m}{k+1} \rfloor$.

Approximate pattern matching based on q -gram uses two phases. The first phase is the hypothesis phase which identifies all potential matches using q -gram filtering operations. Based on partial exact matching, the algorithm can find $O(N^2)$ potential matches. The second phase is the verification phase. Here the hypothesized potential matches from the first phase are verified to determine whether they are *true* k -approximate matches. Our ACPM2 algorithm is based on q -gram filtration. First we generate hypotheses for potential approximate circular pattern matches by q -gram filter operations. Then we verify each hypothesis to find the true circular patterns.

ACPM Hypothesis Generation

Algorithm 5.7 represents our hypothesis generation phase using suffix trees. First, it builds the generalized suffix tree ST for the sequence database. Then, the algorithm treats each sequence as a pattern. For each pattern, q is calculated using $q = \lfloor \frac{m}{k+1} \rfloor$, where m is the length of the current pattern. There exists $O(m)$ overlapping q -grams in a given m -length pattern. We search each overlapping q -gram of the pattern from left to right in the suffix tree ST . Each exact match is a hypothesis with the current q -gram. The use of the suffix tree implies that for each given distinct q -gram in P , all the occurrences in T will be found as the leaf nodes from the same parent node in the suffix tree. Finding this parent node requires only one q -gram match in $O(q \log |\Sigma|)$ time. Similar to the ECPM algorithm, after searching the first q -gram, say $q_i = P[i \dots i + q - 1]$ in ST , the next q -gram $q_{i+1} = P[i + 1 \dots i + q]$ can be matched incrementally from q_i by using suffix links in constant time (Using the function Search-in-ST). Thus, counting all the matching q -grams or locating all the parent nodes for each unique matching q -gram can be done in $O(m \log |\Sigma|)$ time, independent of n or N , where n is the length of the current sequence (the text), and N is the total length of all the sequences in the database (the number of leaf nodes in the generalized suffix tree).

Given pattern $P = P[1..m]$ and text $T = T[1..m]$, there are $O(m)$ q -grams in P and $O(n)$ q -grams in T . So each q -gram of P can produce at most $(n - q + 1)$ exact matches in T . The number of hypotheses is $O(m(n - q + 1)) = O(mn)$. We notice the added difficulty introduced by the ACPM2 problem. Unlike for the ACPM1 problem where only one occurrence of P in T is required, here, we have to verify each of the potential $O(mn)$ hypotheses, in order to identify all the occurrences. In the sequence database $seqDB$, we have the total length of all sequences as: $N = \sum_{i=1}^Z m_i$. Each q -gram can produce potentially $O(N)$ exact matches. Thus, the number of hypotheses will be in $O(N^2)$. In the worst case, there exist $O(|\Sigma|^q)$ unique q -grams. The number of hypotheses will be $O(|\Sigma|^q \times (\frac{N}{|\Sigma|^q})^2) = O(\frac{N^2}{|\Sigma|^q})$ on average. When q increases, the number of hypotheses will decrease exponentially. It will be $O(N)$ when $|\Sigma|$ is $O(N)$.

ACPM Verification Algorithm Description

Our verification algorithm makes use of a novel **bidirectional edit distance** that uses both the direct sequence and its reverse. Before we introduce our ACPM verification algorithm, we first describe some important characteristics of edit distances.

Lemma 5.2 : Suppose there is an exact matching q -gram common to both the text $T = T[1...n]$ and the pattern $P = P[1...m]$. Let (i, j) be the position of occurrence of the q -gram in T and P respectively. If this q -gram is part of a true approximate match between P and T , then the edit distance between T and P is given by $ED(P, T) = ED(P[1...j-1], T[1...i-1]) + ED(P[j+q...m], T[i+q...n])$, where $1 \leq j \leq m, 1 \leq i \leq n, q \geq 1$.

Proof: There exists one q -length block in exact match between the pattern and the text (Figure 5.3). Since this q -gram is involved in a true approximate match between P and T , it must be involved in the edit distance computation between P and T . Thus the optimal edit path must contain the subpath between these regions of the pattern and the text. $P[j...j+q-1]$ only compares with $T[i...i+q-1]$, so the edit cost of the subpath is zero. We don't need to compare $P[1...j-1]$ with $T[i...n]$, nor do we need to compare $P[j...m]$ with $T[1...i-1]$, because the optimal edit paths do not cross these areas. Figure 5.3 illustrates this using a q -gram exact match (the solid line). Three optimal paths (the dashed lines) pass through point (i, j) , but no optimal path can pass through point (i, L) or point (H, j) , where $L > j$ and $H > i$. $P[1...j-1]$ only compares with $T[1...i-1]$ and $P[j+q...m]$ compares with $T[i+q...n]$. Hence the edit distance between pattern P and text T is the sum of three components $ED(P[1...j-1], T[1...i-1])$, $ED(P[j...j+q-1], T[i...i+q-1])$ and $ED(P[j+q...m], T[i+q...n])$. Since $ED(P[j...j+q-1], T[i...i+q-1])=0$, the Lemma holds. $\square$

Lemma 5.3 : For unit cost edit operations, the $ED(P, T) = ED(P^R, T^R)$, where P^R and T^R are reversed version of P and T respectively.

Proof: This Lemma has been proved in the proof of [118] Lemma 2. $\square$

We need to verify each hypothesis generated by Algorithm 5.7. Assume Q_T and Q_P are the q -grams from the text T and the pattern P respectively. Let $Q_T = Q_P = Q$. So there is

an exact matching q -gram between the text and the pattern using this q -gram. We present an $O(km)$ verification algorithm to determine whether the q -gram Q is part of a true k -approximate circular match between P and T .

We now describe the idea of bidirectional edit distance, based on which we verify a given hypothesis. Suppose Q_P is a substring of pattern P which starts at position j and Q_T is a substring of text T which starts at position i , where $0 \leq j \leq m$ and $0 \leq i \leq n$. That is, $Q_P = P[j \dots j + q - 1]$ and $Q_T = T[i \dots i + q - 1]$. Our goal is to compute the circular edit distance between pattern P and the substring of T denoted $subT_i$, where $subT_i = T[i + q - 1 - m - k \dots i + m + k]$. First, we construct two strings from the pattern $P_1 = P[j + q \dots m] \circ P[1 \dots j - 1]$ and $P_2 = P_1^R$. We also construct two strings $T_1 = T[i + q \dots i + m + k]$ and $T_2 = T[i - 1 - m - k + q \dots i - 1]^R$ from the text. We compute the edit distance between P_1 and T_1 using Ukkonen's algorithm [121]. This can be done in $O(km)$ and returns an array ED_1 which contains the minimum edit distance of each row. The value of $ED_1[h]$ indicates the minimum distance between $P_1[1 \dots h]$ and $T_1[1 \dots r]$, where $h - k \leq r \leq h + k$. Similarly, We use the same algorithm to calculate the minimum edit distance array ED_2 between P_2 and T_2 . $ED_2[h]$ is the minimum distance between $P_2[1 \dots h]$ and $T_2[1 \dots r]$, where $h - k \leq r \leq h + k$.

Figure 5.4 shows the comparisons made by the algorithm. Here we have $P_1 = P[j + q \dots m] \circ P[1 \dots j - 1]$ and $P_2 = P_1^R = (P[j + q \dots m] \circ P[1 \dots j - 1])^R = P[1 \dots j - 1]^R \circ P[j + q \dots m]^R$. P_2 is matched against T_2 in reverse direction, while P_1 is matched against T_1 in the regular (forward) direction.

We construct an array ED from ED_1 and ED_2 as follows.

$$ED[h] = \begin{cases} ED_2[m - q - h] & : 1 \leq h \leq m - q \\ 0 & : m - q + 1 \leq h \leq m \\ ED_1[h - m] & : m + 1 \leq h \leq 2m - q \end{cases} \quad (5.1)$$

Lemma 5.4 : For a given hypothesis occurring at positions i and j in T and P respectively,

$$ED(f^{h+j+q-2}(P), subT_i) = ED[h] + ED[h + m - 1], \text{ where } 1 \leq h \leq m.$$

Proof : According to Figure 5.4, we construct a new string PP as $P_2^R \circ P[j \dots j + q - 1] \circ P_1$, where $P_2^R = P_1$. Then $PP = P[j + q \dots m] \circ P[1 \dots j - 1] \circ P[j \dots j + q - 1] \circ P[j + q \dots m] \circ P[1 \dots j - 1]$.

There are three cases in representing $f^h(P)$. These cases are indicated as numbers 1, 2, and 3 respectively on the double-headed arrows in Figure 5.4.

CASE 1 : $h = 1$, where $f^{h+j+q-2}(P) = PP[1...m]$. $PP[1...m]$ is constructed from two parts. One is P_2^R and the other is $P[j...j+q-1]$. The edit distance in this case is $ED[1] + ED[m]$ by Lemma 5.2, where $ED[1]$ is $ED_2[m-q]$ and $ED[m] = 0$.

CASE 2 : $2 \leq h \leq m-q$, where $f^{h+j+q-2}(P) = PP[h...h+m-1]$. $PP[h...h+m-1]$ is constructed from three parts. The first part of P_2^R . The second part is $P[j...j+q-1]$ and the last part is P_1 . From Lemma 5.2, we know the edit distance is $ED[h] + ED[h+m-1]$.

CASE 3 : $h = j$. This case is similar to the first case, so the edit distance is $ED[h] + ED[h+m-1]$.

In these three cases, we only compute $1 + m - q - 1 + 1 = m - q + 1$ circular shifts of the pattern. We do not calculate the circular shifts $f^r(P)$, where $j+1 \leq r \leq j+q-1$. There exists an exact match involving $P[j...j+q-1]$ (the matching q -gram), and $f^r(P)$ does not contain this substring, when $j+1 \leq r \leq j+q-1$. Therefore, it is not necessary to calculate $f^h(P)$, when $j+1 \leq r \leq j+q-1$. $\square$

Lemma 5.5 : For a given hypothesis occurring at positions i and j in T and P respectively, $ED_c(P, subT_i) = \min_{0 \leq h \leq m-1} \{ED[h] + ED[h+m-1]\}$, where $subT_i = T[i+q-1-m-k...i+m+k]$.

Proof : There are m circular shifts in the pattern P . From Lemma 5.4, we calculate all of the minimum edit distances between possible circular shifts $f^h(P)$ and the text $subT_i$ in this hypothesis, where $h \in [0, j] \cup [j+q, m]$. The circular edit distance of P against $subT_i$ is the minimum edit distance of all possible circular shifts against the text $subT_i$ in current hypothesis. The lemma holds. $\square$

Lemma 5.6 : For a given hypothesis, the time complexity of the verification algorithm is $O(km)$.

Proof : Algorithm 5.8 presents the verification processes following the above method. Clearly reversing the strings can be done in linear time, and computing edit distances on the

reversed string does not change the time required. Similarly, $ED_c()$ in Lemma 5.5 can be computed in $O(km)$ time, by maintaining an $O(m)$ array to record intermediate results during the computation of standard edit distance using dynamic programming. Line 4 calls the k -approximate pattern matching algorithm in $O(km)$ time using Ukkonen's algorithm [121] by bidirection. Line 5 implements equation (5.1). Line 6 to Line 9 run in $O(m)$ time cost to check the matching by Lemma 5.4. Thus the time complexity is $O(km)$. $\square$

Theorem 5.2: *Given a text $T = T[1..n]$ and a circular pattern $P = P[1..m]$, with symbols from an alphabet Σ , Algorithm ACPM2 solves the ACPM2 problem in $O(km^2n)$ time, and $O(n)$ space.*

Proof: Suffix tree construction (including suffix links) can done in linear time and linear space. After constructing the suffix tree, hypothesis generation phase is performed in $O(m \log |\Sigma|)$ time, independent of n or N , since at this stage we only need a count of the number of hypothesis, and the the parent nodes of the leaf nodes in the ST that correspond to the start positions of the matching q -grams in the text or database. Given pattern P and text T , there exists $O(mn)$ possible hypotheses. From Lemma 5.6, the time complexity of verification algorithm is $O(km)$ for each hypothesis. This means that the algorithm solves the ACPM2 problem in $O(km \times mn) = O(km^2n)$ time. $\square$

Theorem 5.3: *Given a database sequences $SeqDB$, with Z sequences and N total symbols ($|\Sigma|$ could be $O(N)$), Algorithm ACPM2 solves the all-against-all ACPM problem in $O(km_a N)$ time cost on average, and $O(km_m N^2)$ time worst case, using $O(N)$ worst case space, where $m_a = \frac{N}{Z}$, m_m is the length of the longest sequence in $SeqDB$.*

Proof: The result essentially from Theorem 5.2. For a sequence database $seqDB$ of length N , hypothesis generation phase is performed in $O(N \log |\Sigma|)$. The number of hypotheses will be $O(N^2)$, so the time complexity is $O(km_m N^2)$ in the worst case, where m_m is the length of the longest sequences. On average, the number of hypotheses is $O(\frac{N^2}{|\Sigma|^q})$, then the time complexity is $O(km_a \frac{N^2}{|\Sigma|^q})$, where m_a is the average length of sequences in $seqDB$. When q increases or $|\Sigma| \rightarrow N$, the time complexity will be $O(km_a N)$. Space requirement is in $O(N)$ to maintain the suffix tree data structure. $\square$

5.3.5 Comparison with Other ACPM Algorithms

In Table 5.2, we compare our ACPM-QGRAM algorithm and ACPM-BIDIRECTIONAL algorithm with the other related algorithms which were introduced in Chapter 2, namely Maes' algorithm [81], Gregor and Thomason's algorithm [44] and Uliel et. al's algorithm [123]. Weiner et. al's algorithm [126, 127] is a greedy algorithm which may miss some important circular relations. We also compare this algorithm with the other algorithms.

Our goal is to develop an algorithm to solve the ACPM problems and to apply this to study circular permutations in multidomain proteins. We make minor changes in Maes' algorithm [81], Gregor and Thomason's algorithm [44], Uliel et. al's algorithm [123] and Weiner et. al's algorithm [126, 127] for adapting them to the ACPM problems. Because these algorithms all focus on computing the circular edit distance, we extend them to match the pattern against all the substring of T . This takes $(n - m)$ steps, and hence the total time complexity will increase by $n - m = O(n)$ times.

The time complexity of our ACPM-QGRAM algorithm is $O(m_a^3 N^2)$ in the worst case. On average, the time complexity is $O(m_a^3 N^2 / |\Sigma|^q)$, where m_a is the average length of sequence and N is the total length of sequences, and $q = \lfloor \frac{m}{k+1} \rfloor$. When q increases, $O(N^2 / |\Sigma|^q)$ will be reduced to $O(N)$, since $|\Sigma|^q \leq O(N)$.

The time complexity of our ACPM-BIDIRECTIONAL algorithm is $O(k m_m N^2)$ in the worst case. On average, the time complexity is $O(k m_a N^2 / |\Sigma|^q)$, where m_a is the average length of sequence and N is the total length of the sequences, and $q = \lfloor \frac{m}{k+1} \rfloor$. When q increases, $O(N^2 / |\Sigma|^q)$ will be reduced to $O(N)$ since $|\Sigma|^q \leq O(N)$.

Table 5.2 shows the time complexity for the worst case of these algorithms. The last row shows the average case for the most challenging problem of all-against-all approximate circular pattern matching. Comparing with the ACPM-QGRAM, when m is large ($m = O(N)$), our q -gram algorithm will be worse. In this case, the Maes [81] algorithm will be the best algorithm in the worst case. But when $\frac{m}{k+1}$ increases and m is not very large, the ACPM-QGRAM algorithm will run in $O(m^3 N)$, where $m = \frac{N}{Z}$. This can be treated as a constant ($\frac{N}{Z} \approx 6$ for the case of multidomain proteins). Therefore, under such conditions, the ACPM-QGRAM algorithm is a linear time algorithm on average. The proposed ACPM-QGRAM algorithm is better than the

Table 5.2. Comparison with other proposed ACPM Algorithms

	Maes [81]	Gregor et. al's [44]	Uliel et. al's [123]	Weiner et. al's ¹ [126, 127]	ACPM- <i>q</i> gram	ACPM-BIDIRECTIONAL
One circular pattern against one text window	$O(m^2 \log m)$	$O(m^3)$	$O(m^3)$	$O(m^2)$	$O(m^2 \log m / \Sigma)$ $= O(m \log m)$	$O(km)$
One-against-One	$O(m^2 \log mn)$	$O(m^3 n)$	$O(m^3 n)$	$O(m^2 n)$	$O(m^3 \log m)$	$O(kmn)$
One-against-All	$O(m^2 \log mN)$	$O(m^3 N)$	$O(m^3 N)$	$O(m^2 N)$	$O(m^2 N \log m)$	$O(kmN)$
All-against-All	$O(\sum_1^Z m^2 N \log m)$ $= O(N^3 \log m)$	$O(\sum_1^Z m^3 N)$ $= O(N^4)$	$O(\sum_1^Z m^3 N)$ $= O(N^4)$	$O(\sum_1^Z m^2 N)$ $= O(N^3)$	$O(m^3 N^2 \log m)$	$O(kmN^2)$
Average Case All-against-All ($m = m_a = \frac{N}{Z}$)	$O(N^2 m_a \log m_a)$	$O(N^2 m_a)$	$O(N^2 m_a^2)$	$O(m_a N^2)$	$O(m_a^3 N \log(m_a))$	$O(kmN)$

other four related algorithms which were introduced in related work. Comparing the ACPM-BIDIRECTIONAL algorithm with Maes' algorithm [81] which is the best available algorithm for the ACPM1 problem, we can see that apply Maes' algorithm to the all-against-all ACPM problem requires time in $\Theta(N^2 m_a \log m_a)$ on average, and $O(N^3 \log m_m)$ worst case. These are still worse than our proposed algorithm that runs in $O(km_a N)$ time on average and $O(km_m N^2)$ worst case. Landau et al's algorithm [67] runs in $O(kmn)$ to solve the ACPM2 problem (for one-against-one). In the all-against-all ACPM2 case, the time complexity will be $\Theta(kN^2)$. This algorithm is worse than our algorithm that run in $O(km_a N)$ time on average.

5.4 Experiments

As discussed in Chapter 1, circular permutations and cyclic pattern have been used in various studies in biology. We performed some experiments using the results of the proposed algorithms to study circular permutations in molecular biology. In our experiments, we apply our algorithms on multidomain proteins to look for potential circular permutation relationships between them. We also use the results of the proposed algorithm to predict potential functions for uncharacterized or unknown proteins. In these experiments, the alphabet size is the number of domains which is a large number, close to 10^6 .

¹Algorithm could produce incomplete results

5.4.1 Data Set

Protein Domain Database (ProDom)

The protein domain is a section of the protein sequence whose structure can evolve, function and it exists independently of the rest of the protein chain [32]. Most proteins consist of several domains. The same protein domain may occur in related proteins. The ProDom is a database of known protein domains. The ProDom web site (<http://ProDom.prabi.fr>) provides a tool to search a protein domain in the protein database. The results are the proteins which contain a given protein domain. Each domain is represented as a unique symbol, thus a multidomain protein is viewed as a sequence of such symbols. The length of the domain representation is generally much smaller than the original protein sequence, but the size of alphabets has increased drastically.

Gene Ontology Database (GO)

The Gene Ontology (GO) project (<http://www.geneontology.org/>) provides a description of genes and protein products in different databases including the known functions of the genes. Currently the GO Consortium includes many databases such as GeneDB (<http://www.genedb.org/>), UniProtKB-Gene Ontology Annotation @ EBI (UniProtKB-GOA) (<http://www.ebi.ac.uk/GOA/>) and FlyDB (<http://flybase.bio.indiana.edu/>). More details on the GO Consortium is available at <http://www.geneontology.org/GO.consortiumlist.shtml>.

The ProDom database provides the Accession Number for the parent protein of each domain. The Accession Number is also provided for UniProtKB-GOA. This establishes a connection between entities in ProDom and their corresponding entities in GO database. In our experiments, we used this relation to obtain the GO terms used to describe the protein function. Based on this, we can predict functions for multidomain proteins using our CPM results obtained using the proteins in ProDom.

5.4.2 CPM Experimental Design

We implemented the exact circular pattern matching algorithm and three approximate circular pattern matching algorithms and applied them to detect circular patterns in ProDom database. We downloaded data from ProDom web site(<http://ProDom.prabi.fr>) on March 12, 2009 (ProDom version 2006.1 as released on November 6th, 2008). There were 1,997,497 proteins in this database. We removed proteins with less than three domains, and also removed redundant proteins with more than 90% similarity to some other protein. The result is a reduced database with 973,686 proteins. This means that ECPM and ACPM will only apply to proteins that contains an entire copy of another protein.

Results: Speed and Completeness

We ran the four algorithms on the reduced database and use the results to analyze the relationship between multidomain proteins. ACPM-QGRAM algorithm was executed on two different parameters, namely $q = 1$ and $q = 2$. When $q = 1$, the result is complete. When $q = 2$, the result is suboptimal (incomplete). We use the complete results as a benchmark to compare with the results from the algorithm.

The exact algorithm is the fastest algorithm. It only needed six minutes to build the suffix tree and perform searches for all circular patterns. ACPM-LIS algorithm is the slowest algorithm. ACPM-GREEDY algorithm is faster than the other ACPM algorithms, but the result has low accuracy (around 50%). Figure 5.5 shows the practical time required by these three algorithms, where q -gram has two instances, $q = 1$ and $q = 2$.

A comparison of the outputs of the algorithms provides some insight in their overall performance. There are 29,625,738 relations in the complete result. ECPM algorithm can be viewed as a greedy algorithm when the objective is approximate matching. The number of relations found using ECPM was 28,096,046 which is close to the complete result. ACPM-GREEDY only identified 15,075,729 relations. The ACPM-QGRAM algorithm with parameter $q = 2$ found 29,345,380 relations. When we run the ECPM algorithm in ProDom, we get almost 95% of the complete relations. When we run ACPM-QGRAM algorithm of $q = 2$, we get more

than 99% of the complete relations.

We run the hybrid algorithm where the ECPM algorithm was applied first and followed by using ACPM-QGRAM algorithm with parameter $q = 1$. We get the complete result and the time cost was reduced from 41 hours to 14 hours.

Analysis of Results

Based on the results, we built a relationship network among the multidomain proteins. This is a directed graph. The proteins are represented as the vertices, while the relations are represented by the edges. The In-edges and Out-edges are defined as follows. If a protein sequence P_1 is a circular pattern in protein sequence P_2 , then there is an Out-edge from P_1 to P_2 . Conversely, there is an In-edge from P_2 to P_1 . Figure 5.6 shows the degree distribution of the network. Panel (a) of Figure 5.6 is the degree distribution of all vertices and panel (b) shows the degree distribution of the Top-100 highest degree nodes. Panel (c) and (d) are log-log plots of panel (a) and (b) respectively.

Each protein sequence is not only used as a pattern to search against the other protein sequences, but also used as text to be searched against using the other protein sequences in the database. 424,888 protein sequences were found to be a pattern in some other protein sequences. 799,044 protein sequences contain at least one other protein sequence as a circular pattern. 374,279 protein sequences have both out-edge and in-edges. 50,609 protein sequences only have out-edges while 424,765 protein sequences only have in-edges. The average degree of this graph was 23 with an average out-degree of 46 and an average in-degree of 24.5.

Figure 5.7(a) shows the number of directly connected pairs in the Top- K highest degree proteins, where K is 10, 20 ... 1000. Let the Top- K highest degree proteins be vertices of a subgraph, the number of directly connected pairs is the number of edges. We define a ratio ρ_K as follows: $\rho_K = \frac{\# \text{ of total edges}}{\# \text{ of edges in Top-}K \text{ complete subgraph}} = \frac{\# \text{ of observed edges}}{\frac{1}{2} \times K \times (K-1)}$. Figure 5.7(b) shows the ratio ρ_K in Top- K proteins. When K is less than 460, the ratio ρ_K stays stable at around 0.5. When K is larger than 460, the ratio ρ_K starts to decrease. Thus in this graph, Top 460 highest degree proteins have higher relations.

Table 5.3. Top 15 highest degree proteins with GO function

Rank	Count	AC Number	Go Description
1	23353	Q7VMZ1	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
2	23344	Q9CPC5	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
3	23338	Q3EG14	Protein not found in GO
4	20508	Q33HH1	Protein not found in GO
5	20446	Q47AY9	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
6	20446	Q4UQ62	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
7	20446	Q8P4K7	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
8	20446	Q8PG73	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
9	20415	Q426Q5	Protein not found in GO
10	20398	Q3BNR9	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
11	20393	Q73PA3	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
12	20273	Q50XK7	Protein not found in GO
13	20246	Q66C16	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity
14	20244	Q5NU40	No function in GO
15	20244	O32748	nucleotide binding ; ATP binding ; ATPase activity ; nucleoside-triphosphatase activity

Protein Function Prediction

Table 5.3 shows the protein function for the Top-15 highest degree proteins. We notice that 10 of the 15 proteins have exactly the same functions. There are four proteins (rank is 3,4,9,12 respectively) that do not have entries in the GO database. Protein Q5NU40 (rank 14) has a record in GO database, but there is no function assigned to it in GO database. With high probability, we can say that the four proteins with no known function are likely to have the same function as the other 10 proteins. We plan to verify these functions by searching the biology literature in future.

We use the z -score as a measure of significance of the relationship between two proteins. For a given random variable x , the z -score is defined as follows. $z = \frac{x - \mu_x}{\sigma_x}$, where μ_x is the mean, and σ_x is the standard deviation.

To predict the function for a protein say P_A , we compute the z -scores for the number of occurrences of given function for the proteins in the respective In-edge and Out-edge sets for protein P_A . We then assign the protein function as the function with z -score above a threshold.

Table 5.4 shows the prediction results on 9 multidomain protein sequences using the union of the functions of the In-edge and Out-edge proteins at different thresholds on the z -scores.

Table 5.5 shows equivalent result using intersection.

We also conducted an experiment to predict the protein functions in the Top-500 highest degree proteins of these 156 proteins were not found in the GO database. Table 5.6 shows the prediction performance in terms of *precision*, *recall* and the *F*-measure, where *FP* is the number of false positive; *FN* is the number of false negative; *TP* is the number of true positive. The *recall* is calculated as $\frac{TP}{TP+FN}$ and the *precision* is calculated as $\frac{TP}{TP+FP}$. The *F*-measure is calculated as $2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$.

From the *F*-measure of Table 5.6, we notice the union method at $z \geq 3$ has the highest *F*-measure 0.84. This indicates the union method at $z \geq 3$ provides the best result of all the combination.

5.4.3 Multidomain Protein Networks using Circular Patterns

Introduction

Philipp et al. [100] introduced a tool to discover the potential relationships between proteins using the protein domain network. This protein domain network was based on the protein domain interaction networks. They built a web resource to explore the Protein Domain Interaction MAP(DIMA). In this network, the nodes are the protein domains and the edges are the interactions between two protein domains. In our work, network formation is based primarily on cyclic relationships between multidomain proteins.

Dataset

In this experiment, we further studied the use of our proposed ECPM and ACPM algorithms on the problem of analyzing multi-domain protein sequences. Based on the patterns found by our algorithms, we constructed multidomain protein networks by connecting different multidomain proteins that are found to be associated by some matching circular or non-circular patterns found by our algorithms. We use the Pfam database [15, 39] to identify the families for the multidomain proteins in ProDom. (We had introduced the Pfam database earlier in Chapter

Table 5.4. The predicted protein functions using union for In-edge and Out-edge

Protein AC Number	Function	Predicted Function ($z \geq 3$)	Predicted Function ($z \geq 2$)	Predicted Function ($z \geq 1$)
Q7VMZ1	GO:0000166 GO:0005524 GO:0016887 GO:0017111	GO:0000166 GO:0005524 GO:0016887 GO:0017111	GO:0000166 GO:0005524 GO:0016887 GO:0017111 GO:0042626	GO:0000166 GO:0005215 GO:0005524 GO:0016787 GO:0016887 GO:0017111 GO:0042626
O32184	GO:0003824 GO:0005488 GO:0016491	GO:0003824 GO:0005488 GO:0016491	GO:0003824 GO:0005488 GO:0016491	GO:0003824 GO:0004316 GO:0005488 GO:0016491
Q2Y7W6	GO:0000156	GO:0000155 GO:0004871	GO:0000155 GO:0004871	GO:0000155 GO:0004871
Q33CH5	GO:0003723 GO:0003968	GO:0003723 GO:0003968	GO:0003723 GO:0003968	GO:0003723 GO:0003968
Q30U32	GO:0000156	GO:0000156 GO:0004871	GO:0000155 GO:0000156 GO:0004871	GO:0000155 GO:0000156 GO:0004871
O93828	GO:0004585 GO:0016597 GO:0016740 GO:0016743	GO:0004585 GO:0016597 GO:0016740 GO:0016743	GO:0004585 GO:0016597 GO:0016740 GO:0016743	GO:0004585 GO:0016597 GO:0016740 GO:0016743
Q30SN9	GO:0003824 GO:0004252			GO:0003824 GO:0004252 GO:0005515
Q2YTY7	GO:0003723 GO:0009982			GO:0003723 GO:0009982
O78911	GO:0008137 GO:0016491	GO:0008137 GO:0016491	GO:0008137 GO:0016491	GO:0008137 GO:0016491

Table 5.5. The predicted protein functions using intersection for In-edge and Out-edge

Protein AC Number	Function	Predicted Function ($z \geq 3$)	Predicted Function ($z \geq 2$)	Predicted Function ($z \geq 1$)
Q7VMZ1	GO:0000166 GO:0005524 GO:0016887 GO:0017111	GO:0000166 GO:0005524 GO:0016887 GO:0017111	GO:0000166 GO:0005524 GO:0016887 GO:0017111	GO:0000166 GO:0005524 GO:0016887 GO:0017111
O32184	GO:0003824 GO:0005488 GO:0016491			
Q2Y7W6	GO:0000156	GO:0004871	GO:0004871	GO:0000155 GO:0004871
Q33CH5	GO:0003723 GO:0003968		GO:0003723 GO:0003968	GO:0003723 GO:0003968
Q30U32	GO:0000156		GO:0004871	GO:0000155 GO:0004871
O93828	GO:0004585 GO:0016597 GO:0016740 GO:0016743			GO:0016597 GO:0016740 GO:0016743
Q30SN9	GO:0003824 GO:0004252			
Q2YTY7	GO:0003723 GO:0009982			
O78911	GO:0008137 GO:0016491	GO:0008137 GO:0016491	GO:0008137 GO:0016491	GO:0008137 GO:0016491

Table 5.6. Performance in Protein Function Prediction using the Top-500 Proteins

Method	Parameter	TP	FP	FN	Recall	Precision	F measure
Union	$z \geq 3$	1349	186	317	0.81	0.88	0.84
	$z \geq 2$	1353	1302	313	0.81	0.51	0.63
	$z \geq 1$	1464	1950	202	0.88	0.43	0.58
Intersection	$z \geq 3$	162	522	1504	0.1	0.24	0.14
	$z \geq 2$	1269	3891	397	0.76	0.25	0.37
	$z \geq 1$	1345	6857	321	0.81	0.16	0.27

4). There are 40807 protein sequences in ProDom database which is also in Pfam database. There are 12104 families in Pfam database. In our experiment, Proteins in ProDom that do not have corresponding families in Pfam were not included in this analysis. Similar to Chapter 4, for function prediction based on the circular pattern networks, we use the protein functions as maintained in the GO database.

To ensure diversity, and reduce the problem of redundancy in the protein families, each family is represented by only two member proteins. We select two proteins from each family in the Pfam database to construct a new network. For each family, we select the respective proteins with the maximum and minimum number of circular pattern relationships. These often correspond to the longest and shortest protein sequences in the family. Some proteins belong to multiple families, thus some proteins will be chosen several times. We only keep one of them on our network. This network presents non-redundant relationships with all protein families. There are 3659 proteins and 4725 families in this network. The resulting data set contained 3,659 proteins from 4,725 families.

Network Formation

We construct two types of network. One is based on the circular permutation relationship between proteins. We call this the “Protein” network. The other is based on the families. Any circular relationships between two proteins will be circular permutation relationship between the two families to which the two proteins belong. We call this the “Family” network. Further, we construct three networks for each of the two networks. The first is a network using only non-circular patterns (non-CPs) found between the proteins. This is constructed based on non-circular matching relationships between proteins. The second network is the circular pattern network. This network is constructed using only the circular matching relationships (excluding non-circular matches) between the multidomain proteins. The last network is the combined network which is constructed from all matching relationships (including both circular and non-circular matches). Thus, we construct six networks from our data. Table 5.7 shows the statistics of these six networks. The networks are shown in Figures 5.8-5.13.

Table 5.7. Network statistics for multidomain protein networks

Parameters	Protein non-circular network	Protein circular network	Protein combined network	Family non-circular network	Family circular network	Family combined network
Clustering Coefficient	0.157	0.102	0.174	0.181	0.084	0.249
Connected Components	756	407	786	600	369	608
Network Diameter	7	9	9	13	15	11
Shortest paths	23515	17104	25593	103683	58164	123380
Characteristic Path Length	2.079	2.099	2.083	3.525	3.814	3.294
Average number of neighbors	2.612	2.496	2.723	5.325	4.073	6.844
Number of nodes	3458	2299	3659	4416	3140	4725
Number of edge	4517	2869	7386	13031	6741	19772

Significance of Circular Pattern Networks

The networks in Figures 5.4.3 and 5.11 have two colors for the network nodes. The red nodes are the nodes in the non-circular pattern networks. The green nodes are the nodes found only in the circular pattern networks. We notice there are very few green nodes in the networks. There are also two colors for the edges. The blue edges indicate edges that occur in the non-circular networks. The pink edges indicate the edges that only occurred in the circular networks. From these two figures, we see that the pink edges show more significant differences between the non-circular pattern network and the circular pattern network.

For example, in the combined Protein network (with CPs and non-CPs), we can observe the pink edges between between some major clusters. This shows an important relationship between these clusters that are only exposed using the circular permutations. For function prediction work based on this network, one can expect that these edges will provide more information about the clusters, potentially leading to improved prediction results.

In the family networks, there are some interesting observations between the combined network and the non-circular network. The part labeled "G" in the combined network (Figure 5.4.3) is connected to the main component, but in the non-circular pattern network, the part G is a large component which is disconnected from the main component. This indicates that there exist some important edges that only occur red in the circular network. Such relationships can not be found using direct pattern matching, and require methods for circular pattern matching, as proposed in this chapter.

Table 5.8 shows the Top 25 proteins that exhibited the highest difference (in terms of node degree) between protein circular-pattern network and non-circular pattern network. Perhaps, the significance of the CP network is clearer in this table, which shows the quantitative differences between selected nodes in the two network. We can observe that, in some cases, the more than half of the associations to some nodes in the Protein networks are due mainly to the circular pattern relationships. We can also see that some small networks are formed only exclusively based on circular patterns, with no associations using direct patterns (i.e non-CPs).

Table 5.9 shows the proteins found on the longest path in the Protein network. Given the nature of the network, it means that these multidomain proteins must have some common circular patterns (and perhaps some non-circular) shared between them. Table 5.10 shows the corresponding longest chain of families in the Family network.

5.5 Summary

In this chapter, we present four ACPM algorithms to solve the ACPM2 problem and one algorithm to solve the ECPM problem. ECPM matching algorithm is not only the best algorithm in theory, but also it is the fastest in practice. The ACPM-BIDIRECTIONAL algorithm is the best of our ACPM algorithms. Comparing with other algorithms in literature, ACPM-BIDIRECTIONAL algorithm also has the best time complexity on average.

Based on the results, we analyzed circular permutations in multidomain proteins and used this to perform protein function prediction. Our results show a performance of 0.88, 0.81 in *precision* and *recall* respectively, at $z \geq 3.0$, using the union of the functions inter In-edge and Out-edge proteins.

Table 5.8. Top 25 proteins with the highest node degree differences between protein networks using the circular and non-circular patterns.

Protein	Degree in circular network	Degree in combined network	Degree in non-circular network	<i>Degree in circular network</i> <i>Degree in combined network (%)</i>
P03304	266	704	438	37.78%
P19525	221	592	371	37.33%
P35409	199	567	368	35.10%
Q00962	175	470	295	37.23%
P17546	175	464	289	37.72%
P42684	122	390	268	31.28%
P43699	147	402	255	36.57%
P48633	174	403	229	43.18%
P29476	123	348	225	35.34%
Q04610	122	320	198	38.13%
Q99323	113	306	193	36.93%
P22009	95	276	181	34.42%
Q06889	94	260	166	36.15%
Q03351	431	588	157	73.30%
P10272	82	238	156	34.45%
P03336	83	225	142	36.89%
P16112	101	240	139	42.08%
P38939	77	216	139	35.65%
P41381	81	203	122	39.90%
P19560	59	175	116	33.71%
P30963	68	176	108	38.64%
P26762	64	168	104	38.10%
P27742	72	169	97	42.60%
P19559	62	150	88	41.33%
P08049	65	150	85	43.33%

Table 5.9. The longest path in the Protein network

order	Protein	Family
1	P25930	Pfam-B_8089
1	P25930	Pfam-B_8090
2	P34540	Pfam-B_2386
3	P43141	Pfam-B_2823
4	Q03351	ig
4	Q03351	Pfam-B_11753
4	Q03351	Pfam-B_3706
4	Q03351	Pfam-B_3707
5	P29276	Pfam-B_10807
6	P13677	Pfam-B_135
6	P13677	Pfam-B_735
7	P11461	Pfam-B_8910
8	P03967	ras
9	P31133	Pfam-B_3874
10	P46870	Pfam-B_210
10	P46870	Pfam-B_229
10	P46870	Pfam-B_461
10	P46870	Pfam-B_547
10	P46870	Pfam-B_548
10	P46870	Pfam-B_616
11	P13068	Pfam-B_5383
12	P42686	Pfam-B_7008
12	P42686	Pfam-B_7009
13	P44768	Pfam-B_10433
14	P32745	Pfam-B_4919
14	P32745	Pfam-B_4920
15	P23678	Pfam-B_7252
15	P23678	PH
16	P25892	Pfam-B_3816
17	P31134	Pfam-B_1832
18	P35409	7tm_1
18	P35409	Pfam-B_3952
18	P35409	Pfam-B_725
19	P21838	Pfam-B_8095
20	P07199	Pfam-B_6062
21	P09803	Pfam-B_4482
21	P09803	Pfam-B_6146
22	P24710	Pfam-B_1620
22	P24710	Pfam-B_1621

Table 5.10. The longest path in the Family network

Order	Family
1	Pfam-B_6544
2	Pfam-B_11160
3	ins
4	vwc
5	Pfam-B_5115
6	Pfam-B_1677
7	Pfam-B_4686
8	Pfam-B_7481
9	Pfam-B_1063
10	Pfam-B_2682
11	Pfam-B_3197
12	Pfam-B_8974
13	Pfam-B_604
14	Pfam-B_818
15	adh_zinc
16	Pfam-B_11159
17	Pfam-B_842
18	Pfam-B_2068

Figure 5.1. Suffix tree for the string $T = \text{mississippi\$}$ with some suffix links.
 (This figure is the same as Figure 3.1 with some suffix link.)

Algorithm 5.1: ECPM Algorithm

ECPM(*ST*, *Pattern*)

```

1  PP ← Pattern ◦ Pattern[1...m-1]
2  Current ← ST.root, top ← 1, len ← 1
3  for ( i ← 1 to 2m-1 ) do
4    if (len = 1) and Current.child.label[1]=PP[i] then
5      Current ← Current.child
6    end if
7    if Current.child.label[1]=PP[i] then
8      len ← len + 1
9      if len > label.length then
10       len ← 1
11     end if
12     if i-top=m-1 then
13       output "(matched, Current)"
14       Current ← Current.SuffixLink, top ← top + 1
15     end if
16   else
17     Current ← Current.SuffixLink
18     top ← top + 1, i ← i - 1
19   end if
20   if i < top or top > m then
21     break
22   end if
23 end for

```

Algorithm 5.2: All vs. All ECPM Algorithm

ALL-VS-ALL-ECPM(*SeqDB*, *Z*)

```

1  ST ← Get Suffix Tree(SeqDatabase)
2  for ( i ← 1 to Z ) do
3    Pattern ← SEQDB[i]
4    ECPM(ST, Pattern)
5  end for

```

Algorithm 5.3: Pattern Matching Using LIS

APM-VIA-LIS(T, P, k)

```

1  Build the mapping table mapTable which stores the positions in  $P$  of each symbol in decreasing order
2   $seq \leftarrow \text{NULL}$ 
3  for ( $i \leftarrow 1$  to  $n$ ) do
4       $seq \leftarrow seq \circ mapTable[T[i]]$ 
5  end for
6  Generate LIS from  $seq$ 
7  Calculate LCS between  $T$  and  $P$  from LIS
8  if verify(LCS,  $k$ ) is true then
9      return matched
10 else
11     return mismatched
8  end if
```

Algorithm 5.4: ACPM2 with Greedy Algorithm

ACPM-GREEDY($SeqDB, Z, k$)

```

1  for ( $i \leftarrow 1$  to  $Z$ ) do
2      for ( $j \leftarrow 1$  to  $Z$ ) do
3           $P \leftarrow SeqDB[i], m \leftarrow |P|, PP \leftarrow P[1...m] \circ P[1...m-1]$ 
4           $T \leftarrow SeqDB[j], n \leftarrow |T|$ 
5          APM-via-LIS( $T, PP, k$ )
6      end for
7  end for
```

Algorithm 5.5: ACPM2 Algorithm with LIS

```

ACPM-LIS(SeqDB, Z, k)
1  for (i  $\leftarrow$  1 to Z) do
2    for (j  $\leftarrow$  i + 1 to Z) do
3      P  $\leftarrow$  SeqDB[i], m  $\leftarrow$  |P|
4      T  $\leftarrow$  SeqDB[j], n  $\leftarrow$  |T|
5      for (v  $\leftarrow$  1 to n - m + k) do
6        subT  $\leftarrow$  T[v...v + m + k]
7        for (h  $\leftarrow$  1 to m) do
8          APM-via-LIS(subT,  $f^h(P)$ , k)
9        end for
10     end for
11  end for
12 end for

```

Figure 5.2. The number of hypotheses with *q*-gram

Algorithm 5.6: ACPM2 Algorithm with q -gram and Suffix Array

```

ACPM-QGRAM( $SeqDB, N, Z, q, k$ )
1   $seq \leftarrow NULL, pos \leftarrow NULL, s \leftarrow 1$ 
2  for ( $i \leftarrow 1$  to  $Z$ ) do
3     $seq \leftarrow seq \circ SeqDB[i]$ 
4    for ( $j \leftarrow 1$  to  $m_i$ ) do
5       $pos[s] \leftarrow i, s \leftarrow s + 1$ 
6    end for
7  end for
8   $\langle SA, lcp \rangle \leftarrow BuildSA(seq)$ 
9  for ( $i \leftarrow 1$  to  $N$ ) do
10    $Candidates \leftarrow \{\}$ 
11   do while ( $lcp[i] \geq q$ )
12      $Candidates \leftarrow Candidates \cup \{i\}, i \leftarrow i + 1$ 
13   end do
14   for each Pair  $\{x, y\} \in Candidates$  do
15      $P \leftarrow SeqDB[pos[SA[x]]], m \leftarrow |P|$ 
16      $T \leftarrow SeqDB[pos[SA[y]]], n \leftarrow |T|$ 
17     for ( $j \leftarrow \min(1, y - m - k + q)$  to  $y + m + k - q$ ) do
18        $subT \leftarrow T[j \dots j + m + k - 1]$ 
19       for ( $h \leftarrow 1$  to  $m$ ) do
20          $APM\text{-}via\text{-}LIS(subT, f^h(P), k)$ 
21       end for
22     end for
23   end for
24 end for

```

Figure 5.3. Dynamic Programming in q -gram matching

Figure 5.4. Three cases in computing the circular edit distance in the ACPM algorithm using the bidirectional edit distance. The numbered double-header show the symbol positions involved in each case.

Algorithm 5.7: q -gram ACPM Hypothesis Generation

ACPM-BIDIRECTIONAL(SeqDB[], k, N)

```

1  ST  $\leftarrow$  Build-Suffix-Tree(SeqDB)
2  for  $i=1$  to  $N$ 
3       $P \leftarrow$  SeqDB[ $i$ ]
4       $m \leftarrow |P|$ 
5       $q \leftarrow \lfloor \frac{m}{k+1} \rfloor$ 
6      start  $\leftarrow 1$ 
7      {< SeqDB ID, position >}  $\leftarrow$  Search-in-ST(ST,  $P[1...q]$ )
8      for each pair < SeqDB ID, position >
9           $T \leftarrow$  SeqDB[SeqDB ID]
10         BIDIRECTIONALED( $P, 1, T, \text{position}, k, q$ )
11     end for
12     start  $\leftarrow$  start + 1
13     do while (start  $\leq$  m-p)
14         {< SeqDB ID, position >}  $\leftarrow$ 
            Search-in-SuffixLink(ST,  $P[\text{start}...\text{start}+q-1]$ ) via suffix link
15         for each pair < SeqDB ID, position >
16              $T \leftarrow$  SeqDB[SeqDB ID]
17             BIDIRECTIONALED( $P, 1, T, \text{position}, k, q$ )
18         end for
19         start  $\leftarrow$  start + 1
20     end do
21 end for
```

Algorithm 5.8: ACPM Hypothesis Verification

```

BIDIRECTIONALED( $P, posP, T, posT, k, q$ )
1   $m \leftarrow |P|$ 
2   $P1 \leftarrow P[posP+q...m] \circ P[1...posP-1]$ 
3   $P2 \leftarrow P1^R$ 
4   $ed1 \leftarrow DP(P1, T[posT+q...posT+q+m+k-1], k)$ 
;  $ed2 \leftarrow DP(P2, T[posT-m+q...posT-1]^R, k)$ 
5   $ED \leftarrow \text{ComputeED}(ED_1, ED_2, m, q)$ 
6  for  $h=1$  to  $m-q+1$ 
7      if  $(ED[h] + ED[h+m-1] \leq k)$  then do
8          return match
9  end for

```

Figure 5.5. The time cost of the CPM algorithms

(a) Degree distribution

(b) Degree distribution in Top-100 degree nodes

(c) Log degree distribution

(d) Log degree distribution in Top-100 degree nodes

Figure 5.6. Degree distributions in the network of multidomain proteins constructed based on the circular patterns they contain.

Figure 5.7. Number of directly connected pairs in Top-K highest degree proteins

Figure 5.8. The Protein network (using both CPs and non-CPs)

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Figure 5.9. The Protein network using only non-circular patterns

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Figure 5.10. The Protein network using only circular patterns

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Figure 5.11. The Family network (using both CPs and non-CPs)

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Figure 5.12. The Family network using only non-circular patterns

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Figure 5.13. The Family network using only circular patterns

Red nodes are nodes found in the non-CP network. Green nodes are nodes found only in the CP-network. Blue edges denote edges found in the CP network. Pink edges are edges found only in the CP network.

Chapter 6

Circular Pattern Discovery

6.1 Introduction

The circular pattern discovery (CPD) problem is to identify “interesting” circular patterns in text T . Here, “interesting” is typically defined in terms of constraints in the search, for instance, based on occurrence frequency, pattern length, proximity between patterns, etc. When T is a database of sequences, additional constraints, may be imposed, for example the coverage or quorum constraint [86]. In biological applications, for instance, interesting circular patterns are likely to have biological relevance, e.g., they could point to proteins with related functions, even with low sequence similarity.

The CPD problem is related to the more well-known circular pattern matching (CPM) problem discussed in Chapter 5.

To our knowledge, there is no existing work that explicitly studied the problem of pattern discovery involving circular patterns. Motivated by the increasing significance of circular permutations in various applications, from computational biology to pattern analysis, we propose methods to address the circular pattern discovery problem.

Main Results. We define and solve the ECPD and ACPD problems to find the “interesting” circular patterns, as defined using specified constraints.

In this chapter, we present an algorithm to solve the ECPD problem. We also present two algorithms to solve the ACPD problem. One algorithm is based on Maes [81] CPM algorithm. Another algorithm is based on our ACPM2 algorithm. On average, the ACPD algorithm based on ACPM2 algorithm is better than the ACPD algorithm based on Maes [81] CPM algorithm.

The following two theorems represent our main contribution on the CPD problem.

Theorem 6.1: *Given a database sequence SeqDB, with r sequences, and the parameters m_1, m_2, k, f, g , The algorithm ECPD uses suffix trees and suffix links to solve the exact circular pattern discovery problem in $O(m_2^2 N)$ time, where N is the total number of symbols in SeqDB.*

Theorem 6.2: *Given a database sequence SeqDB, with r sequences, and the parameters m_1, m_2, k, f, g , Algorithm ACPD which based on ACPM2 algorithm uses suffix array to solve the ACPD problem in $O(km_2^3 N^2)$ worst case, and $O(km_2^3 N)$ on average, where N is the total number of symbols in SeqDB.*

Organization. In the next section, we define the circular pattern discovery problems. Algorithms for the ECPD problem are presented and analyzed in Section 3. The ACPD problems are introduced and solved in Section 4. In Section 5, we show experiments on analyzing circular permutations in multidomain proteins using our algorithms. In Section 6, we summarize our work on circular pattern matching problems.

6.2 The Circular Pattern Discovery Problem

Although a lot of progress has been made in pattern discovery, sequential data mining, and pattern matching, there has not been much attention to the issue of pattern discovery with cyclic patterns. While the pattern matching problem assumes that a pattern of interest will be provided before matching can start, pattern discovery does not require any initial pattern. Starting with no specific query pattern, one may seek to find “interesting” circular substrings within the sequence, or database of sequences, for example, circular substrings that occurred with a minimal number of occurrences. We call this the *circular pattern discovery* (CPD) problem. We consider three variations of the CPD problem below.

Exact Circular Pattern Discovery Problem (ECPD). Given a text T and a number f , return all the high frequency cyclic substrings s (i.e. with $\text{frequency} \geq f$) and their respective circular shifts in T .

Approximate Circular Pattern Discovery Problem (ACPD). Given a text T and the numbers f and k , return all the high frequency circular substrings s (i.e. with $\text{frequency} \geq f$) and their respective circular shifts with k -approximate matches in T .

Circular Pattern Discovery Problem (parameterized form). Given a database sequence Q , with r sequences, and the parameters m_1, m_2, k, f, g , return all circular substrings s and their respective circular shifts that have a k -approximate match in Q , with a high occurrence frequency ($\text{frequency} \geq f$), occurs in at least g sequences (where $g \leq r$), and the length m of each matching substring satisfies the constraint $m_1 \leq m \leq m_2$.

The parameter g models the coverage or quorum constraints, often imposed in motif discovery for biological sequences [86]. For instance, we may want a subsequence to appear in a certain proportion of the members of a protein family, before we can accept the subsequence as a valid motif for that family. The ECPD problem corresponds to the case with $k = 0$. Clearly, the parameterization could be modified to impose more or less constraints in the discovery, as desired.

The Challenge. To see the difficulty involved in the CPD problem, we can consider the complexity of the naïve algorithm for the problem. First, consider pattern discovery for patterns of a specific length, say m (that is, $m_1 = m_2 = m$), on a text T of length n . For the ECPD problem, we will have $(n - m + 1)$ substrings, each with m cyclic shifts, with each m -length pattern requiring $O(nm)$ time to search in T . The overall time will be in $O(m^2n^2)$. We can improve this to $O(m^2n)$ using standard linear time pattern matching algorithms, such as the KMP or Boyer-Moore algorithms. When we consider a range of pattern lengths (for example, $m_1 \leq m \leq m_2$), the overall complexity becomes $O(m_2^3n)$. When no length is specified, we will need to consider all possible pattern lengths, hence, the overall time complexity will be in $O(n^4)$. Using a similar analysis, for the ACPD problem, we will need time in $O(km^2n^2)$ for one single pattern length m , and in $O(km_2^3n^2)$ for a range of pattern lengths ($m_1 \leq m \leq m_2$). We have assumed the use of Ukkonen's $O(kn)$ algorithm for k -approximate matching of a given m -length substring of the text. With no specified constraint on the length, we will require time

in $O(kn^5)$ for the parametrized ACPD problem.

The CPD problem is related to the CPM problem, but is much more complicated. Applying CPM algorithms directly will be making the assumption that we know what we are trying to "discover", or would require an exhaustive consideration of all cyclic substrings. In the following, we first propose a fast ECPD algorithm, by exploiting suffix links, which are typically part of a standard suffix tree. We then address the ACPD problem, using suffix arrays.

6.3 The ECPD Algorithm

Our ECPD algorithm is based on our algorithm for the ECPM problem as presented in Section 5.2.

First the ECPD algorithm (Algorithm 6.1) will build a suffix tree ST from the sequence T . Then the algorithm checks each m -length pattern from T for possible circular pattern matching, using the suffix tree. Each operation of checking an m -length pattern for possible CPM will need an $O(m)$ time cost using the ECPM algorithm. There are $O((m_2 - m_1)N)$ patterns whose length m is in the range $[m_1, m_2]$. So the time complexity of the ECPD algorithm will be in $O((m_2 - m_1 + 1)(m_2 + m_1)N) = O(m_2^2 N)$.

Algorithm 6.1: ECPD Algorithm

```

ECPD( $T, N, m_1, m_2, f$ )
1   $ST \leftarrow \text{BuildSuffixTree}(T)$ 
2  for  $m = m_1$  to  $m_2$  do
3    for each  $m$ -length substring  $P$  of  $T$  do
4       $\eta_{occ} \leftarrow \text{ECPM}(ST, P)$ 
5      if  $\eta_{occ} \geq f$  then do
6        Output the circular pattern  $P$ 
7      end if
8    end for
9  end for

```

Based on the foregoing, we summarize our results on the ECPD problem in the following theorem:

Theorem 6.1: *Given a database sequence SeqDB, with r sequences, and the parameters m_1, m_2, k, f, g , The ECPD Algorithm uses suffix trees and suffix links to solve the exact circular pattern discovery problem in $O(m_2^2 N)$ time, where N is the total number of symbols in SeqDB.*

6.4 The ACPD Algorithm

We first apply an existing ACPM algorithm directly on the ACPD problem. This models the use of a generic ACPM algorithm for the ACPD problem. In particular, we modify Maes's ACPM algorithm to solve the ACPD problem. Subsequently, we describe our ACPD algorithm, and compare the two methods.

6.4.1 ACPD using Maes' Algorithm

Algorithm 6.2 shows a method to solve the ACPD problem based on Maes' algorithm. The algorithm compares each m -length pattern from the text with each $(m+k)$ -length subtext, where $m_1 \leq m \leq m_2$. For each length m , there are $\min\{O(N), O(|\Sigma|^m)\}$ patterns and subtext, so there are $O(N^2)$ comparisons. The total number of comparisons is in $O((m_2 - m_1)N^2)$. For each comparison, the time cost is in $O(m^2 \log m)$. Thus, the time complexity of Algorithm 5.10 is $\sum_{m=m_1}^{m_2} O(N^2 m^2 \log m) = O(m_2^3 N^2 \log m_2)$.

Landau's Algorithm

Algorithm 6.3 is based on Landau's incremental algorithm [67]. Similar to Maes' algorithm, this algorithm compares each m -length pattern with each $m+k$ -length subtext, where m is the length of pattern and $m_1 \leq m \leq m_2$. For each length m , there are $\min(O(N), O(|\Sigma|^m))$ patterns and subtext, so there are $O(N^2)$ comparisons. The total number of comparisons is $O((m_2 - m_1)N^2)$. For each comparisons, the time cost is $O(km)$. Thus, the time complexity of algorithm 6.3 is $\sum_{m=m_1}^{m_2} O(N^2 km) = O(km_2^2 N^2)$.

Algorithm 6.2: ACPD Based on Maes' algorithm

ACPD-MAES(T, N, k, m_1, m_2, f)

```

1  for  $m = m_1$  to  $m_2$  do
2    for each  $m$ -length substring  $P$  of  $T$  do
3       $\eta_{occ} \leftarrow 0$ 
4      for  $i = 1$  to  $N - m - k$  do
5        if MAESALGORITHM( $P, T[i \dots i + m + k]$ ) is true then do
6           $\eta_{occ} \leftarrow \eta_{occ} + 1$ 
7        end if
8      end for
9      if  $\eta_{occ} \geq f$  then do
10       Output  $P$ 
11      end if
12    end for
13  end for

```

Algorithm 6.3: ACPD Based on Landau's algorithm

ACPD-LANDAU(T, N, k, m_1, m_2, f)

```

1  ST  $\leftarrow$  BuildSuffixTree( $T$ )
2  for  $m = m_1$  to  $m_2$  do
3    for each  $m$ -length substring  $P_m$  do
4      for  $i = 1$  to  $N - m - k$  do
5        LANDAUALGORITHM( $P_m, T[i \dots i + m + k]$ )
6      end for
7    end for
8  end for

```

6.4.2 Proposed ACPD Algorithm

Our ACPD algorithm (Algorithm 6.4) uses the same framework as the described ACPM algorithm (Section 5.3.4). The algorithm first constructs the hypotheses on potential circular pattern matches using q -gram filtration. For each hypothesis (i.e. each matching q -gram in T), the ACPD algorithm verifies *all* possible circular shifts that involves this q -gram match.

The algorithm constructs $O(N^2)$ worst-case number of hypotheses with parameter $q = \lfloor \frac{m_1}{k+1} \rfloor$, where N is the total length of the concatenated sequences (for a database of sequences). On average, the number of hypotheses is $O(\frac{N^2}{|\Sigma|^q})$. When $|\Sigma|^q$ is close to $O(N)$, the number of hypotheses will reduce to $O(N)$.

At verification, the algorithm checks the circular shifts of length $m_1, m_1 + 1, \dots, m_2$. For each pattern with length m , there are $O(m)$ circular shifts to check, where $m_1 \leq m \leq m_2$. The time for checking all of m -length circular shifts in the same circular pattern involving the current hypothesis is in $O(km)$ using the verification algorithm (Algorithm 5.5). In each hypothesis, the number of circular pattern with length m is m . So the time cost of checking all circular shifts of circular pattern with length m in a hypothesis is $O(km^2)$. The total time for verification will be $O(\sum_{m=m_1}^{m_2} km^2) = O(km_2^3)$. The worst case time complexity of Algorithm 6.4 will thus be in $O(km_2^3N^2)$. On average, the running time will be in $O(\frac{km_2^3N^2}{|\Sigma|^q})$. When $|\Sigma|^q$ is close to $O(N)$, as is typically the case, the time complexity will reduce to $O(km_2^3N)$.

A further improvement will be to exploit the huge redundancy in going from a pattern of length m to a pattern of length $(m + 1)$. We observe that to go from an m -length pattern starting at position i in T to an $(m + 1)$ -length pattern starting from the same position involves adding just one symbol at the end. Most k -approximate cyclic pattern matches at one length will also be cyclic matches at the next length, if the cyclic edit distance is less than $k - 1$. Also, non-matching regions in T for the m -length pattern cannot be a match for the $(m + 1)$ -length pattern if the circular edit distance with the m -length pattern is greater than $k + 1$. Exploiting this redundancy by keeping the full dynamic programming table, will lead to a further reduction of the overall complexity to $O(m_2^2N^2)$ worst case and $O(m_2^2N)$ on average. This is made possible only by the nature of the CPD problem.

Based on the foregoing, we summarize our results on the ACPD problem in the following theorem:

Theorem 6.2: *Given a database sequence SeqDB, with r sequences, and the parameters m_1, m_2, k, f, g , Algorithm ACPD solves the ACPD problem in $O(km_2^3N^2)$ worst case, and $O(km_2^3N)$ on average, where N is the total number of symbols in SeqDB.*

In the above CPD algorithms, we have assumed one single input sequence, T , for simplicity. For a database of sequences, we can simply concatenate the sequences in the database to form one long sequence, and then construct the generalized suffix tree (or suffix array) for the long sequence. Then, we can keep track of the number of occurrence of each circular pattern in each sequence in the database to support quorum constraints.

6.4.3 Comparison

The time complexity of the ACPD solution via Maes' algorithm is $O(m_2^3N^2 \log m_2)$. The time complexity of the ACPD solution via Landau's algorithm is $O(km_2^2N^2)$. The time complexity of the proposed ACPD algorithm is $O(km_2^3N^2)$. The number of patterns is $\min N, |\Sigma|^m$. The proposed ACPD algorithm is based on q -gram filtration. The average number of hypotheses is in $O(\frac{N^2}{|\Sigma|^{\frac{m}{k}}})$. When $|\Sigma|^{\frac{m}{k}}$ is close to $O(N)$, the time complexity of the ACPD algorithm will reduce to $O(km_2^3N)$. By exploiting the nature of the CPD problem in terms of the redundancy between m -length patterns and $(m+1)$ -length patterns starting at the same location in the text, the complexity for the proposed ACPD algorithm can be reduced to $O(m_2^2N^2)$ worst case, and $O(m_2^2N)$ on average.

6.5 Experiments

We performed circular pattern discovery on the same protein multidomain sequences with pattern length m , where $4 \leq m \leq 30$. Figure 6.1 shows the variation of the number of distinct patterns (both circular and non circular pattern) with pattern lengths. When the length is 4, the number of distinct patterns is large. When the length increases, the number of distinct patterns

Algorithm 6.4: Proposed ACPD algorithm

```

ACPD-QGRAM( $T, N, k, m_1, m_2, f$ )
1   $\langle SA, LCP \rangle \leftarrow \text{COMPUTESA}(T)$ 
2   $q \leftarrow \lfloor \frac{m_1}{k+1} \rfloor, Q_{set} \leftarrow \text{NULL}$ 
3  for  $i = 1$  to  $N$  do
4      if  $LCP[i] \geq q$  then do
5           $Q_{set} \leftarrow c(Q_{set}, SA[i])$ 
6      else
7          for  $j = 1$  to  $\text{length}(Q_{set})$  do
8              for  $l = j+1$  to  $\text{length}(Q_{set})$  do
9                   $P_{pos} \leftarrow Q_{set}[j], T_{pos} \leftarrow Q_{set}[l]$ 
10                 for  $m = m_1$  to  $m_2$  do
11                      $T_1 \leftarrow T[T_{pos}+q+1 \dots T_{pos}+m+k]; T_2 \leftarrow \text{Text}[T_{pos}-(m-q+k) \dots T_{pos}-1]$ 
12                     for  $h = 0$  to  $m-q$  do
13                          $P \leftarrow T[P_{pos}+q+1 \dots P_{pos}+q+h] \circ \text{Text}[P_{pos}-m+q+h \dots P_{pos}-1]$ 
14                         if  $\text{BIDIRECTIONALED2}(P, T_1, T_2, k)$  is true then do
15                             if  $P$  is the first occurrence then do
16                                  $\eta_{occ}(P) \leftarrow 1$ 
17                             else
18                                  $\eta_{occ}(P) \leftarrow \eta_{occ}(P) + 1$ 
19                             end if
20                         end if
21                     end for
22                 end for
23             end for
24         end for
25          $Q_{set} \leftarrow \text{NULL}$ 
26     end if
27 end for
28  $\forall P$ , Output  $P$  if  $\eta_{occ}(P) \geq f$ 

```

Algorithm 6.5: Modified ACPM Hypothesis Verification

 BIDIRECTIONALED2(P, T_1, T_2, k)

```

1   $P2 \leftarrow P^R$ 
2   $ED1 \leftarrow DP(P, T_1, k)$ 
3   $ED2 \leftarrow DP(P2, T_2^R, k)$ 
4   $ED \leftarrow ED2^R \cup \text{rep}(0, q) \cup ED1$ 
5  for  $h=1$  to  $|P| + 1$ 
6      if  $(ED[h] + ED[h+m-1] \leq k)$  then do
7          return true
8      end if
9  end for
```

decreases quickly. Figure 6.2 shows the variation of the number of the circular patterns (i.e. excluding the non-circular patterns) with pattern lengths. As expected, the number decreases rapidly with increasing length. In Figure 6.3 we show the maximum number of occurrences of one pattern at different pattern lengths. Table 6.1 shows the number of distinct patterns and the number of non-circular patterns at different pattern lengths. Table 6.1 also shows the ratio with $\frac{\text{Number of distinct patterns}}{\text{Number of non-circular patterns}}$. Interestingly, some circular patterns were discovered to occur with lengths as large as 26 symbols. That is, 26 domains were repeated, but in the form of a cyclic permutation between two multidomain proteins in the database.

Table 6.2 shows an example of a circular pattern with five domains. The pattern is {PDA1N075, PD000041, PD000041, PD248344, PD000041}. We assign the symbol "A" to protein domain PDA1N075, the symbol "B" to protein domain PD000041 and the symbol "C" to protein domain PD248344. It occurs in multi-domain protein sequences Q99407_HUMAN, ANK1_HUMAN and Q13768_HUMAN with different orders.

Table 6.3 shows an example of circular pattern with thirteen domains. The pattern is {PD000768, PD000767, PD005993, PD000768, PD000767, PDA1J766, PD005993, PD000768, PD000767, PDA1J7O1, PD000165, PD272501, PDA1J3F2}. We assign the string {FGHFGIHFGJKLM} to the pattern. It occurs in the multi-domain protein sequences Q9Y4V9_HUMAN, Q5JR23_HUMAN and Q9UJ57_HUMAN with different orders.

Figure 6.1. Variation of the number of distinct patterns (including non-circular and circular patterns) with pattern length.

Figure 6.2. Variation of number of circular patterns with pattern length.

Table 6.1. The number of distinct patterns with pattern length

Pattern Length	# Distinct Patterns	# Non-circular Patterns	# Circular patterns	Ratio
4	80333	79934	399	100.50%
5	8199	7944	255	103.21%
6	4720	4705	15	100.32%
7	2754	2677	77	102.88%
8	836	830	6	100.72%
9	550	532	18	103.38%
10	855	855	0	100.00%
11	937	921	16	101.74%
12	1011	983	28	102.85%
13	727	676	51	107.54%
14	299	296	3	101.01%
15	265	262	3	101.15%
16	141	141	0	100.00%
17	139	139	0	100.00%
18	116	100	16	116.00%
19	50	50	0	100.00%
20	58	57	1	101.75%
21	51	51	0	100.00%
22	19	19	0	100.00%
23	37	37	0	100.00%
24	18	18	0	100.00%
25	34	34	0	100.00%
26	18	17	1	105.88%
27	19	19	0	100.00%
28	2	2	0	100.00%
29	8	8	0	100.00%
30	5	5	0	100.00%

Table 6.2. Sample discovered circular patterns with length five.

A≡PDA1N075, B≡PD000041, C≡PD000041, D≡PD248344, E≡PD000041

Protein	Pos	order	Pattern
Q99407_HUMAN	6	0	ABBCB
Q13768_HUMAN	7	0	ABBCB
ANK1_HUMAN	6	3	CBABB
Q13768_HUMAN	6	4	BABBC
Q99407_HUMAN	5	4	BABBC

Figure 6.3. Variation of maximum number of occurrences with pattern length.

Table 6.3. Sample discovered circular patterns with length thirteen.

F≡PD000768, G≡PD000767, H≡PD005993, I≡PDA1J766, J≡PDA1J7O1, K≡PD000165, L≡PD272501, M≡PDA1J3F2

Protein	Pos	order	Pattern
Q9Y4V9_HUMAN	20	0	FGHFGIHFGJKLM
Q5JR23_HUMAN	21	1	GHFGIHFGJKLMF
Q9UJ57_HUMAN	38	2	HFGIHFGJKLMFG

6.6 Summary

In this chapter, we introduced the ECPD and ACPD problems, and proposed two algorithms for their solution. The first algorithm uses suffix trees and suffix links to solve the ECPD problem in $O(m_2^2 N)$ time. The second algorithm solves the more challenging ACPD problem in $O(km_2^2 N^2)$ worst case, and $O(km_2^2 N)$ on average, using suffix arrays. By exploiting the redundancy between patterns of different lengths that start at the same position in the text, the overall complexity is reduced to $O(km_2 N^2)$ worst case, and $O(km_2 N)$ on average. Our results can be compared with the approach that directly uses Maes' algorithm, one of the best available ACPM algorithms for the ACPD problem, which runs in $O(m_2^3 N^2 \log m_2)$ time, or $O(m_2^3 N \log m_2)$ for small values of m_2 . Although pattern discovery has been well studied, to our knowledge, this is the first attempt at a focused study on the CPD problem.

We show an experiment for discovering the exact circular patterns in ProDom database. We reported interesting exact circular patterns discovered by our algorithm. To our knowledge, this is the first experiment at a focused study on discovering the exact circular patterns.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this work, we present two novel data structures to solve the space problem of the suffix tree and its applications. These two structures are the *virtual suffix tree* (VST) and the *Probabilistic Suffix Array* (PSA). We consider the circular pattern matching problem and define the circular pattern discovery problem. We present algorithms using suffix data structures to solve both the exact and inexact variants of the CPM problem. Our focus is on efficiently answering enumerative queries involving circular patterns. We also present algorithms based on our CPM algorithms to solve the CPD problem. To our knowledge, this is the first attempt at a focused study on discovering circular patterns.

The Virtual Suffix Tree. We introduce the VST (virtual suffix tree), an efficient data structure for suffix trees and suffix arrays. The VST provides the same functionality as the suffix tree, including support for pattern matching, and suffix links. But the VST requires a much smaller space than the suffix tree and the other recently proposed space-efficient data structures for suffix trees and suffix arrays. With n =length of the sequence, the worst case space is $18n$ bytes compared with $20n$ bytes for the other data structures for suffix trees and suffix arrays (such as the enhanced suffix array [2], and the linearized suffix tree [62]). On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the

regular VST, and $12.05n$ bytes in its compact form.

The Probabilistic Suffix Array. We have presented the probabilistic suffix array (PSA), a data structure for representing information in variable length Markov models. The PSA provides the same functionality as the probabilistic suffix tree (PST), but the space is significantly smaller than that of the PST. We construct the PSA in linear time and linear space, independent of the order of the Markov model. The space is dependent on the number of interval nodes of the suffix array. In the worst case, the memory requirement is $33N$ bytes. On average, the needed space will be $26N$ bytes, which includes the work space of the construction phase. The needed space for PSA is significantly smaller than that for the PST which is implemented on a regular suffix tree. Prediction using the PSA is in $O(m \log \frac{n}{|\Sigma|})$ time, where m is the pattern length, and Σ is the symbol alphabet.

Circular Pattern Matching. We defined the circular pattern matching (CPM) problems in the related work. We present a linear time algorithm to solve the ECPM problem which aims at finding an exact circular pattern P in the text T . The ACPM problem is to find an approximate occurrence of circular pattern P in text T . We present three algorithms for the ACPM2 problem and one greedy algorithm that produces an incomplete result. Of all the algorithms reported in the literature for the ACPM problem, our ACPM q -gram-based bidirectional algorithm with suffix trees provides the best results on average, with respect to both time and space complexity. Using our algorithms, we performed experiments on the analysis of circular permutations in multidomain proteins. Based on the results, we developed a method for function prediction for such multidomain proteins.

Circular Pattern Discovery. Based on the work on the circular pattern matching problem, we defined the Exact Circular Pattern Discovery (ECPD) problem and the Approximate Circular Pattern Discovery (ACPD) problem. We present an ECPD algorithm that uses suffix trees and suffix links to solve the exact circular pattern discovery problem in $O(m_2^2 N)$ time. We also present an efficient algorithm to solve the ACPD problem based on our bidirectional ACPM2 algorithm. Comparing with the ACPD algorithm based on Maes [81] CPM algorithm, our algorithm is the better solution on average.

7.2 Future Work

We briefly describe some potential future work, based on the material presented in this work.

7.2.1 Circular Pattern Discovery

We have presented algorithms for the circular pattern discovery problem. Our presented algorithms are based on our CPM algorithm. In our ACPD algorithm, there are lot of repeating operations that compares the circular shifts vs. the subtext. In our future thinking, we may find a way to avoid the repeated comparisons.

7.2.2 Network Analysis for Circular Multidomain Proteins

In this work we proposed efficient algorithms for rapid detection of cyclic permutations in multidomain proteins, using suffix trees and suffix arrays. Based on the results, we formed networks linking different multidomain proteins based on cyclic patterns observed in the proteins, using current data in the ProDom database [21]. Using these networks we performed functional annotation of the multidomain proteins. Significance of functional relatedness between two multidomain proteins was accessed using z -scores and p -scores. We performed only simple analysis of the networks, for instance, based on simple in-degree and out-degree characteristics of the nodes in the network. The overall performance using the method on a network formed with the Top 500 proteins (as ranked by degree) was as follows: **sensitivity 0.81; precision 0.88, F-measure 0.84**. Although based only on sequence data, and with no alignment, these results are comparable with ProFunc [69, 70] and ProKnow [101], both of which perform at around 70% accuracy. These are web-based servers that predict protein function from its 3D structure, using initial structure alignment, and a combination of algorithms. Motivated by these impressive results, an interesting future direction will be a more detailed study of these networks of cyclic patterns, built on just protein sequence data. Network characterization using more rigorous methods such as betweenness [43], centrality [52], and pairwise discontinuity [102] could show more light on the potential relationships between related multidomain proteins, or non-

related multidomain proteins with potentially similar function.

7.2.3 From PSA to PFA

The *probabilistic suffix automata* is a subclass of the *probabilistic finite automata* (PFA) which is a simple model for learning in Markov models with order L . Ron et al. [109] proved that “every distribution generated by a probabilistic suffix automata can equivalently be generated by a PST”. In our future work, one can consider how to construct a probabilistic suffix automata from our probabilistic suffix array data structure. Previous work [9, 109] has constructed the probabilistic suffix automata from the probabilistic suffix tree (PST). But the space and time complexity is $O(Ln^2)$ which is huge. The compressed suffix array [45] and the compact suffix array [82, 83] are space-efficient suffix data structures. It would be interesting to study how these space-efficient suffix data structures can be used to improve the PSA.

7.2.4 Approximate Pattern Matching Using PSA

Dynamic programming [114] is a well know method to find approximate patterns, but the time complexity is $O(m^2n)$ to find all the occurrences, where m is the length of pattern and n is the length of the text. Ukkonen’s algorithm [121] improved the time complexity of this problem to $O(mnk)$, where k is maximum error allowed in the pattern. This is still large in practice. In our thinking, the probability generated from PSA prediction may give us a useful measurement for the approximate pattern matching problem. This simple thinking will work for existential queries in $O(n + m \log |\Sigma|)$ time, where $O(n)$ is the PSA construction cost and $O(m \log |\Sigma|)$ is the prediction cost. But for enumerative queries, finding all the possible matching positions using the PSA will be a major challenge.

7.2.5 Prediction with PSA using Inexact Matching

Markov models in information retrieval are often based on item frequency and document frequency. However, in the real world, there are often some mismatch/errors (i.e. insert, delete, substitute) in the sequence. If we could compute the Markov Model based on an approximation

of the item frequency and document frequency, we will get a more robust model in prediction. Here, approximate frequencies are computed by considering possible mutations in the sequence.

Calculating Approximate Probability

The transition matrix of the Markov model will be calculated based on an approximate probability. The approximate probability is calculated by the approximate item frequency and document frequency. We use the following equation in place of the regular equation which calculates the transition matrix using the exact term frequency (TF) or document frequency (DF).

$$P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1} \dots, X_0 = i_0) = \arg \max_{ED(Y,X) \leq k} P(Y_{m+n} = j | Y_n Y_{m+n-1} Y_{m+n-2} \dots Y_{m+n-L})$$

where $ED(Y,X) = ED(Y[m+n-L+1 \dots m+n], X[n-L+1 \dots n])$

Difficulty

For this extension of the PSA, we need to cope with two hard problems.

1. How to calculate the approximate term frequency (TF) and document frequency (DF) in efficient time and space complexity. This is similar to the motif discovery problem. We have to find a better algorithm to calculate the approximate term frequency (TF) and document frequency (DF).
2. How to associate nodes in the PSA to the approximate probability. In our current PSA, symbols in the same edge share the same node. Hence they have the same probability, tf and df. Thus the space for the PSA is linear with respect to the length of sequence. In this future work, symbols in the same edge may not share the same probability. Thus in the naïve implementation, the space for the PSA will be quadratic with respect to the length of the sequence. Reducing this space requirement is an interesting challenge.

If these two problems can be solved, a robust representation for Markov models will be achieved.

7.3 Publications from the Dissertation

1. Jie Lin, Yue Jiang, and Don Adjero. The Virtual Suffix tree: An efficient data structure for suffix trees and suffix arrays, the Prague Stringology Conference (PSC), 2008.
2. Jie Lin, Yue Jiang, Donald A. Adjero. The Virtual Suffix Tree, International Journal of Foundations of Computer Science, Vol:20, Issue No:6, pp1109-1133, 2009.
3. Jie Lin and Don Adjero. All-against-all circular pattern matching. 2011. Under review.
4. Jie Lin and Don Adjero. Circular Pattern Discovery. 21st International Workshop on Combinatorial Algorithms, 2011.
5. Jie Lin, Don Adjero and Bing-Hua Jiang. Probabilistic Suffix Array: Efficient Modelling and Prediction of Protein Families. 2011. Under review.
6. Jie Lin, Don Adjero and Bing-Hua Jiang. Algorithms for Efficient Detection of CPs in Multidomain Protein. 2011. To be submitted.

Bibliography

- [1] N Abe and M Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.
- [2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53 – 86, 2004.
- [3] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*. Springer-Verlag, 2008.
- [4] Don Adjeroh and Fei Nan. Suffix sorting via Shannon-Fano-Elias codes. *Algorithms*, 3(2):145–167, 2010.
- [5] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.
- [6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.
- [7] Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. In *FOCS*, pages 144–153, 1997.
- [8] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. 25(2):129–141, 1995.
- [9] Alberto Apostolico and Gill Bejerano. Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. *Journal of Computational Biology*, 7(3-4):381–393, 2000.

- [10] Alberto Apostolico and Maxime Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991.
- [11] Alberto Apostolico and Raffaele Giancarlo. The Boyer Moore Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.
- [12] H. Arimura, H. Asaka, H. Sakamoto, and S. Arikawa. Efficient discovery of proximity patterns with suffix arrays. In A. Amir and G.M. Landau, editors, *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 152–156. Springer, 2001.
- [13] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In N.J. Belkin and C.J. van Rijsbergen, editors, *SIGIR 89, Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, volume 23, pages 168–75. ACM, New York (published as a special issue of SIGIR Forum, Vol. 23, 1-2, Fall 88/Winter 89), 1989.
- [14] Ricardo A. Baeza-Yates and Chris H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996.
- [15] Alex Bateman, Lachlan Coin, Richard Durbin, Robert D. Finn, Volker Hollich¹, Sam Griffiths-Jones, Ajay Khanna, Mhairi Marshall, Simon Moxon, Erik L. L. Sonnhammer¹, David J. Studholme, Corin Yeats, and Sean R. Eddy. The Pfam protein families database. *Nucleic Acids Res*, 32 (Database issue):D138–D141, 2004.
- [16] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order Markov models. *J. Artif. Intell. Res. (JAIR)*, 22:385–421, 2004.
- [17] Gill Bejerano and Golan Yona. Variations on probabilistic suffix trees: Statistical modeling and prediction of protein families. *Bioinformatics*, 17(1):23–43, 2001.
- [18] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
- [19] Kellogg S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980.

- [20] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1977.
- [21] Catherine Bru, Emmanuel Courcelle, Sbastien Carrre, Yoann Beausse, Rine Dalmar, and Daniel Kahn. The ProDom database of protein domain families: More emphasis on 3D. *Nucleic Acids Res*, 33:212–215, 2005.
- [22] H. Bunke and U. Buhler. Applications of approximate string matching to 2D shape recognition. *Pattern Recognition*, 26(12):1797–1812, December 1993.
- [23] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.
- [24] Y. Cao, A. Janke, P.J. Waddell, M. Westerman, O. Takenaka, S. Murata, N. Okada, S. Paabo, and M. Hasegawa. Conflict among individual mitochondrial proteins in resolving the phylogeny of eutherian orders. *J Mol Evol.*, 47:307–322, 1998.
- [25] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988.
- [26] William I. Chang and Jordan Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 175–184, London, UK, 1992. Springer-Verlag.
- [27] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
- [28] John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997.
- [29] Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. In *STOC*, pages 407–415, 2000.
- [30] S Cong, J Han, and D A Padua. Parallel mining of closed sequential patterns. In *KDD*. ACM, 2005.

- [31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [32] Florence Corpet, Florence Servant, Jerome Gouzy, and Daniel Kahn. ProDom and ProDom-CG: Tools for protein domain analysis and whole genome comparisons. *Nucleic Acids Res*, 28(1):267–269, 2000.
- [33] Paul D. Cotter, Colin Hill, and R. Paul Ross. Bacterial lantibiotics: Strategies to improve therapeutic potential. *Current Protein and Peptide Science*, 6:61–75, 2005.
- [34] Maxime Crochemore and Thierry Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Information Processing Letters*, 63:195–203, 1997.
- [35] Craik DJ. Circling the enemy: Cyclic proteins in plant defence. *Trends Plant Sci.*, 14(6):328–335, 2009.
- [36] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
- [37] Yariv Ephraim, Amir Dembo, and Lawrence R. Rabiner. A minimum discrimination information approach for hidden Markov modeling. *IEEE Transactions on Information Theory*, 35(5):1001–1013, 1989.
- [38] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
- [39] Robert D. Finn, John Tate, Jaina Mistry, Penny C. Coghill, Stephen John Sammut, Hans rudolf Hotz, Goran Ceric, Kristoffer Forslund, Sean R. Eddy, Erik L. L. Sonnhammer, and Alex Bateman. The Pfam protein families database. *Nucleic Acids Res*, 36:281–288, 2008.
- [40] S. Garcia-Vallve, A. Rojas, J. Palau, and A. Romeu. Circular permutants in beta-glucosidases (family 3) within a predicted double-domain topology that includes a (beta/alpha)₈-barrel.. *Proteins*, 31:214–223, 1998.
- [41] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software — Practice and Experience*, 33(11), 2003.

- [42] David Gillman and Michael Sipser. Inference and minimization of hidden Markov chains. In *COLT*, pages 147–158, 1994.
- [43] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *PNAS*, 99(12):7821–7826, June 2002.
- [44] Jens Gregor and Michael G. Thomason. Dynamic programming alignment of sequences representing cyclic patterns. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(2):129–135, 1993.
- [45] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- [46] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997.
- [47] U. Heinemann and M. Hahn. Circular permutation of polypeptide chains: Implications for protein folding and stability. *Prog. Biophys. Mol. Biol.*, 64:121–143, 1995.
- [48] R.N. Horspool. Practical fast searching in strings. 10(6):501–6, 1980.
- [49] M Hu, J Yang, and W Su. Permu-pattern: Discovery of mutable permutation patterns. In *KDD*. ACM, 2008.
- [50] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.
- [51] Costas S. Iliopoulos and M. Sohel Rahman. Indexing circular patterns. In *WALCOM*, pages 46–57, 2008.
- [52] H. Jeong, S. P. Mason, A. L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.
- [53] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts (extended abstract). In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991: Proc. of the 16th International Symposium*, pages 240–248. Springer, Berlin, Heidelberg, 1991.

- [54] Maizel JV Jr and Lenk RP. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proc Natl Acad Sci U S A*, 78(12):7665–7669, 1981.
- [55] Jongsun Jung and Byungkook Lee. Circularly permuted proteins in the protein structure database. *Protein Sci.*, 10(9):1881–1886, 2001.
- [56] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *CPM: 6th Symposium on Combinatorial Pattern Matching*, 1995.
- [57] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
- [58] S. Karlin, G. Ghandour, F. Ost, S. Tavaré, and L.J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proceedings, National Academy of Sciences*, 80(18):5660–5664, 1983.
- [59] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. 31(2):249–260, 1987.
- [60] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In *12th Annual Symposium on Combinatorial Pattern Matching*, 2001.
- [61] Dong Kyue Kim, Jeong Eun Jeon, and Heejin Park. An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In *SPIRE 2004*, 2004.
- [62] Dong Kyue Kim, Minhwan Kim, and Heejin Park. Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays. *Algorithmica*, 2007.
- [63] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005.
- [64] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. 6(2):323–350, 1977.

- [65] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.
- [66] R. Kohli and C. Walsh. Enzymology of acyl chain macrocyclization in natural product biosynthesis. *Chemical Communications*, 3:297–307, 2003.
- [67] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27:557–582, 1998.
- [68] G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.
- [69] Roman A. Laskowski, James D. Watson, and Janet M. Thornton. Profunc: A server for predicting protein function from 3D structure. *Nucleic Acids Research*, 33(Web-Server-Issue):89–93, 2005.
- [70] Roman A. Laskowski, James D. Watson, and Janet M. Thornton. Protein function prediction using local 3D templates. *Journal of Molecular Biology*, 351:614–626, 2005.
- [71] Florencia G. Leonardi. A generalization of the PST algorithm: Modeling the sparse nature of protein sequences. *Bioinformatics*, 22(11):1302–1307, 2006.
- [72] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).
- [73] M. Li, J.H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17:149–154, 2001.
- [74] Jie Lin and Don Adjero. All-against-all circular pattern matching. 2011. Under review.
- [75] Jie Lin and Don Adjero. Circular pattern discovery. *21st International Workshop on Combinatorial Algorithms*, 2011.
- [76] Jie Lin, Don Adjero, and Binghua Jiang. Algorithms for efficient detection of cps in multidomain protein. 2011. To be submitted.

- [77] Jie Lin, Don Adjero, and Binghua Jiang. Probabilistic suffix array: Efficient modelling and prediction of protein families. 2011. Under review.
- [78] Jie Lin, Yue Jiang, and Don Adjero. The virtual suffix tree: An efficient data structure for suffix trees and suffix arrays. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 68–83, Czech Technical University in Prague, Czech Republic, 2008.
- [79] Jie Lin, Yue Jiang, and Don Adjero. The virtual suffix tree. *Int. J. Found. Comput. Sci.*, 20(6):1109–1133, 2009.
- [80] Mortitz G. Maaß. Computing suffix links for suffix trees and arrays. *Information Processing Letters*, 101(6), 2007.
- [81] Maurice Maes. On a cyclic string-to-string correction problem. *Inf. Process. Lett.*, 35(2):73–78, 1990.
- [82] Veli Mäkinen. Compact suffix array – a space-efficient full-text index. *Fundam. Inform.*, 56(1-2):191–210, 2003.
- [83] Veli Mäkinen and Gonzalo Navarro. Compressed compact suffix arrays. *Combinatorial Pattern Matching*, pages 420–433, 2004.
- [84] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.
- [85] Tobias Marschall and Sven Rahmann. Probabilistic arithmetic automata and their application to pattern matching statistics. *Combinatorial Pattern Matching*, pages 95–106, 2008.
- [86] Tobias Marschall and Sven Rahmann. Efficient exact motif discovery. *Bioinformatics*, 25(12), 2009.
- [87] Andres Marzal and Sergio Barrachina. Speeding up the computation of the edit distance for cyclic strings. *Pattern Recognition, Int’l Conference on*, 2:2891, 2000.

- [88] Geoffrey Mazeroff, Jens Gregor, Michael G. Thomason, and Richard Ford. Probabilistic suffix models for API sequence analysis of Windows XP applications. *Pattern Recognition*, 41(1):90–101, 2008.
- [89] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, April 1976.
- [90] R. A. Mollineda, E. Vidal, and F. Casacuberta. Cyclic sequence alignments: Approximate versus optimal techniques. *International Journal of Pattern Recognition and Artificial Intelligence*, 16:291–299, 2002.
- [91] R. A. Mollineda, E. Vidal, and F. Casacuberta. A windowed weighted approach for approximate cyclic string matching. In *ICPR '02: Proceedings of the 16th International Conference on Pattern Recognition (ICPR'02) Volume 4*, page 40188, Washington, DC, USA, 2002. IEEE Computer Society.
- [92] Ramón Alberto Mollineda, Enrique Vidal, and Francisco Casacuberta. Efficient techniques for a very accurate measurement of dissimilarities between cyclic patterns. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*, pages 337–346, London, UK, 2000. Springer-Verlag.
- [93] Krisztián Monostori, Arkady Zaslavsky, and Heinz Schmidt. Suffix vector: Space- and time-efficient alternative to suffix trees. In Michael J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.
- [94] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.
- [95] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. *Proceedings, Combinatorial Pattern Matching, LNCS 1848*, pages 166–180, 2000.
- [96] Saul Ben Needleman and Christian Dennis Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

- [97] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In *CPM*, pages 54–67, 2009.
- [98] Jose Oncina. The Cocke-Younger-Kasami algorithm for cyclic strings. In *ICPR '96: Proceedings of the 13th International Conference on Pattern Recognition*, page 413, Washington, DC, USA, 1996. IEEE Computer Society.
- [99] Hasan H. Otu and Khalid Sayood. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics*, 19(16):2122–2130, November 2003.
- [100] Philipp Pagel, Matthias Oesterheld, Volker Stümpflen, and Dmitrij Frishman. The DIMA web resource - exploring the protein domain network. *Bioinformatics*, 22(8):997–998, 2006.
- [101] Debnath Pal. Inference of protein function from protein structure. *Structure*, 13:121–130, 2005.
- [102] Anatolij Potapov, Björn Goemann, and Edgar Wingender. The pairwise disconnectivity index as a new metric for the topological analysis of regulatory networks. *BMC Bioinformatics*, 9, 2008.
- [103] Elise Prieur and Thierry Lecroq. From suffix trees to suffix vectors. In *Prague Stringology Conference(PCS2005)*, Prague, 2005.
- [104] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.
- [105] A. Reyes, C. Gissi, G. Pesole, F.M. Catzefflis, and C. Saccone. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Mol. Biol. Evol.*, 17:979–983, 2000.
- [106] J. Rissanen. Complexity of strings in the class of Markov sources. *IEEE Transactions on Information Theory*, IT-32(4):526–532, 1986.
- [107] Jorma Rissanen. Universal coding information prediction and estimation. *IEEE Transactions on Information Theory*, IT-30(4):629–636, 1984.

- [108] Dana Ron, Yoram Singer, and Naftali Tishby. Learning probabilistic automata with variable memory length. In *COLT*, pages 35–46, 1994.
- [109] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.
- [110] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully-compressed suffix trees. In *LATIN'08: Proceedings of the 8th Latin American Conference on Theoretical Informatics*, pages 362–373, Berlin, Heidelberg, 2008. Springer-Verlag.
- [111] Kunihiro Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [112] GK Sandve and F Drabls. A survey of motif discovery methods in an integrated framework. *Biology Direct*, 1(11), 2006.
- [113] Thomas B. Sebastian, Philip N. Klein, and Benjamin B. Kimia. On aligning curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:116–125, 2003.
- [114] P.H. Sellers. The theory and computation of evolutionary distances: Pattern Recognition. *jalg*, 1:359–373, 1980.
- [115] Y. Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981.
- [116] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981.
- [117] W.F. Smyth. *Computing Patterns in Strings*. Addison-Wesley, 2003.
- [118] Kenneth Sorensen. Distance measures based on the edit distance for permutation-type representations. *Journal of Heuristics*, 13:35–47, 2007.
- [119] YQ. Tang, J. Yuan, G. Osapay, K. Osapay, D. Tran, CJ. Miller, AJ. Ouellette, and ME. Selsted. A cyclic antimicrobial peptide produced in primate leukocytes by the ligation of two truncated alpha-defensins. *Science*, 286(5439):498–502, 1999.

- [120] Steven L. Tanimoto. A method for detecting structure in polygons. *Pattern Recognition*, 13(6):389–394, 1981.
- [121] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.
- [122] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [123] Shai Uliel, Amit Fliess, Amihod Amir, and Ron Unger. A simple algorithm for detecting circular permutations in proteins. *Bioinformatics*, 15(11):930–936, 1999.
- [124] Shai Uliel, Amit Fliess, and Ron Unger. Naturally occurring circular permutations in proteins. *Protein Eng.*, 14(8):533–542, August 2001.
- [125] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.
- [126] J. Weiner and E. Bornberg-Bauer. Evolution of circular permutations in multidomain proteins. *Mol. Biol. Evol.*, 23(4):734–743, 2006.
- [127] J. Weiner, G. Thomas, and E. Bornberg-Bauer. Rapid motif-based prediction of circular permutations in multi-domain proteins. *Bioinformatics*, 21(7):932–937, 2005.
- [128] P. Weiner. Linear pattern matching algorithm. *Proceedings, 14th IEEE Symposium on Switching and Automata Theory*, 21:1–11, 1973.
- [129] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.
- [130] S. Wu and U. Manber. Agrep — a fast approximate pattern matching tool. In *Proceedings of the Winter 1992 USENIX Conference*, pages 153–62. USENIX Association, Berkeley, CA, 1992.
- [131] Mikio Yamamoto and Kenneth Ward Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, 2001.

- [132] J Yang, W Wang, P Yu, and J Han. Mining long sequential patterns in a noisy environment. In *SIGMOD*. ACM, 2002.
- [133] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [134] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.