WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2008

Direct suffix sorting and its applications

Fei Nan
West Virginia University

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

Recommended Citation
Nan, Fei, "Direct suffix sorting and its applications" (2008). *Graduate Theses, Dissertations, and Problem Reports*. 2692.
https://researchrepository.wvu.edu/etd/2692

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.

Direct Suffix Sorting and Its Applications

Fei Nan

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 Science

Donald Adjeroh, Ph.D., Chair
Bojan Cukic, Ph.D.
Elaine Eschen, Ph.D.
Bing-Hua Jiang, Ph.D.
Amar Mukherjee, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown,West Virginia
2008

Keywords: Suffix Sorting, Suffix Tree, Suffix Array, Data Compression,
Multiple Sequence Alignment, Motif Identification

© 2008, Fei Nan

ABSTRACT

Direct Suffix Sorting and Its Applications

Fei Nan

The suffix sorting problem is to construct the suffix array for an input sequence. Given a sequence $T[0 \ldots n-1]$ of size $n = |T|$, with symbols from a fixed alphabet Σ, ($|\Sigma| \leq n$), the suffix array provides a compact representation of all the suffixes of T in a lexicographic order. Traditionally, the suffix array is often constructed by first building the suffix tree for T, and then performing an inorder traversal of the suffix tree. The direct suffix sorting problem is to construct the suffix array of T directly *without* using the suffix tree data structure. We propose a direct suffix sorting algorithm which rearranges the biological sequences of interests and facilitates high throughput pattern query, retrieval and storage in $O(n)$ time. The improved algorithm requires only $7n$ bytes of storage, including the n bytes for the original string, and the $4n$ bytes for the suffix array. The basis of our improved algorithm is an extension of Shannon-Fano-Elias codes used in information theory. This is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem.

The direct suffix sorting algorithm is then applied to solve the multiple sequence alignment problem. The sequences to be aligned are concatenated and then passed to the suffix sorting algorithm to locate the repetitive regions. Gaps are determined by those repetitive regions and consequently inserted back to the input sequences to attain the optimal alignment with reference to the biological mutation matrix.

We also apply the direct suffix sorting algorithm to the problem of compressibility of protein sequences. The basis is the observation of genome-scale long-range correlation in concatenated protein sequences from the same organism. We propose a method to exploit this unusual redundancy in compressing the protein sequences. The result is a significant reduction in the number of bits required for representing the sequences. The observed long-range correlations could have significant implications beyond compression and complexity analysis of protein sequences.

Acknowledgments

I would like to take this great opportunity to thank my supervisor, Dr. Donald Adjeroh, for his endless and selfless support, guidance, insights and advice during the hard times, giving me constant encouragement. I would like to express my most profound gratitude to members of my advisory committee members, Dr. Donald Adjeroh, Dr. Bojan Cukic, Dr. Elaine Eschen, Dr. Bing-Hua Jiang and Dr. Amar Mukherjee, for their great scientific insights, inspiration and patience to keep me on the right track. I would like to thank Yong Zhang for his time and constructive criticism in my project, which has helped develop my critical thinking. I would like to thank Jianan Feng for his help and advice on experimental data as well as with presentation of my data. I would like to thank David Woo and Thomas Wessel for their open-minded suggestions and discussions, which make everyday an enjoyable day. I would also like to thank all people who helped me throughout my study.

Contents

Abstract ii

Acknowledgments iii

List of Tables viii

List of Figures xii

1 Introduction 1

1.1 Introduction 1

1.2 The Problem 2

1.2.1 Biological Domains 3

1.3 Research Motivation 7

1.4 Contributions 8

1.5 Dissertation Organization 9

2 Literature Review 11

2.1 Biological Sequences and Redundancy 11

2.2 Suffix Trees 12

2.3 Suffix Arrays 13

2.4 Suffix Sorting and the BWT 15

2.5 Direct Suffix Sorting 16

2.6 Sequence Alignment 18

2.7 Protein Sequence Compression 19

3 Direct Suffix Sorting 22
3.1 Overview . . . 22
3.2 Algorithm I: Basic Algorithm . . . 23
3.2.1 Notations . . . 23
3.2.2 Overview of Algorithm . . . 23
3.2.3 Divide T . . . 24
3.2.4 Merge SA for T_0 and T_1 . . . 25
3.2.5 Recursive Call . . . 26
3.2.6 Conflict Resolution . . . 26
3.2.7 Complexity Analysis . . . 30
3.3 Algorithm II: Improved Algorithm . . . 32
3.3.1 Overview . . . 32
3.3.2 Sort T_1 to Form SA_1 . . . 32
3.3.3 Sort T_0 to form SA_0 using SA_1 . . . 33
3.3.4 Improved Sorting . . . 35
3.4 Algorithm III: Generalized Case $1 : \eta$ Scheme . . . 42
3.4.1 Overview . . . 42
3.4.2 Partition Tree Height . . . 43
3.4.3 Conflict Resolution . . . 44
3.5 Concluding Remarks . . . 46

4 Multiple Sequence Alignment 48
4.1 Overview . . . 48
4.1.1 Anchor Based Methods . . . 49
4.1.2 The Problem and Main Contribution . . . 52
4.2 Sort-based Alignment Algorithm . . . 53
4.2.1 Notation . . . 54
4.2.2 Overview . . . 55
4.2.3 Anchor Points Selection . . . 56
4.3 Mutation Mapping . . . 61
4.3.1 Motivation . . . 61

4.3.2 Computing the Mutation Map 62
4.3.3 Mutation Procedure 63
4.3.4 Gap Handling 65
4.4 Complexity Analysis 66
4.5 Results 67
4.6 Concluding Remarks 69

5 Repetitive Structures and Data Compression 71
5.1 Overview 71
5.1.1 Challenge of Protein Sequence Compression 71
5.1.2 Compressibility of Protein Sequences 73
5.2 Long-Range Correlation in Protein Sequences 75
5.2.1 Efficient Analysis of Sequence Correlation 77
5.3 Compressing Protein Sequences 79
5.3.1 Parsing and Encoding 80
5.3.2 Multi-level Hierarchical Decomposition 83
5.3.3 Results 85
5.4 Concluding Remarks 86

6 Discussion 87
6.1 Summary 87
6.2 Contributions 89

A List of Publications 91

References 92

List of Tables

2.1 A typical suffix array representation of the string *BANANA\$*. The suffix array is listed in the leftmost column. 14

2.2 A typical Burrows-Wheeler Transform of the string *BANANA\$*. The leftmost column indicates the input to Burrows-Wheeler Transform. The rightmost column indicates the output of Burrows-Wheeler Transform. . 15

3.1 Conflict pairs for $T = aaaabaaaabxaaaab$. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p. 29

3.2 Conflict pairs for $T = aaaaaaaaaaaaaaaa = a^{16}$. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p. 30

3.3 Upper and lower bounds on the current interval on the number line for each individual step in Figure 3.5 and Figure 3.6. 40

3.4 The merge routine for ternary partition scheme at level 2. Every *Pos* share the same modulus. 46

3.5 The merge routine for ternary partition scheme at level 3. Every *Pos* share the same modulus. 46

4.1 The proposed algorithm introduces gaps to align two anchor points which were originally on the same vertical location on the backbone sequence. . 66

4.2 Pairwise protein sequences of enteropathogenic E. Coli (EPEC) in FASTA format to be aligned. 67
4.3 Final alignment results by proposed algorithm on pairwise protein sequence alignment of enteropathogenic E. Coli (EPEC). 67
4.4 Final alignment results by DiAlign [1,2] and Clustal W [3,4] on pairwise protein sequence alignment of enteropathogenic E. Coli (EPEC). 68
4.5 Multiple protein sequences of enteropathogenic E. Coli (EPEC) in FASTA format to be aligned. 69
4.6 Final alignment results by proposed algorithm on multiple protein sequence alignment of enteropathogenic E. Coli (EPEC). 69

5.1 Failure of general compression algorithms on protein sequences. Results in bits/symbol (smaller values imply better performance). 74
5.2 Failure of biological sequence compression algorithms on protein sequences. Results in bits/symbol (smaller values imply better performance). 74
5.3 SCP statistical information for six common DNA sequences of humEpsBarr, HUMGHCSA, MitoMPOMTCG, YSCCHRIII, YSCCHRIV and E. Coli. 77
5.4 SCP statistical information for four concatenated DNA sequences of H. Influenzae, M. Jannaschii, S. Cerevisiae and H. Sapiens. 77
5.5 Different types of tandem repeats. $t(r) = 1$ indicates direct repeat. $t(r) = 2$ indicates reverse repeat. $t(r) = 3$ indicates complemented palindrome. . 82
5.6 Statistics of maximal repeats and compression results using the observed long-range correlations. Compression results in bits/symbol (small values imply better performance). 85
5.7 Comparative compression performance using the observed long-range correlations. Compression results in bits/symbol (small values imply better performance). 85

List of Figures

1.1 Growth of Genbank: The Human Genome Project was undertaken in 1990, which exponentially expanded the size of GenBank over the past decades due to widespread international cooperation and advances in sequence analysis and computing technology. 4

2.1 A typical suffix tree representation of the string *BANANA$* with suffix links. The suffix links are indicated by the dashed lines. 12

3.1 Basic working of proposed algorithm using an example sequence $T = aaaabaaaabxaaaab$. Solid arrows indicate partition procedure. Dotted arrows indicate trivial sort procedure. Dashed arrows indicate merge procedure. 23

3.2 Conflict tree for an example sequence. The original sequence is indicated at the root node. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p. 27

3.3 Improved Algorithm II: Asymmetric recursive partitioning for improved algorithm, using $T = a^{64}$. Dotted arrows (from right to left) denote propagation of the sorted array at the given level of recursion. 33

3.4 Improved Algorithm II: Code assignment by successive partitioning of a number line. Code 021 is selected by following the path of 0, 2 and 1. . . 34

3.5 Code assignment procedure, using an example sequence: $Q_i = acabd$. The vertical line represents the current state of the number line. The current interval at each step in the procedure is shown with a darker shade. The symbol considered at each step is listed under its corresponding number line. 37
3.6 Evolution of code assignment procedure, after removing the first symbol in the previous sequence *acabd*, and bringing in a new symbol *a* to form a new sequence: *cabda*. 38
3.7 Ternary 1 : 2 partition tree with $\eta = 2$. Every $(1 + \eta)$-th symbols are selected to create T_0. Solid arrows indicate partition procedure. 44
3.8 Quaternary 1 : 3 partition tree with $\eta = 2$. Every $(1 + \eta)$-th symbols are selected to create T_0. Solid arrows indicate partition procedure. 45

4.1 Schematic overview of anchor points along the sequences to be aligned. The horizontal lines represent the source sequences to be aligned. The vertical solid lines represent higher priority anchors with stronger constraints across different sequences. The dashed lines represent lower priority anchors with weaker constraints. The intersection between horizontal and vertical lines represent the prioritized anchor points, which play an essential role in multiple sequence alignment. 50
4.2 A set of four DNA-binding protein residues Π using the same data set used in [5]. They are concatenated to form a single sequence of T by appending T_{i+1} to the end of T_i for each i, where $0 \leq i \leq (m-1)$. 57
4.3 The N-block for sequences used in Figure 4.2. (a) The panel table is sorted suffix according to their alphabetical order. (b) The panel table is sorted by their position k. After the sorted symbols are obtained, a sliding window of size Q is scanned vertically down the sequence segments. For each sliding window, the number of distinct symbols across sequence segments are computed, which is used to compute the anchor score for the corresponding anchor points. \$ is the end-of-sequence marker. 58

4.4 Suffix sorting routine. The direct suffix sorting is applied on the source sequences. The strongest anchor points $B - A$ and $A - C$ are identified. The source sequences are fixed by the strongest anchor points. $\varphi(h) = 3$ for the left most column. $\varphi(h) = 2$ for the central column. $\varphi(h) = 1$ for the right most column. 60
4.5 Biological mutation graph for PAM250 matrix [6]. More lines between symbols imply a stronger probability of mutation from one symbol to the other. 61
4.6 Biological mutation graph for BLOSUM62 matrix [7]. More lines between symbols imply a stronger probability of mutation from one symbol to the other. 62
4.7 κ mutation table for BLOSUM62 matrix. κ group is defined to contain all mutated vertices where there are K parallel edges between each other, i.e., $\kappa = 3$ group lists all the mutated vertices where there are 3 parallel edges between them in Figure 4.6. The κ grouping table is to be used in the mutation-based suffix sorting routine and depends on the biological mutation table only and it is independent of the input source sequences. It will be produced once during the alignment procedure, or can be pre-stored before the alignment starts. 63
4.8 Intermediate mutation based suffix sorting alignment results between $B - A$ anchor point. The biological mutation matrix mapping is performed in a descending order of κ. 64
4.9 Intermediate mutation based suffix sorting alignment results between $A - C$ anchor point. The biological mutation matrix mapping is performed in a descending order of κ. 65
4.10 The final concatenated results with biological mutation matrix information. Notice how the current method aligns more biologically similar protein on the same column. 65

5.1 Sorted probabilities and higher order entropy. The protein sequences are taken from four organisms: HI: H. influenzae; MJ: M.jannaschii; HS: H. sapiens; SC: S. cerevisiae. 72

5.2 Sorted probabilities for the four sequences in the protein corpus. The protein sequences are taken from four organisms: HI: H. influenzae; MJ: M.jannaschii; HS: H. sapiens; SC: S. cerevisiae. 73

5.3 Different forms of repetitions observed in the protein sequences, as exposed by the SCP. A: Overlapping repeats (pattern S_1 occurred 5 times). B: One subsequence covers the other subsequence. We break them down into smaller sequences S_1. C: Repeated sequences are separated by possibly long stretches of amino acids between them. This is the most common case. D: The triple overlap case. We break the overlaps down into subsequence S_1. 79

5.4 Flowchart for proposed compression algorithm. Dictionary and remaining sequences are recursively passed to the compression program until the overall compression gain is not significant. 80

5.5 Multi-level hierarchical decomposition. Dictionary and remaining sequences are recursively passed to the compression program until the overall compression gain is not significant. 84

Chapter 1

Introduction

1.1 Introduction

Pattern matching plays a central role in different aspects of biological sequence analysis, and has been used in applications as diverse as short-gun sequencing, multiple sequence alignments, gene finding [8], analysis of repetition structures [9], searching for unique oligonucleotides [10], prediction of protein function and structure, sequence homology search, finding DNA-binding protein motifs [11], identifying single nucleotide polymorphisms (SNPs), identification of pseudo-genes via cDNA matching, etc. The popularity of search tools such as BLAST [12] and FASTA [13] suite of programs is a further testament of the importance of both theoretical algorithm and practical pattern matching applications in biosequence analysis.

Suffix trees and suffix arrays are primary data structures used in rapid pattern matching. They have also found applications in various search-based problems in computational biology [14–16]. The major motivation in their use is their theoretical time and space complexity, and the logarithmic search performance that becomes possible after their construction. The suffix array is used more often than suffix trees in some pattern matching applications due to its smaller memory footprint. The problem of suffix sorting is a fundamental problem in constructing suffix arrays. There is a great interest in constructing efficient algorithms to build the suffix array, and hence improve analysis of biological sequences.

The suffix array and suffix tree data structures also have significant applications in repetitive motif identification and protein sequences compression for biological data. To our knowledge, the function of the repetitive non-coding areas are not completely understood. Although the major attention has been on the coding areas, it has long been shown that the non-coding areas could be performing some important function [17, 18], and hence should not be ignored. Repetition structures represent an important characteristic of genomic sequences. Although the precise biological function of these repetition structures is still a topic of intensive debate, it is, however, well known that the redundancy due to the repetition structures provides some form of stability for the genome. Tandem repeats in particular play a major role in various regulation mechanisms in the genome, such as in protein binding [19]. They have also been linked with different recombination hot spots [20]. Repetition structures have been implicated in various diseases and genetic disorders. For instance, the triplet repeats $(CTG)_n/(CAG)_n$ have been associated with the Huntington's disease, while the hairpins formed in $(CGG)_n/(CCG)_n$ repeats have been linked to the Fragile-X mental retardation syndrome [21]. Sinden et al [22] identified fourteen of such genetic diseases that are linked with triplet repeats. An important observation for computational analysis of such repetition structures is that, in every single case listed, the susceptibility to (or incidence of) the disease critically depends on the number of copies, i.e., the copy exponents in the repeat and *how many* times the triple repeat occurs with a given exponent. Identifying such copy exponents or the number of times a given structure is repeated in a sequence, or over a large database of sequences, therefore, represent important problems in computational biology and bioinformatics.

1.2 The Problem

One of the inherent disadvantages of those existing pattern matching algorithms, either keyword trees or suffix trees, is the usage of complicated tree data structures. The suffix array data structure proposed in [23] is very closely related to the suffix tree and with some extra data structures can be considered equivalent to the suffix tree to a large extent. The process of building a suffix array is called suffix sorting. Considering the

complication of the suffix tree based multiple pattern matching algorithm, we propose a direct suffix sorting algorithm. The constructed suffix array can locate all the occurrences of a pattern string in a string of texts. It is array based, and relies on simple memory allocation strategies for successful implementation. The direct suffix sorting algorithm is designed with easier programming paradigm compared with the suffix tree, such that to answer questions whether a set of words "shore", "she" and "sea" appear in the sentence "She sells sea shells by the sea shore", requires $O(m+n)$ time overall. The dissertation will elaborate on the direct suffix sorting algorithm.

The applications of the direct suffix sorting algorithm are not merely to answer multiple string pattern matching queries. It can also be used to address the problems of multiple sequence alignment and data compression in the biological domain.

1.2.1 Biological Domains

Background

Proteins are large organic compounds made of amino acids arranged in a linear chain and joined together by peptide bonds between the carboxyl and amino groups of adjacent amino acid residues. They are large molecules and are responsible for differing functions in an organism. There are different types of protein, and the exact function of a given protein depends on its three dimensional structure. Proteins are made up of amino acids, which are often joined in chains to form polypeptides or simply peptides for smaller sequences. Deoxyribonucleic acids, or DNA, are antiparallel strands of double helical molecules, which contain genetic instructions used in the development and functioning of all known living organisms. Proteins are constructed from an alphabet of 20 amino acids. Each amino acid has a corresponding DNA triple, taken from the 4-symbol DNA or RNA alphabet. According to the Central Dogma of molecular biology [24], the information flow is unidirectional: from DNA to RNA to Proteins. That is, the exact sequence of amino acids in a given protein sequence is determined by the primary DNA sequence of the genes that produced the protein. The exact rules used in this mapping from DNA bases to amino acids is defined in the Genetic Code [17]. The genetic code is a non-overlapping block code, whereby three successive RNA nucleotides code for one amino

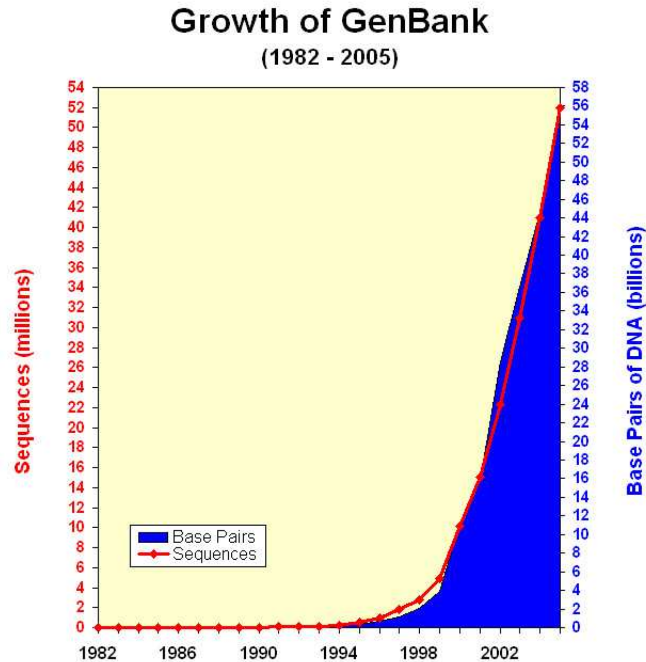

Figure 1.1: Growth of Genbank: The Human Genome Project was undertaken in 1990, which exponentially expanded the size of GenBank over the past decades due to widespread international cooperation and advances in sequence analysis and computing technology.

acid, or for a stop translation signal. The areas of the DNA that contain genes are thus called coding regions, while the remaining parts are called non-coding regions.

It's has been the most challenging task to uncover the coding messages behind DNA and proteins for scientists all over the world. Technology advancements over the past decades have raised the necessity of efficient storage, quick retrieval and effective management. The major aim of the human genome project was to determine more than 3 billion base pairs in the human genome with a minimal error rate as well as to identify all the genes in this vast amount of data. Meanwhile, it dramatically raised the demand to develop faster, more efficient methods for DNA mining and sequence management.

Multiple Sequence Alignment

In the research area of computational biology, sequence alignment is a method to derive the arrangement of the sequence for DNA, RNA and protein sequence on the symbol-wise base-pair comparison basis so that the maximum similarity between sequences can be discovered. The biological objective is to identify the functional, systematic and phylogenetic significance. Under most circumstances, a matching equation with some data score value is used to measure how well two or more residue sequences can be aligned among them. The matching equation is related to the total occurrence of mismatch symbols or match symbols either on a percentage basis or an absolute score value with specific emphasis on special matching or mismatching. Gaps are defined as the blank spaces inserted between the nucleotide or amino acid residues to justify or align them with optimal matching score value. There are several existing approaches to solve either the global sequence alignment problem or the local sequence alignment problem. Theoretically speaking, to identify an optimal global sequence alignment is an NP-Complete problem. However, giving some assumptions, good approximation to the optimal solution can be achieved by converting the sequence alignment problem to some equivalent solvable problems.

Data Compression

Given the Central Dogma, we can say that the protein sequence, at least at the primary sequence level, is a function of the original DNA sequence. From information theory, we know that the entropy of a function of a random variable cannot be any greater than the entropy of the original random variable. Thus, the protein sequence cannot have more entropy than the original DNA sequence from which it is formed [17]. But this is only with respect to the coding regions of the DNA sequence. Further, with the assumption that organisms are purposeful, and not just a random collection of symbols, we can expect that there will be constraints placed on how the DNA sequences (hence protein sequences) are ordered. Such constraints are likely to lead to some form of redundancy, which could be exploited for compression.

We also expect that nature will find a way to protect important genes or gene products

in a given organism. This means that there is likely to be some other form of redundancy in biological sequences, at least, for the purpose of more reliable transmission of genetic information from one generation to the other. The various forms of repetitions in biological sequences could be one way to ensure this reliability. More importantly, the phenomenon of gene duplications, multiple gene copies, and histone clusters in genomes of higher organisms imply that their protein sequences (at least, at the primary sequence level) will have some redundancy. For instance, [25] reports of 1509 duplicate paralogous genes from 5766 yeast open reading frames for baker's yeast, S. Cerevisiae. Similar observations on the extent of duplicate genes in other model organisms were reported in [26]. A number of experiments have shown that the inactivation of certain genes has no apparent phenotypical effect on the fitness of certain species [25,27–30]. This observation relates to the functional redundancy of genes in different organisms, and it has been observed that most of these functionally similar genes tend to be quite similar at both the transcription level and at the sequence level [31]. We believe biological sequences can be compressed because of the significant redundancy in such sequences.

Redundancy and Genetics

Genetics is scientific study of the mechanism of heredity. While Gregor Mendel first presented his findings on the statistical laws governing the transmission of certain traits from generation to generation in 1856, it was not until the discovery and detailed study of the chromosome and the gene in the 20-th century when scientists found the physical basis of hereditary characteristics. Redundancy between two genes occurs only with respect to a given function like the metabolic system or the immune system etc. Those genes are maintained by selection because of another independent function. Thus, a functional overlap gene segment is produced. The ability of those genetic techniques to operate both within and between chromosomes implies that realistic models of the evolutionary dynamics of redundancy, and of the potential interaction with natural selection in a sexual species, need to consider the diffusion of various repeats across multiple chromosome lineages, in a population context.

Nowak et al [30] suggests that redundancy is wide spread in genome or higher organism. They are performing the same function and the inactivation of one of these genes has little or no effect on the biological phenotype. They present 4 cases that could explain

why the genetic redundancy is common. They also define 3 types of genetic redundancy within an organism.

1. “True Redundancy” denotes an individual with a redundant genotype is not fitter than the one with knocked out genes.

2. “Generic Redundancy” denotes the occasional more fitness.

3. “Almost Redundancy” denotes an individual with a redundant genotype is always fitter than the one with knocked out genes.

A repetition of an evolution experiment always leads to different sequences of genotypes and phenotypes on the way from initial structure to target. But there are also conserved and hence reproducible features.

Genetic drift is the random fluctuations in the frequency of the appearance of a gene in a small isolated population, presumably owing to chance rather than natural selection. It’s resulting mainly from chance mating. In small breeding populations an entire generation might, by chance alone, be born with the same genotype with respect to a particular allelic pair of genes, thus leading to either the elimination or dominance of a particular gene. Because fluctuations in the proportions of alleles are more significant in the gene pools of small, isolated breeding populations, genetic drift is a unavoidable factor of producing the redundancy in the phylogenetic evolution.

1.3 Research Motivation

We propose a direct suffix sorting algorithm which rearranges the biological sequences of interests and facilitate high throughput pattern query, retrieval and storage in $O(n)$ time. Then, we apply the direct suffix sorting algorithm to solve practical problems in multiple sequence alignment and data compression.

We consider sequence alignment as a way to derive the arrangement of a set of biological sequences in the form of symbol-wise rows within a matrix such that the maximum similarity between sequences can be exploited. Instead of shifting symbols back and forward, we consider symbols are fixed and shift gaps around as the blank spaces inserted

between nucleotide or amino acid residues to justify or align them to achieve the best matching score value. The biological significance is the possibility of identifying regions with related or similar functional, systematic and phylogenetic characteristics.

In our view, one major problem with current approaches to protein sequence compression in particular, and biological sequence compression in general is the focus on statistical compression methods. As seen from the practical results from existing algorithms, simple Markovian models clearly are not adequate. The statistical significance of repetitions may not be enough to build an adequate model for the symbol probabilities. Further, most methods seem to perform simple pattern searches, while biological sequences are known to have various types of repetition, whether one considers approximate or exact repetition. More importantly, the methods tend to ignore the fact that biological phenomena such as gene duplication could lead to long-range genome-wide correlation in protein sequences, with possible large-scale duplications occurring between different chromosomes. Our objective is to exploit these genome-wide long-range correlations in protein sequences.

1.4 Contributions

We propose a linear time direct suffix sorting algorithm to construct the suffix array data structure. It features easy implementation and does not employ the intermediate suffix tree data structure. We address an open problem in suffix sorting [32] by reducing the gap between actual space requirement of current suffix sorting algorithms, and the minimal requirement of $5n$ bytes for storing both the original string, and its suffix array. We propose a divide-and-conquer sort-and-merge algorithm for performing direct suffix sorting on a given input string. Given a string of length n, our algorithm runs in $O(n)$ worst case time and space. The algorithm recursively divides an input sequence into two parts, performs suffix sorting on the first part, then sorts the second part based on the sorted suffix from the first. It then merges the two smaller sorted suffixes to provide the final sorted array. Our algorithm differs from previous approaches in the use of a simple partitioning step, and how it exploits this simple partitioning scheme for conflict resolution. The space requirement for the proposed algorithm is $7n$ bytes, including the

n bytes required to store the original string and the $4n$ bytes needed for the suffix array. This is a significant improvement when compared with the $13n$ bytes required by the KS algorithm [33]. The method is also unique in its use of Shannon-Fano-Elias codes in efficient construction of a global partial order for the suffixes. To our knowledge, this is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem.

The direct suffix sorting algorithm is then applied to solve the multiple sequence alignment problem with reference to given biological mutation matrix. The sequences to be aligned are concatenated and then passed to the suffix sorting algorithm to locate the repetitive regions. Gaps are determined by those repetitive region and consequently inserted back to the input sequences to attain the optimal alignment. The protein symbols are mutated according to the biological mutation matrix to reflect their biological distance from each other. Individual protein symbols are not considered as equal. The incorporation of biological mutation matrix is believed to better exploit the inter-protein closeness and thus produces a better alignment.

We also apply the direct suffix sorting algorithm to the problem of compressibility of protein sequences based on the observation of genome-scale long-range correlation in concatenated protein sequences from different organisms. We propose a method to exploit this unusual redundancy in compressing the protein sequences. The result is a significant reduction in the number of bits required for representing the sequences. We report results in bits per symbol (bps) of 2.27, 2.55, 3.11 and 3.44 for protein sequences from *M. Jannaschii*, *H. Influenzae*, *S. Cerevisiae*, and *H. Sapiens* respectively, the same protein sequences used by Nevill-Manning and Witten in the "Protein is incompressible" paper [34]. The observed long-range correlations could have significant implications beyond compression and complexity analysis of protein sequences.

1.5 Dissertation Organization

The dissertation is organized as follows. Chapter 2 reviews the literature and related work. We describe the relationship between suffix trees and suffix arrays, as well as their respective time and space complexity. Chapter 3 presents our primary methodology. We

describe the systematic approach to obtain the suffix array data structure directly. We give the proof for the correctness, followed by time and space complexity analysis. Chapter 4 focuses on multiple sequence alignment with reference to given biological mutation matrix. Chapter 5 presents another application to repetitive pattern identification and data compression. We conclude with a summary of the dissertation and future directions in Chapter 6.

Chapter 2

Literature Review

2.1 Biological Sequences and Redundancy

Redundancy exists in every aspect of our life and challenges our work in different forms, including but not limited to repetition, correlation, approximation and symmetry. Repetition is the character-wise symbol-by-symbol redundancy. It is the most simple form of redundancy we will deal with in this dissertation. A replacement with a fixed entry in the dictionary, either online or offline, can shrink the size. Symmetry is another type of redundancy which differs from repetition by a mapping correspondence in relative position of symbols on opposite sides of a center or axis point. The correlation redundancy is a measure of the mutual information or a normalized variant between two variables. Total correlation gauges the redundancy among multiple variables. Correlation includes a wide collection of repetitions, among which tandem repeats are of great interests to biologists. Tandem repeats are universal phenomena. Sequences with the highest degree of polymorphism are very useful for DNA analysis in forensics cases and paternity testing. Scientists found chimpanzee and human genes are 99 percent similar. This type of similarity is not straightforward repetition. The difference is distributed sporadically along the whole genome. At any segment of the chromosome, the majority of the sequences are identical but differences do exist. We call this type of redundancy approximate redundancy, since the repetitions involved are not necessarily exact. We believe there is still a significant amount of information hidden behind the approximate redundancy.

2.2 Suffix Trees

The suffix tree was first introduced by Weiner [35] in 1973 as a way to present the suffixes of a given string and facilitate fast implementation of string operations. McCreight [36] greatly improved the construction algorithm in 1976. Ukkonen [37] was the first to propose a linear-time algorithm for online-construction of suffix trees in 1995. Suffix trees have found numerous applications in rapid pattern matching and computational biology [14].

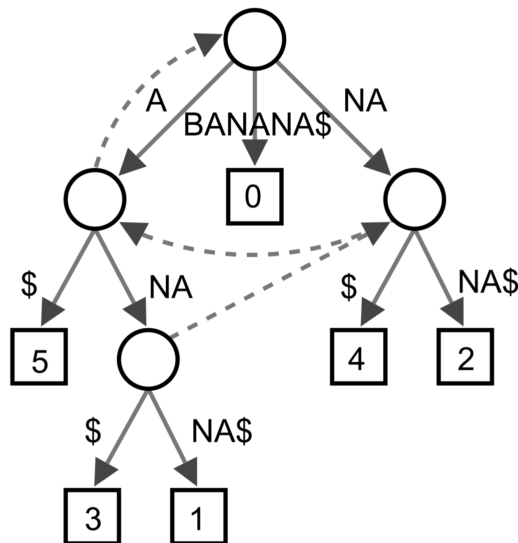

Figure 2.1: A typical suffix tree representation of the string *BANANA$* with suffix links. The suffix links are indicated by the dashed lines.

Given a string $T[0...n-1]$ of length n, a suffix tree is a rooted tree with n leaves, whereby the i-th leaf node corresponds to the suffix $T[i...n-1]$. Each edge in the tree is a substring, and no two edges out of a node start with the same character. A typical suffix tree for string *BANANA$* is shown in Figure 2.1. Ukkonen's algorithm makes a heavy use of suffix links. In a complete suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string $\alpha\beta$, where α is a single character and β is a string, possibly empty, the suffix link from the node will point to the internal node representing β. For examples, in Figure 2.1, the suffix link from the node for ANA goes to the node for NA. Following Ukkonen's algorithm,

the suffix tree can be constructed in linear time. After that, many string operations can be performed quickly, such as locating a substring in the given string. The suffix tree presents an elegant solution to the longest common substring problem but at the expense of the storage space. Storing the suffix tree takes significantly more space than storing the string itself.

The suffix tree for the string T of length n is defined by Gusfield [14] as a tree such that

1. The paths from the root to the leaves have a one-to-one relationship with the suffixes of T.

2. Edges spell non-empty strings.

3. All internal nodes (except perhaps the root) have at least two children.

There is a concern that such a tree does not exist for all strings. Thus, T is padded with a terminal symbol not seen in the string, usually denoted \$. This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of T. Since all internal non-root nodes are branching, there can be at most $n-1$ such nodes, and $n + (n-1) + 1 = 2n$ nodes in total.

Suffix trees have many applications [14] in bioinformatics, where they are used for searching for patterns in DNA or protein sequences, which can be viewed as long strings of characters. Gusfield [14] provides a detailed study on the use of suffix trees in the analysis of biological sequences. The ability to search efficiently with mismatches might be the suffix tree's greatest strength. It is also used in data compression, where on the one hand it is used to find repeated data and on the other hand it can be used for the sorting stage of the Burrows-Wheeler Transform (BWT). Variants of the *LZW* compression schemes also use the suffix tree.

2.3 Suffix Arrays

The common suffix tree implementation will involve parent-child relationships between nodes. The typical representation is with linked lists. Each node has pointer to its first

child, and to the next node in the child list it is a part of. A close analysis of the linked lists implementation of suffix tree will show that the large amount of information in each edge and node makes the suffix tree very expensive, consuming about ten to twenty times the memory size of the source text in good implementations.

To solve the space consumption problem, Manber and Myers [23] were the first to propose the suffix array as a new and conceptually simple data structure for online string searching. They suggest an $O(n \log n)$ algorithm to construct the suffix array with three to five times less space than the traditional suffix tree. The suffix array provides a compact representation of the list of all suffixes of a given string in a lexicographic order. The sequence of positions in T of the first symbols from the sorted suffixes in this lexicographic order is the suffix array for the sequence.

Suffix Arrays	Suffixes
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Table 2.1: A typical suffix array representation of the string *BANANA$*. The suffix array is listed in the leftmost column.

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of these indices in lexicographic order. Table 2.1 shows the sorted suffixes for the string “BANANA$”, the same example in Figure 2.1. Using one-based indexing, the suffix array is,

$$[6, 5, 3, 1, 0, 4, 2]$$

The suffix “A” begins at the 5th character. “ANANA” begins at the 1st character, and so forth.

The suffix array of a string can be used as an index to quickly locate every occurrence of a substring within the string. Finding every occurrence of the substring is equivalent

to finding every suffix that begins with the substring. With the lexicographic ordering, these suffixes will be grouped together in the suffix array, and can be found efficiently using binary search. Given the suffix array, the array of longest common prefix (lcp) can easily be constructed in linear time.

The suffix array reduces the space requirement by a factor of four, and researchers have continued to find smaller indexing structures. Suffix trees and suffix arrays have drawn a significant attention in recent years due to their theoretical linear time and linear space construction, and their logarithmic search performance. As a result, the suffix array data structure began the trend towards compressed suffix arrays and BWT-based compressed full-text indices.

2.4 Suffix Sorting and the BWT

The suffix array is a primary data structure in our approach to implement multiple sequence alignment and biological data compression. Suffix trees [35,36] and suffix arrays [23] are primary data structures that have found applications in rapid pattern matching and computational biology [14]. Suffix sorting is also an important problem [38–40] in data compression, especially for compression schemes that are based on the Burrows-Wheeler Transform (BWT) [41].

Burrows-Wheeler Transform

Input	All Rotations	Sort the Rows	Output
	BANANA$	$BANANA	
	$BANANA	A$BANAN	
	A$BANAN	ANA$BAN	
BANANA$	NA$BANA	ANANA$B	ANNB$AA
	ANA$BAN	BANANA$	
	NANA$BA	NA$BANA	
	ANANA$B	NANA$BA	

Table 2.2: A typical Burrows-Wheeler Transform of the string *BANANA$*. The leftmost column indicates the input to Burrows-Wheeler Transform. The rightmost column indicates the output of Burrows-Wheeler Transform.

Given a sequence T of length n, the Burrows-Wheeler Transform proposes a sorting

of all the n cyclic rotations of T in a lexicographic order. The sequence of positions in T of the first symbols from the sorted suffixes in this lexicographic order is the suffix array for the sequence. Taking the same example in Figure 2.1, for the string "BANANA$", the Burrows-Wheeler Transform result is *ANNB$AA* as indicated in Table 2.2.

Suffix sorting algorithms can be used to perform the Burrows-Wheeler transform (BWT). Technically the BWT requires sorting cyclic permutations of a string, not suffixes. This is fixed by appending to the string a special end-of-string character which sorts lexicographically before every other character. Sorting cyclic permutations is then equivalent to sorting suffixes. Thus, the problem of suffix sorting is sometimes viewed as being equivalent to that of constructing suffix arrays. In fact, it is known that the suffix sorting stage is a major bottleneck in BWT-based compression schemes [41, 42]. The suffix array is sometimes favored over suffix trees due to its smaller memory footprint. Suffix sorting is a fundamental problem in constructing suffix arrays.

The traditional method of obtaining the suffix array, or suffix sorting, is to construct the corresponding suffix tree first. Suffix tree is a mature data structure to hold all the suffixes for a given input string. All the suffixes are saved in an order of the suffix tree such that the retrieval of any specific suffix is in constant time. A depth first search on the suffix tree will print out all the leaf nodes in such a way that the order of printing will match the corresponding suffix array exactly. This observation has lead to many applications in obtaining the suffix array by first constructing the suffix tree for a given string and then performing the depth first search on the suffix tree to print the suffix array. This is an indirect method of obtaining the suffix array, or suffix sorting. There is thus a significant interest in designing an algorithm to derive the suffix array data structure directly.

2.5 Direct Suffix Sorting

Most of the methods that attempted to construct suffix arrays have done so by first constructing the suffix tree, and then building the suffix array by performing an inorder traversal of the suffix tree. Other methods for fast suffix sorting in $O(n \log n)$ time have been reported in [43], while memory efficient constructions were considered in [32].

Puglusi et al [44] provide a short comparison of different recently proposed linear time algorithms for suffix sorting. Farach et al [45,46] proposed a divide and conquer method to construct the suffix tree for a given sequence in linear time. The basic idea is to divide the sequence into odd and even sequences, based on the position of the symbols. Then, the suffix tree is constructed recursively for the odd sequence. Using the suffix tree for the odd sequence, they construct the suffix tree for the even sequence. The final step merges the suffix tree from the odd and even sequences into one suffix tree using a coupled depth-first search. The result is a linear time algorithm for suffix tree construction.

Given the memory requirement and implementation difficulty of the suffix tree, it is desirable to construct the suffix array directly, without using the suffix tree. Also, for certain applications where only the suffix array is needed, avoiding the construction of the suffix tree will have some advantages. More importantly, direct suffix sorting without the suffix tree raises some interesting algorithmic challenges. Thus, more recently, methods have been proposed to construct the suffix array directly from the sequence [33,47,48], without the need for a suffix tree.

Kim et al [47] followed an approach similar to Farach's above, but for the purpose of constructing the suffix array directly. They introduce the notion of equivalent classes between strings, which they use to perform the coupled depth-first searches at the merging stage. Let $T_i' = t_i t_{i+1} t_{i+2} \ldots t_{n-1}$ denote the suffix T_i' of sequence T starting at position i. They define the equivalence class as follows [47]: $E_l = \{(i,j)|\text{pref}_l(T_i') = \text{pref}_l(T_j')\}$. That is, two suffixes T_i' and T_j' have a common prefix of length l if and only if i and j are in the same equivalence class, E_l. Thus, the equivalence classes provides a partitioning of the suffixes, which can be exploited for improved merging. The merging step between the odd and even suffix arrays is then accomplished in linear time based on the equivalent classes. Kim *et al* also made use of minimal range queries in their algorithm. The equivalence between the minimal range queries and the lowest common ancestor problem implies that they may have to use complicated tree data structures at some stage in their algorithm.

In [33] a divide and conquer approach similar to Farach's above was used, but for direct construction of the suffix arrays. Here, rather than dividing the sequence into two symmetric parts, the sequence was divided into two unequal parts, by considering suffixes that begin at positions ($i \ mod \ 3 \neq 0$) in the sequence. These suffixes are recursively

sorted, and then the remaining suffixes are sorted based on information in the first part. The two sorted suffixes are then combined using a merging step to produce the final suffix array. Thus, a major difference is in the way they divided the sequences into two parts, and in the merging step. Also, the use of a 2/3 recursion (rather than the traditional half recursion) significantly simplified the later merging stage, since a relative order between any conflicting symbols can be found in at most 2 steps of comparison.

Ko and Aluru [48] also used recursive partitioning, but following a fundamentally different approach to construct the suffix array in linear space and time. They use a binary marking strategy whereby each suffix in T is classified as either an S-suffix or an L-suffix, depending on its relative order with its next neighbor. An S-suffix is a suffix that is lexicographically smaller than its right neighbor in T, while an L-suffix is one that is lexicographically larger than its right neighbor. That is, T'_i is an S-suffix if $T'_i \prec T'_{i+1}$, otherwise T'_i is an L-suffix. This classification is motivated by the observation that an S-suffix is always lexicographically greater than any L-suffix that starts with the same first character. The two types of suffixes are then treated differently, whereby the S-suffixes are sorted recursively by performing some special distance computations. The L-suffixes are then sorted using the sorted order of the S-suffixes. The classification scheme is very similar to the approach earlier used by Itoh and Tanaka [49]. But the algorithm in [49] runs in $O(n \log n)$ time on average.

2.6 Sequence Alignment

Most approaches to multiple sequence alignment are based on dynamic programming methods, epitomized by the Needleman-Wunsch algorithm [50] for global alignment and the Smith-Waterman algorithm [51] for local alignment. Dynamic programming produces an optimal alignment for a given set of sequences and a given cost function. Aligning two sequences of length n using dynamic programming requires time $O(n^2)$. However, extending this pairwise approach to aligning m sequences (with $m > 2$), each of length n, requires time $O(2^m n^m)$. This makes this approach very difficult to apply for many sequences, with say $m > 3$, or to long sequences, such as whole genomes with potentially billions of symbols. Thus, progressive approaches are often adopted whereby the pairwise

alignment strategy is iteratively applied to the sequences [3, 52, 53]. Here, to align the m sequences, a pair of the sequences is first aligned, and the pair is replaced with the alignment. This reduces the alignment problem to $(m - 1)$ sequences. The procedure is then performed on another pair of sequences, reducing the number of sequences needed to be aligned further. This process is carried out iteratively until only one aligned sequence is left. The resulting alignment will often be suboptimal, however, this approach significantly reduces the complexity. The problem is that the overall performance of this algorithm depends heavily on the order used in selecting the pairs at each pairwise alignment step. The exhaustive search needed for finding the optimal alignment has been traded for potential loss in robustness and quality of the alignment. CLUSTAL W and its family [3, 4] and T-COFEE [54] are among the popular algorithms that are based on progressive alignment.

Non-progressive iterative methods have also been proposed, such as, DIALIGN [1, 2] and MUSCLE [55]. Here, multiple sequence alignment starts with an initial alignment. This initial alignment is then successively refined until no further improvement, as defined by the alignment value, can be observed. For example, in DIALIGN [1,2], fragments of exact matches between the multiple sequences are assembled, and using a weighting function based on these fragments, the program tries to determine a consistent collection of fragments that will result in the minimal overall cost. MUSCLE [55], is another member of the family of iterative, non-progressive multiple sequence alignment algorithms. Probabilistic methods for sequence alignment have also been proposed, whereby existing alignments or biological knowledge about sequence substitution probabilities are used to improve the sensitivity and specificity of the alignment. Example probabilistic methods include the method based on hidden Markov models reported in [15,56], and SAGA [54].

2.7 Protein Sequence Compression

The recent exponential growth of available biological sequences has heightened significantly the need for efficient storage, retrieval and communication of such data. Compression methods have emerged to address this challenge, with varying degrees of success. Protein in particular is known to be very difficult to compress, especially when compared

with the general DNA which could have various forms of repetition. The interest in compression is, however, not just for efficient storage and data transport. For biological sequences in particular, the intrinsic relationship between complexity, redundancy and compression imply that an algorithm that can compress biological sequences could provide a means for analyzing such sequences for hidden structures. Such hidden structures could be exploited for various applications, from sequence classification to construction of phylogenic trees, to comparative genomics.

Given that we represent the information in protein sequences using 20 symbols, if the sequences were to be completely random, i.e., completely unpredictable or incompressible [57]). Then, we should need $\log_2 20$ or about 4.32 bits to code each amino acid. Traditional compression methods that have done very well on text often find it difficult to compress biological sequences, such as DNA or protein sequences. In fact, using the classical compression methods, such as word-based Huffman, arithmetic coding, or LZ-based methods, directly on such sequences often result in data expansion, rather than compression [58, 59]. The major reason is the fact that these methods often use models that were derived for traditional text, and hence fail to consider certain special characteristics of biological sequences. Biological sequences are however known to convey important purposeful information between different generations of an organism. Moreover, biological sequences are known to contain different types of repetitions and other hidden regularities. Long runs of tandem repeats and of randomly interspersed repeats are prominent features of DNA sequences. Thus, from the viewpoint of compression and sequence understanding, the repetitions inherent in biological sequences imply redundancies which can provide an avenue for a significant compaction. The identification of such dependencies is the starting point for biological sequence compression.

Different specialized algorithms have been proposed for compressing biological sequences, with varying degrees of success. Examples here include BIOCOMPRESS1 and BIOCOMPRESS2 [58], CFACT [59], GENCOMPRESS [60], and GTAC [61], and the context-tree weighting method of Matsumoto et al [62]. Maximum likelihood approaches to DNA sequence compression were proposed in [63], while methods that use offline dictionaries were reported in [40, 64]. Loewenstern and Yainilos [65] proposed a method to estimate the entropy of DNA sequences by using inexact matches based on a family

of distance measures. The different algorithms differ primarily in the type(s) of repetitions they consider, and in how the repetitions are represented and exploited to achieve compression.

Compression methods have also been proposed with specific consideration of protein sequences. In a well-cited paper [34], the PPM algorithm [66] was modified by considering mutation probabilities for the amino acids that make up protein sequences. Though the results produced were relatively better than those from the original PPM, which led to data expansion, the compression was not significant. This led the authors to conclude that protein is not compressible by any appreciable degree. In [62] context tree weighting was combined with simple $LZ77$ parsing to provide a better compression over the same data corpus used in [34]. While this is a significant result, the method did not do well on 2 of the 4 sequences in the corpus. More recently, Sampath [67] presented a block-based method that could provide a significant compression on one sequence in the corpus, namely the HI sequence, from Haemophilus influenzae, with compression ratios of up to 3.66bps. However, attempts to use the same method on the other three sequences lead to data expansion.

Chapter 3

Direct Suffix Sorting

3.1 Overview

We propose a divide-and-conquer sort-and-merge algorithm for performing direct suffix sorting on a given input string. Given a string of length n, our algorithm runs in $O(n)$ time and space in the worst-case. The algorithm recursively divides an input sequence into two parts, performs suffix sorting on the first part, then sorts the second part based on the sorted suffixes from the first. It then merges the two smaller sorted suffixes to provide the final sorted array. Thus our work is similar in spirit to the two direct suffix construction algorithms, all of which are also based on some form of recursive partitioning. Our algorithm differs from previous approaches in the use of a simple partitioning step, and how it exploits this simple partitioning scheme for conflict resolution. The method is also unique in its use of information theoretic methods in efficient construction of a global partial order for the suffixes.

We present three algorithms for solving the suffix sorting problem. In the next section 3.2, we provide a background to the problem, describe related work and present our basic algorithm for suffix sorting. Section 3.3 improves the complexity of the basic algorithm using information theoretic ideas. Section 3.4 extends the partition to arbitrary $1 : \eta$ scheme.

[1]Part of the work reported in this chapter has appeared in the papers [68–70].

3.2 Algorithm I: Basic Algorithm

3.2.1 Notations

Let $T = t_0 t_1 t_2 \ldots t_{n-1}$ denote the input sequence of length n, with symbol alphabet Σ, $\Sigma = \{\sigma_0, \sigma_1, \ldots \sigma_{|\Sigma|-1}\}$. Let $T'_i = t_i t_{i+1} t_{i+2} \ldots t_{n-1}$ denote the suffix T'_i of sequence T starting at position i. Let $T[i]$ denote the i-th symbol in T. For any two strings, say α and β, we use $\alpha \prec \beta$ to indicate that string α lexicographically precedes string β. Clearly, α and β could be individual symbols, from the same alphabet, i.e., $|\alpha| = |\beta| = 1$. We use \$ as the end of sequence symbol, where $\$ \notin \Sigma$ and $\$ < \sigma, \forall \sigma \in \Sigma$. Further, we use SA to denote the suffix array of T, and S to denote the sorted list of first characters in the suffixes. Essentially, $S[i] = T[SA[i]]$. Given SA, we use SA' to denote its inverse. We define SA' as follows: $SA'[i] = k$ if $SA[k] = i; i, k = 0, 1, \ldots, n-1$. That is, $S[k] = S[SA'[i]] = T[i]$. We use p_i to denote the probability of symbol $T[i]$, and P_i the probability of the substring $T[i, (i+1), \ldots, (i+m-1)]$, the m-length substring starting at i.

3.2.2 Overview of Algorithm

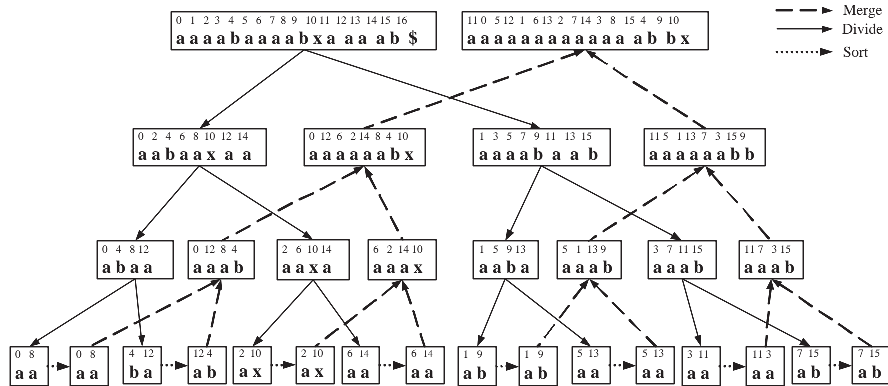

Figure 3.1: Basic working of proposed algorithm using an example sequence $T = aaaabaaaabxaaaab$. Solid arrows indicate partition procedure. Dotted arrows indicate trivial sort procedure. Dashed arrows indicate merge procedure.

We take the general divide and conquer approach: Divide the sequence into two

groups; Construct the suffix array for the first group; Construct the suffix array for the second group; Merge the suffix arrays from the two groups to form the suffix array for the parent sequence; Perform the above steps recursively to construct the complete suffix array for the entire sequence. Figure 3.1 shows a schematic flow for the working of the basic algorithm using an example sequence, $T = aaaabaaaabxaaaab$. The basic idea is to recursively partition the input sequence in a top-down manner into two equal-length subsequences according the odd and even positions in the sequence. After reaching the subsequences with length ≤ 2, the algorithm then recursively merges and sorts the subsequences using a bottom-up approach, based on the partial suffix arrays from the lower levels of the recursion. Thus, the algorithm does not start the merging procedure until it reaches the last partition on a given branch.

Each block in the figure contains two rows. The first row indicates the position of the current block of symbols in the original sequence T. The second row indicates the current symbols. The current symbols are unsorted in the downstream dividing block and sorted in the upstream merging block. To see how the algorithm works, starting from the very top left follow the solid division arrow, the horizontal trivial sort arrow (dotted arrow), and then the dashed merge arrows. The procedure ends at the top right block. We briefly describe each algorithmic step in the following. Later, we modify this basic algorithm for improved complexity results.

3.2.3 Divide T

If the length of T is greater than or equal to 2, divide T into 2 subsequences, T_0 and T_1. T_0 contains all the symbols at the even positions in T. T_1 contains all the symbols at the odd positions of T. i.e., $T_0 = \{t_j, j \in [0, |T|)| \ j \mod 2 \ = 0\}$, $T_1 = \{t_j, j \in [0, |T|)| \ j \mod 2 \ = 1\}$

Example:

$$\begin{aligned} \text{j} &= 0 \;\; 1 \;\; 2 \;\; 3 \;\; 4 \;\; 5 \;\; 6 \;\; 7 \;\; 8 \;\; 9 \;\; 10 \\ T[0, n-1] &= \text{C A T T A T T A G G A} \end{aligned}$$

$$\begin{array}{l} \quad\;\; 0\; 2\; 4\; 6\; 8\; 10 \qquad\quad 1\; 3\; 5\; 7\; 9 \\ T_0 = \mathrm{C\; T\; A\; T\; G\; A} \qquad T_1 = \mathrm{A\; T\; T\; A\; G} \end{array}$$

3.2.4 Merge SA for T_0 and T_1

Let SA_0 and SA_1 be the suffix array of T_0 and T_1 respectively. Let SA be the suffix array of T. If T_0 and T_1 have been sorted to obtain their respective suffix arrays SA_0 and SA_1, then we can merge SA_0 and SA_1 in linear time on average, to form the suffix array SA. Without loss of generality, we assume $SA_0 = a_0a_1a_2 \ldots a_u$, $SA_1 = b_0b_1b_2 \ldots b_v$ and $SA = c_0c_1c_2 \ldots c_{u+v+1}$ are the sorted indices of T_0, T_1 and T respectively. Given a_k, we use $\hat{a}_k$ to denote its corresponding position in T. That is, $T_0[SA_0[k]] = T_0[a_k] = T[\hat{a}_k]$. Similarly, for $\hat{b}_k$. $\hat{a}_k$ and $\hat{b}_k$ are easily obtained from a_k, based on the level of recursion, and whether we are on a left branch or a right branch. For $k = 0$ $(0 \leq k \leq u), l = 0$ $(0 \leq l \leq v)$ and $g = 0$ $(0 \leq g \leq (u + v + 1))$, we compare the partially ordered subsequences using a_k and b_l, viz.

$$\text{If } T[\hat{a}_k] \prec T[\hat{b}_l], \quad \begin{cases} SA[c_g] \leftarrow \hat{a}_k, S[c_g] \leftarrow T[\hat{a}_k] \\ k{+}{+}, g{+}{+} \end{cases}$$

$$\text{If } T[\hat{b}_l] \prec T[\hat{a}_k], \quad \begin{cases} SA[c_g] \leftarrow \hat{b}_l, S[c_g] \leftarrow T[\hat{b}_l] \\ l{+}{+}, g{+}{+} \end{cases}$$

$$\text{If } T[\hat{a}_k] = T[\hat{b}_h], \quad \begin{cases} c_x \leftarrow \text{ResolveConflict}(\hat{a}_k, \hat{b}_l) \\ SA[c_g] \leftarrow c_x, S[c_g] \leftarrow T[c_x] \\ g{+}{+} \end{cases}$$

Whenever we compare two symbols, $T[\hat{a}_k]$ from the first subsequence T_0 and $T[\hat{b}_l]$ from the second subsequence T_1, we might get into the ambiguous situation whereby the two symbols are the same (i.e., $T[\hat{a}_k] = T[\hat{b}_l]$). Thus, we cannot easily decide which suffix precedes the other, (i.e., whether $T'_{\hat{a}_k} \prec T'_{\hat{b}_l}$, or $T'_{\hat{b}_l} \prec T'_{\hat{a}_k}$), based on the individual symbols. We call this situation a **conflict**. The key to the approach is how efficiently we can resolve potential conflicts as the algorithm progresses. We exploit the nature of the division procedure, and the fact that we are dealing with substrings (suffixes) of the same string, for efficient conflict resolution. Thus, the result of the merging step will be a partially sorted subsequence, based on the sorted order of the smaller child subsequences.

An important difference here is that unlike in other related approaches [33, 47, 48], our sort-order at each recursion level is global with respect to T, rather than being local to the subsequence at the current recursion step. This is important, as it significantly simplifies the subsequent merging step.

Example:

$$\begin{array}{rccccccccccc} & 10 & 7 & 4 & 1 & 0 & 9 & 8 & 6 & 3 & 5 & 2 \\ SA = & \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{array}$$

$SA_0 = 10\ 4\ 0\ 8\ 6\ 2$ $\quad SA_1 = 7\ 1\ 9\ 3\ 5$

$S_0 = \mathrm{A\ A\ C\ G\ T\ T}$ $\quad S_1 = \mathrm{A\ A\ G\ T\ T}$

3.2.5 Recursive Call

Using the above procedure, we recursively construct the suffix array of T_0 from its two children T_{00} and T_{01}. Similarly, we obtain the suffix array for T_1 from its children T_{10} and T_{11}. We follow this recursive procedure until the base case that can be solved trivially.

3.2.6 Conflict Resolution

An important issue in the proposed algorithm is how conflicts are resolved. We use the notions of **conflict sequence** or **conflict pairs**. Two suffixes T'_i and T'_j form a conflict sequence in T if $T'_i[0 \dots k] = T[i \dots i+k] == T[j \dots j+k] = T'_j[0 \dots k]$, for some $k \geq 1$. That is, the two suffixes can not be assigned a total order after considering their first k symbols. We say that the conflict between T'_i and T'_j is resolved whenever $T[i \dots i+l_{k-1}] = T[j \dots j+l_{k-1}]$, but $T[i+l_k] \neq T[j+l_k]$. Here, l_k is the conflict length. We call the triple (T'_i, T'_j, l_k) a conflict pair. We use the notation $CP(i, j, l_k)$ to denote a conflict pair (conflict sequence) T'_i and T'_j with a conflict length of l_k. We also use $CP(i, j)$ to denote a conflict pair where the conflict length is yet to be determined, or is not important given the context of the discussion.

The challenge, therefore, is how to resolve conflicts efficiently given the recursive partitioning framework proposed. Obviously, l_k, the conflict length cannot be greater than n, the sequence length. Thus, the minimum distance between the start of each

conflicting suffix and the end of the sequence ($\min\{(n-i), (n-j)\}$), or the distance from an already sorted symbol can determine how long it will take to resolve a conflict. Here, we consider how the previously resolved conflicts can be exploited for a quick resolution of other conflicts. We maintain a traceback record on such previously resolved conflicts in a **conflict tree**, or a **conflict table**.

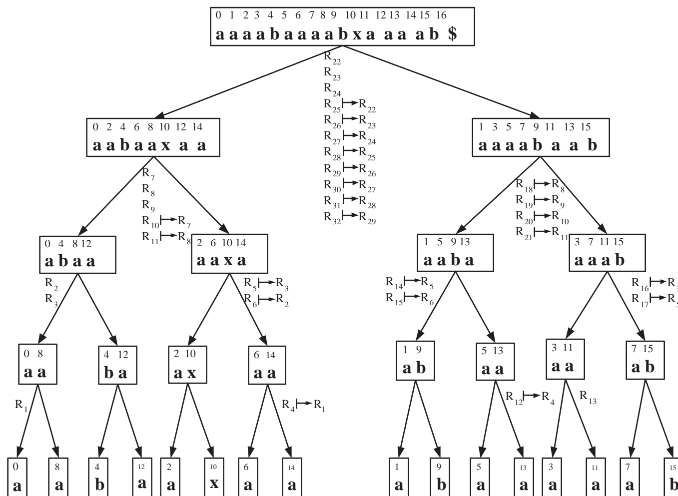

Figure 3.2: Conflict tree for an example sequence. The original sequence is indicated at the root node. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p.

Figure 3.2 shows the conflict tree for the sequence, $T = aaaabaaabxaaaab$ used previously. Without loss in generality and for easier exposition of the basic methodology, we assume that $n = 2^x$, for some positive integer x. Our conflict resolution strategy is based on the following properties of conflict sequences and conflict trees.

The sequence $T = a^n$ represents the worst case for suffix sorting algorithms, as it results in both the maximum **lcp** and the maximum average **lcp** for any n-length sequence.

Given the even-odd recursive partitioning scheme, at any given recursion level, conflicts can only occur between specific sets of positions in the original sequence T. For instance, at the lowest level, conflicts can only occur between T_0, and $T_{n/2}$, or more generally, between T_i, and $T_{i+n/2}$. See Figure 3.1. Further, given the conflict sequence, T'_i and T'_j, we can see that the corresponding conflict pair $CP(i, j, l_k)$ is unique in T. That

is, only one conflict pair can have the pair of start positions (i, j). Thus the conflict pairs $CP(i, j, l_k)$ and $CP(j, i, l_k)$ are equivalent. Hence, we represent both $CP(i, j, l_k)$ and $CP(j, i, l_k)$ as $CP(i, j, l_k)$, where, $i < j$.

Given the conflict tree, the **decision** on whether a given conflict pair $CP(i, j, l_k)$ can be resolved based on previous trace back information (i.e., previously resolved conflicts) can be made by considering only those at the same level in the conflict tree, including those in the same node with $CP(i, j, l_k)$. If we determine that the current conflict **can be resolved** based on previous conflicts, we may need little or no extra work to perform the actual resolution. However, if it is determined that the conflict cannot be resolved based on previously recorded traceback information, we will need to perform symbol-wise comparisons to break the tie.

From the conflict tree, we see that given a conflict pair $CP(i, j)$, we can make the decision by checking only a **fixed** number of neighboring conflict pairs, where the neighborhood is defined based on i, j, and h, the level in the conflict tree. Consider $CP(i, j)$ at level $h, 0 \leq h \leq \lceil \log n \rceil$ in the tree. The **immediate neighbors** will be the conflict pairs: $CP(i', j')$, with $i' = i + k2^h, j' = j + k2^h$, or $i' = i - k2^h, j' = j - k2^h$, where $k = 1, 2, \ldots$, and $i', j' \leq n$. Similarly, we define the **left neighbors** of $CP(i, j)$: $CP(i', j')$, with $i' = i - k2^{h-1}, j' = j - k2^{h-1}$, where $k = 1, 2, \ldots$, and $i', j' \leq n$.

A neighbor of $CP(i, j)$ is then defined as either an immediate neighbor or a left neighbor. Essentially, neighboring conflicts are found on the same level in the conflict tree, from the leftmost node to the current node. For example, for $R_4 = CP(6, 14)$ at $h = 3$ in the example sequence (see Figure 3.2 and Table 3.1), the neighbor will be: $\{R_1\}$, or equivalently $\{CP(0, 8)\}$. Similarly, for $R_6 = CP(2, 14)$ at $h = 2$, the neighbor will be: $\{R_2\}$. We notice that, by our definition, not all conflicts in the same node are neighbors. In fact, to make the decision, we only need to check the relatively nearest neighbors in the conflict tree. For $CP(i, j)$, these will be the conflict pairs in the set $CP(i', j')$, with $i' = i - 2^h, j' = j - 2^h$; $i' = i + 2^h, j' = j + 2^h$; and $i' = i - 2^{h-1}, j' = j - 2^{h-1}$. Therefore, only three probes need to be made.

Let $CP(i, j)$ be the current conflict. Let $CP(i', j', l'_k)$ be a previously resolved neighboring conflict. We determine whether $CP(i, j)$ can be resolved based on $CP(i', j', l'_k)$ using a simple check:

Let $\Delta = i' - i$. $CP(i,j)$ can be resolved with $CP(i',j',l'_k)$ iff: $(i'-i) = (j'-j) = \Delta$ **and** $|\Delta| \leq l'_k$. If this condition holds, we say that $CP(i,j)$ is resolved by $CP(i',j',l'_k)$, after Δ steps. Essentially, this means that after Δ comparison steps, $CP(i,j)$ becomes equivalent to $CP(i',j',l'_k)$. We denote this equivalence using the notation: $CP(i,j) \mapsto CP(i',j',l'_k)$. The actual resolution is then performed as follows:

1. If $\Delta < 0$, no extra work is needed; compute $l_k = l'_k + \Delta$.

2. If $\Delta = 0$, no extra work is needed; compute $l_k = l'_k$.

3. If $\Delta > 0$, we need $\Delta - 1$, extra comparison steps. If conflict is not resolved after the $\Delta - 1$ comparisons, then conflict is resolved using $CP(i',j',l'_k)$; compute $l_k = l'_k + \Delta$.

The complexity of conflict resolution (when it can be resolved using previous conflicts) thus depends on the parameter Δ. Table 3.1 shows the conflict pairs for the example sequence used in Figure 3.2. That table includes the corresponding Δ value where they are resolved based on previous conflicts. Figure 3.2 shows the original conflict tree.

R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ
R_1	$(0,8,8)$	$-$	$-$	R_9	$(6,8,1)$	$-$	$-$	R_{17}	$(3,7,1)$	R_5	-1	R_{25}	$(1,12,4)$	R_{22}	-1
R_2	$(0,12,3)$	$-$	$-$	R_{10}	$(2,8,1)$	R_7	-2	R_{18}	$(5,11,5)$	R_8	$+1$	R_{26}	$(1,6,4)$	R_{23}	-1
R_3	$(8,12,1)$	$-$	$-$	R_{11}	$(8,14,2)$	R_8	-2	R_{19}	$(5,7,2)$	R_9	$+1$	R_{27}	$(6,13,2)$	R_{24}	-1
R_4	$(6,14,1)$	R_1	-6	R_{12}	$(5,13,2)$	R_4	$+1$	R_{20}	$(1,7,2)$	R_{10}	$+1$	R_{28}	$(2,13,3)$	R_{25}	-1
R_5	$(2,6,2)$	R_3	$+6$	R_{13}	$(3,11,1)$	$-$	$-$	R_{21}	$(7,13,3)$	R_{11}	$+1$	R_{29}	$(2,7,3)$	R_{26}	-1
R_6	$(2,14,1)$	R_2	-2	R_{14}	$(1,5,3)$	R_5	$+1$	R_{22}	$(0,11,5)$	$-$	$-$	R_{30}	$(7,14,1)$	R_{27}	-1
R_7	$(0,6,3)$	$-$	$-$	R_{15}	$(1,13,2)$	R_6	$+1$	R_{23}	$(0,5,5)$	$-$	$-$	R_{31}	$(3,14,2)$	R_{28}	-1
R_8	$(6,12,4)$	$-$	$-$	R_{16}	$(7,11,2)$	R_3	$+1$	R_{24}	$(5,12,3)$	$-$	$-$	R_{32}	$(3,8,2)$	R_{29}	-1

Table 3.1: Conflict pairs for $T = aaaabaaaabxaaaab$. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p.

The final consideration is how to store the conflict tree to ensure constant-time access to the resolved conflict pairs. Since for a given $CP(i,j)$, the (i,j) pair is unique, we can use these for direct access to the conflict tree. We can store the conflict tree as a simple linear array, where the positions in the array are determined based on the (i,j) values, and hence the height in the tree. To avoid the sorting that may be needed for fast access

R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ	R	$CP(i,j,l_k)$	$\mapsto$	Δ
R_1	$(0,8,8)$	–	–	R_{14}	$(6,8,8)$	R_{13}	$+2$	R_{27}	$(3,7,9)$	R_{10}	-1	R_{40}	$(9,10,6)$	R_{39}	$+1$
R_2	$(4,12,4)$	R_1	-4	R_{15}	$(4,6,10)$	R_{14}	$+2$	R_{28}	$(13,15,1)$	R_{11}	-1	R_{41}	$(8,9,7)$	R_{40}	$+1$
R_3	$(8,12,4)$	–	–	R_{16}	$(2,4,12)$	R_{15}	$+2$	R_{29}	$(11,13,3)$	R_{12}	-1	R_{42}	$(7,8,8)$	R_{41}	$+1$
R_4	$(4,8,8)$	R_3	$+4$	R_{17}	$(0,2,14)$	R_{16}	$+2$	R_{30}	$(9,11,5)$	R_{13}	-1	R_{43}	$(6,7,9)$	R_{42}	$+1$
R_5	$(0,4,12)$	R_4	$+4$	R_{18}	$(1,9,7)$	R_1	-1	R_{31}	$(7,9,7)$	R_{14}	-1	R_{44}	$(5,6,10)$	R_{43}	$+1$
R_6	$(2,10,6)$	R_1	-2	R_{19}	$(5,13,3)$	R_2	-1	R_{32}	$(5,7,9)$	R_{15}	-1	R_{45}	$(4,5,11)$	R_{44}	$+1$
R_7	$(6,14,2)$	R_2	-2	R_{20}	$(9,13,3)$	R_3	-1	R_{33}	$(3,5,11)$	R_{16}	-1	R_{46}	$(3,4,12)$	R_{45}	$+1$
R_8	$(10,14,2)$	R_3	-2	R_{21}	$(5,9,7)$	R_4	-1	R_{34}	$(1,3,13)$	R_{17}	-1	R_{47}	$(2,3,13)$	R_{46}	$+1$
R_9	$(6,10,6)$	R_4	-2	R_{22}	$(1,5,11)$	R_5	-1	R_{35}	$(14,15,1)$	–	–	R_{48}	$(1,2,14)$	R_{47}	$+1$
R_{10}	$(2,6,10)$	R_5	-2	R_{23}	$(3,11,5)$	R_6	-1	R_{36}	$(13,14,2)$	R_{35}	$+1$	R_{49}	$(0,1,15)$	R_{48}	$+1$
R_{11}	$(12,14,2)$	–	–	R_{24}	$(7,15,1)$	R_7	-1	R_{37}	$(12,13,3)$	R_{36}	$+1$				
R_{12}	$(10,12,4)$	R_{11}	$+2$	R_{25}	$(11,15,1)$	R_8	-1	R_{38}	$(11,12,4)$	R_{37}	$+1$				
R_{13}	$(8,10,6)$	R_{12}	$+2$	R_{26}	$(7,11,5)$	R_9	-1	R_{39}	$(10,11,5)$	R_{38}	$+1$				

Table 3.2: Conflict pairs for $T = aaaaaaaaaaaaaaaa = a^{16}$. $R_q \mapsto R_p$ indicates that conflict pair R_q is resolved by a previously resolved conflict pair R_p, after a fixed number of steps. That is, after the fixed number of steps Δ, conflict pair R_q becomes equivalent to conflict pair R_p.

using the (i,j) indices, we use a simple hash function, where each hash value can be computed in constant time. The size of this array will be $O(n \log n)$ in the worst case, since there are at most $n \log n$ conflicts (see analysis below). The result will be an $O(1)$ time access to any given conflict pair, given its (i,j) index in T.

3.2.7 Complexity Analysis

Clearly, the complexity of the algorithm depends on the complexity of the conflict resolution strategy. The time depends on the number of conflicts that can be resolved using the traceback information, and the number that require symbol-wise comparison. We notice that we require symbol-wise comparison mainly for the conflicts at the leftmost nodes on the conflict tree, and the first few conflicts in each node. On average, the number of such conflicts will be $O(n)$, and all the conflicts can be resolved using no more than $O(n)$ number of comparisons. Also, on average, the number of conflicts that can be resolved using previous conflicts will be in $O(n)$, and each can be resolved in constant time. Thus, on average, the proposed algorithm will run in $O(n)$ time. In the worst case, each node in the tree will have the maximum number of conflicts. This maximum is simply given

by $(\frac{n}{2^h} - 1)$, where h is the height of the node. Thus, the worst case total number of conflicts will be: $\sum_{h=0}^{\lceil \log n \rceil} 2^h(\frac{n}{2^h} - 1) \leq n \log n$. These can be resolved in $O(n \log n)$ time.

We can reduce the space required for conflict resolution as follows. Consider level h in the conflict tree, $0 \leq h \leq \lceil \log n \rceil$. We had already observed that, if a level-h conflict pair can be resolved based on previous conflicts, then only information from previous conflict pairs at the same level are needed. Thus, after we resolve all conflicts at level-h, we can save memory by discarding (or re-using) the memory used to store level-h conflicts. Thus, using the current depth-first traversal (as used to enumerate the conflict pairs in the tree, see Figure 3.2), we will need space for only conflicts in the left half of the conflict tree, and at the lowest level on the right half. Alternatively, we can change the traversal order, and use a bottom-up breadth-first traversal. Thus, starting with the lowest level, $h = \lceil \log n \rceil$, we resolve all conflicts at a given level before moving to the next level. Then, we modify the size of the conflict table after each level and clear all previous entries. This implies a maximum of $(n - 1)$ entries in the conflict table at any given time, or $O(n)$ space.

We make a final observation about the nature of the algorithm above. It may be seen that, in deed, we do not need to touch T_1, the odd tree at all, until the very last stage of final merging. Since we can sort T_0 to obtain SA_0 without reference to T_1, we can therefore use SA_0 to sort T_1 using radix sort, since positions in T_0 and T_1 are adjacent in T. This will eliminate consideration of the $(n - 1)$ worst case conflicts at level $h = 0$, and all the conflicts in T_1. This will however not change the complexity results, since the main culprit is the $O(n \log n)$ time required to resolve all conflicts in the worst case. We make use of this observation in the next section, to develop an $O(n)$ time and space algorithm for suffix sorting. We conclude this section with the following theorem:

Theorem 1: *Given an input sequence $T = t_0, t_2 \ldots, t_{n-1}$, with $|T| = n$,* **Algorithm I** *solves the suffix sorting problem in $O(n)$ time and space on average, and a worst case complexity of $O(n \log n)$ time using $O(n)$ space.*

3.3 Algorithm II: Improved Algorithm

From the analysis above, the major problem with Algorithm I is the time taken for conflict resolution. Since the worst case number of conflicts is in $O(n \log n)$, an algorithm that performs a sequential resolution of each conflict can do no better than $O(n \log n)$ time in the worst case. We improve the algorithm by modifying the recursion step, and the conflict resolution strategy. Specifically, we still use binary partitioning, but we use a non-symmetric treatment of the two branches at each recursion step. That is, only one branch will be sorted directly, and the other branch will be sorted based on the sorted results from the other branch. This is motivated by the observation at the end of the last section. We also use a preprocessing stage inspired by methods from information theory to facilitate fast conflict resolution.

3.3.1 Overview

In Algorithm I, we divide T into two subsequences T_0 and T_1, and then merge their respective suffix arrays SA_0 and SA_1 to form SA, the suffix array of T. SA_0 and SA_1 in turn are obtained by recursive division of T_0 and T_1 and subsequent merging of the suffix arrays of their respective children. The improvement in Algorithm II is that when we divide T into T_0 and T_1, we make the recursive call on only one branch of T, namely T_1 in our case. After we obtain the suffix array SA_1 for T_1, we radix sort T_0 based on the values in SA_1 to construct SA_0. This radix sorting step only takes linear time. The merging step and division step remain similar to Algorithm I. However, we now need an ordering stage.

3.3.2 Sort T_1 to Form SA_1

After dividing T into T_0 and T_1, we need to sort each child sequence to form its own smaller suffix array. Consider T_1. We form SA_1 by performing the required sorting recursively, using a non-symmetric treatment of each branch. T_1 is thus sorted by a recursive subdivision, and local sorting at each step. The suffix arrays of the two children are then merged to form the suffix array of their parent. Figure 3.3 shows this procedure

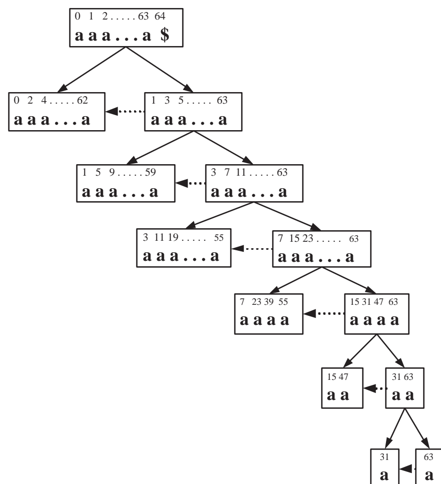

Figure 3.3: Improved Algorithm II: Asymmetric recursive partitioning for improved algorithm, using $T = a^{64}$. Dotted arrows (from right to left) denote propagation of the sorted array at the given level of recursion.

for $T = a^{64}$. We have used a longer sequence to show the structure in this asymmetric partition-and-merge process more clearly. Merging can be performed as before (we also discuss an improved merging procedure below). The key to the algorithm is how to obtain the sorted array of the left child from that of the right child at each recursion step.

3.3.3 Sort T_0 to form SA_0 using SA_1

Without loss in generality, we assume T_0 is unsorted and T_1 has been sorted to form SA_1. Given the odd/even partitioning scheme, we have that for any $t_k \in T_0$, $t_{k+1} \in T_1$. There must exist an ordering of $t_{k+1} \in SA_1$, since the indices in SA_1 are unique. For each k, we construct the pair of $P = \{(t_k, SA_1'[k+1]) | t_k \in T_0\}$. Each pair in P is unique. Then, we radix sort P to generate the suffix array SA_0 of T_0. This step can thus be accomplished in linear time. This only works for the highest level, i.e., obtaining SA_0 from SA_1). However, we can use a similar procedure, but with some added work, at the other levels of recursion.

Consider a lower level of recursion, say at level h in the tree. See Figure 3.3. We can

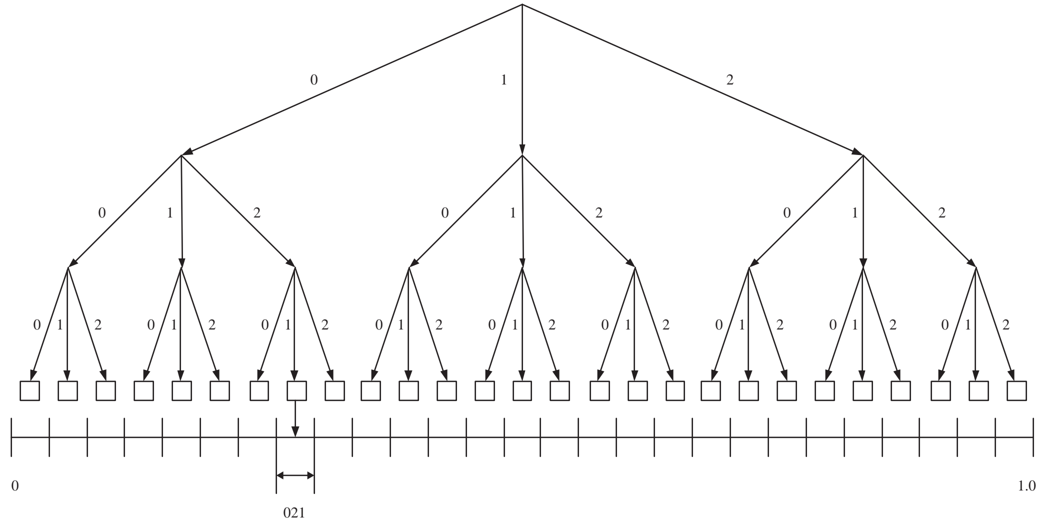

Figure 3.4: Improved Algorithm II: Code assignment by successive partitioning of a number line. Code 021 is selected by following the path of 0, 2 and 1.

append the values in the SA from the right tree so that we can perform successive bucket sorts. Thus, we use the SA from the right tree as the tie breaker, after a certain number of steps. In general, this will not change the number of comparisons at the lowest level of the tree, since we still may need to perform $\frac{n}{4}+1$ comparisons before reaching the tie breaker. However, for the general case, this will ensure that at most 2^{h-1} bucket sorts are performed, involving substrings with at most $2^{h_{max}-h}$ symbols in each bucket sort, where $h_{max} = \lceil \log n \rceil$ is the lowest level of recursion.

The single conflict at this level $CP(\frac{n}{2}-1, n-1)$ can be resolved using at most 2 comparisons, to form the short suffix array at this node with length 2. See Figure 3.4. Then, we use this to derive the suffix array at the left branch: $CP(\frac{n}{4}-1, \frac{3n}{4}-1)$, or $CP(15, 47)$ in the figure. This will require at most $\frac{n}{4}+1$ comparisons.

For completeness, we can append the values from the SA of the right tree so that we can perform successive bucket sorts ($\frac{n}{4}+1$ bucket sorts) using the two symbols.

That is, we form the sequence: $T[\frac{n}{4}-1 \ldots \frac{3n}{4}-1] \circ SA[1]$, and $T[\frac{n}{4}-1 \ldots \frac{3n}{4}-1] \circ SA[0]$, where the symbol $\circ$ denotes concatenation. For the example with $T = a^{64}$, this will form sequences: $T[15 \ldots 31] \circ 1$ and $T[47 \ldots 61] \circ 0$, since the suffix array of the right hand tree will be $SA = [1, 0]$.

For instance, using the example in Figure 3.3, at $h = 4$, we will have $T_1 = a^{15}a^{31}a^{47}a^{63}$, and $T_0 = a^7a^{23}a^{39}a^{55}$. For convenience, we use superscripts to denote the respective positions in the original sequence T. Assume T_1 has been sorted to obtain $SA_1 = [3, 2, 1, 0]$. Using SA_1, we now form the sequences: $T[7 \dots 15] \circ 3$; $T[23 \dots 31] \circ 2$; $T[39 \dots 47] \circ 1$; and $T[55 \dots 63] \circ 0$, where we have used the symbol "$\circ$" to denote concatenation. The last symbol in each sequence is obtained from SA_1 the suffix array of the right tree. These sequences are then radix-sorted to obtain SA_0. Since radix sorting is linear in the number of input symbols, in the worst case, this will result in a total of $O(n)$ time at each level. Thus, the overall time for this procedure will still be in $O(n \log n)$ worst case.

3.3.4 Improved Sorting

The key to improved sorting is to reduce the number of bucket sorts in the above procedure. We do this by pre-computing some information before hand, so that the sorting can be performed based on a small block of symbols, rather than one symbol at a time. Let m be the block size. With the pre-computed information, we can perform a comparison involving an m-block symbol in $O(1)$ time. This will reduce the number of bucket sorts required at each level h from 2^{h-1} to $\frac{2^{h-1}}{m}$, each involving $2^{h_{max}-h}$ symbols. By an appropriate choice of m, we can reduce the complexity of the overall sorting procedure. For instance, with $m = \log \log n$, this will lead to an overall worst case complexity in $O(\frac{n \log n}{\log \log n})$ for determining the suffix arrays of the left tree from the right tree. With $m = \log n$, this gives $O(n)$. We use $m = \log n$ in subsequent discussions.

The question that remains then is *how* to perform the required computations, such that *all* the needed block values can be obtained in linear time. Essentially, we need a pair-wise global partial ordering of the suffixes involved in each recursive step. First, we observe that we only need to consider the ordering between pairs of suffixes at the same level of recursion. The relative order between suffixes at different levels is not needed. For instance, for the sequence $T = a^{64}$, the sets of suffixes for which we need the pair-wise orderings will be: $\{\{1, 5, 9, \dots, 59\}; \{3, 11, 19, \dots, 55\}; \{7, 23, 39, 55\}; \{15, 47\}\}$. Each subset corresponds to a level of recursion, from $h = 1$ to $h = h_{max} - 1$.

We need a procedure to return the order between each pair of suffix positions in

constant time. Given that we already have an ordering from the right tree in SA_1, we only need to consider the prefixes of the suffixes in the left tree up to the corresponding positions in T_1, such that we can use entries in SA_1 to break the tie, after a possible $\frac{2^{h-1}}{m}$ bucket sorts. Let Q_i be the m-length prefix of T'_i: $Q_i = T[i \dots i+m-1]$. We can use a simple hash function to compute a representative of Q_i, for instance using the polynomial hash function:

$h(Q_i) = \sum_{j=0}^{m-1} |\Sigma|^{m-1} f(Q_i[j]) \mod n'$

where $f(x) = k$, if x is the k-th symbol in Σ, $k = 0, 1, \dots, |\Sigma|-1$, and n' is the nearest prime number $\geq n$. The problem is that the ordering information is lost in the modulus operation. Although order-preserving hash functions exist [20, 71, 72], these run in $O(n)$ time on average, without much guarantees on their worst case. Also, with the m-length blocks, this may require $O(mn) = O(n \log n)$ time on average.

We use an information theoretic approach to determine the ordering for the pairs. We consider the required representation for each m-length block as a codeword that can be used to represent the block. The codewords are constrained to be order preserving: That is, $\mathcal{C}(Q_i) < \mathcal{C}(Q_j)$ iff $Q_i \prec Q_j$ and $\mathcal{C}(Q_i) = \mathcal{C}(Q_j)$ iff $Q_i = Q_j$, where $\mathcal{C}(x)$ is the codeword for sequence x. Unlike in traditional coding in data compression where we are given one long sequence to produce its compact representation, here, we have a set of short sequences, and we need to produce their respective compact representations, and these representations must be order preserving.

Let P_i be the probability of Q_i, the m-length block starting at position $T[i]$. Let p_i be the probability of symbol $t_i = T[i]$. We compute the quantity: $P'_i = \prod_{k=i}^{i+m-1} p_k$. Recall that $t_i = T[i] \in \Sigma$, $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_{\sigma_{|\Sigma|-1}}\}$ and $\sum_{j=0}^{|\Sigma|-1} Pr\{\sigma_j\} = 1$. For a given sequence T, we should have: $\sum_{i=0}^{n-1} P_i = 1$. However, since T may not contain all the possible m-length blocks in Σ^m, we need to normalize the product of probabilities to form a probability space: $P_i = \frac{P'_i}{\sum_{i=0}^{n-1} P'_i}$. If necessary, we can pad T with $(m-1)$ \$ symbols, to form a valid m-block at the end of the sequence. To determine the code for Q_i, we then use the cumulative distribution function (cdf) for the P_i's, and determine the corresponding position for each P_i in this cdf. Essentially, this is equivalent to dividing a number line in the range [0,1], such that each Q_i is assigned a range proportional to its probability, P_i. The total number of divisions will be equal to the number of unique

m-length blocks in T. The problem is then to determine the specific interval on this number line that corresponds to Q_i, and to choose a tag to represent Q_i.

We use the following assignment procedure to compute the tag, q_i. First we determine the interval for the tag, based on which we compute the tag. Define the cumulative distribution function for the symbols in $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma|-1}\}$: $F_x(\sigma_j) = \sum_{v=0}^{j} Pr\{\sigma_v\}$.

The symbol probabilities, $Pr\{\sigma_v\}$'s are simply obtained based on the p_i's. For each symbol σ_k in Σ, we have an open interval in the cdf: $[F_x(\sigma_{k-1})\ F_x(\sigma_k))$. Now, given the sequence $Q_i = s_1 s_2 \ldots s_k \ldots s_m,\ s_i \in \Sigma$, the procedure steps through the sequence. At each step $k, k = 1, 2, \ldots, m$ along the sequence, we can compute $U(k)$ and $L(k)$ the respective current upper and lower ranges for the tag using the following relations:

$$L(0) = 0$$
$$U(0) = 1$$

for $k = 1$ to m :

$$U(k) = L(k-1) + [U(k-1) - L(k-1)]F_x(s_k)$$
$$L(k) = L(k-1) + [U(k-1) - L(k-1)]F_x(s_k - 1)$$

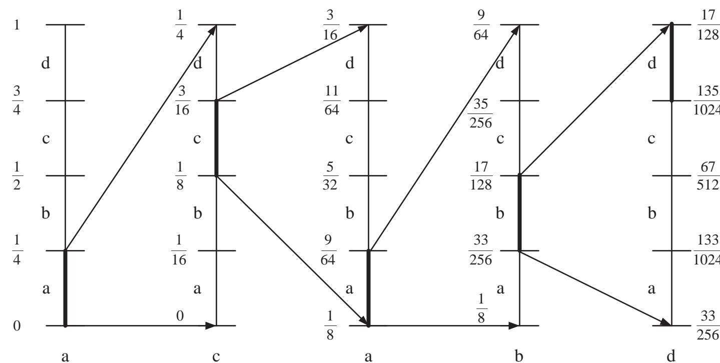

Figure 3.5: Code assignment procedure, using an example sequence: $Q_i = acabd$. The vertical line represents the current state of the number line. The current interval at each step in the procedure is shown with a darker shade. The symbol considered at each step is listed under its corresponding number line.

The procedure stops at $k = m$, and the value of $U(k)$ and $L(k)$ at this final step will be the range of the tag, q_i. We can choose the tag q_i as any number in the range: $L(m) \leq q_i < U(m)$. Thus we chose q_i as the mid-point of the range at the final step: $q_i = \frac{U(m)+L(m)}{2}$. Figure 3.5 and Figure 3.6 show an example run of this procedure for

a short sequence: $Q_i = acabd$ with a simple alphabet, $\Sigma = \{a, b, c, d\}$, and where each symbol has an equal probability $p_i = \frac{1}{4}$. This gives $q_i = \frac{271}{2048}$.

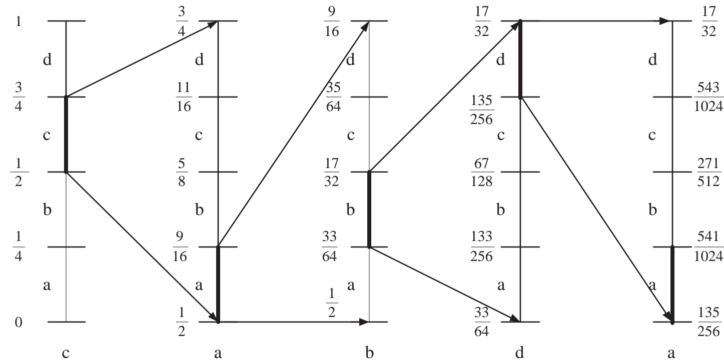

Figure 3.6: Evolution of code assignment procedure, after removing the first symbol in the previous sequence *acabd*, and bringing in a new symbol *a* to form a new sequence: *cabda*.

Lemma 1: *The assignment procedure results in a tag q_i that is unique to Q_i, and also order preserving.*

Proof: The procedure described can be seen as an extension of the Shanon-Fano-Elias coding procedure in information theory [73]. Each tag q_i is analogous to an arithmetic coding sequence of the given m-length block, Q_i. The open interval defined by $[(L(m), U(m))$ for each m-length sequence is unique to the sequence. To see this uniqueness, we notice that the final number line at step m represents the cdf for all m-length blocks that appeared in the original sequence T. Denote this cdf for the m-blocks as F_x^m. Given Q_i, the i-th m-block in T, the size of its interval is given by $(U(m) - (L(m)) = P_i$. Since all probabilities are positive, we see that $F_x^m(Q_i) \neq F_x^m(Q_j)$ whenever $Q_i \neq Q_j$. Therefore, we can use $F_x^m(Q_i)$ to determine Q_i uniquely. Thus, $F_x^m(Q_i)$ serves as a unique code for Q_i. Choosing any number q_i within the upper and lower bounds for each Q_i define a unique tag for Q_i. Thus the chosen tag defined by the midpoint of this interval is unique to Q_i.

Now consider the ordering of the tags for different m-length blocks. Each step in the assignment procedure uses a fixed order of the symbols on a number line, based on their

order in Σ. Thus, the position of the upper and lower bounds at each step depends on the previous symbols considered, and the position of the current symbol in the ordered list of symbols in Σ. Therefore the q_i's are ordered with respect to the lexicographic ordering of the Q_i's:

$q_i < q_j \Leftrightarrow Q_i \prec Q_j$, and $q_i = q_j \Leftrightarrow Q_i = Q_j$.

□

Suppose we have already determined P_i and q_i for m-block Q_i as described above. For efficient processing, we can compute P_{i+1} and the tag q_{i+1} in the $[0, 1]$ number line, using the previous values for P_i and q_i. This is based on the fact that Q_i and Q_{i+1} are consecutive positions in T[1]. In particular, given $Q_i = T[i \ldots i+m-1]$, and $Q_{i+1} = T[i+1 \ldots i+m]$. We compute P'_{i+1} as:

$$P'_{i+1} = \frac{P'_i \cdot p_{i+m}}{p_i}.$$

Thus, all the required P_i's can be computed in $O(n+m) = O(n+\log n) = O(n)$ time. Similarly, given the tag q_i for Q_i, and its upper and lower bounds $U(m)$ and $L(m)$, we can compute the new tag $q_{i+1}$for the incoming m-block, Q_{i+1} based on the structure of the assignment procedure used to compute q_i. Figure 3.6 shows a continuation of the previous example, with the old sequence: $Q_i = acabd$, and a new sequence $Q_{i+1} = cabda$. That is, the new symbol a has been shifted in, while the first symbol in the old block has been shifted out. We observe that the general structure in, Figure 3.5, is not changed by the incoming symbol, except at the end, and at the very first step. We can compute the new tag q_{i+1} by first computing it's upper and lower bounds. Denote the upper and lower bounds for q_i as: $U^i(m), L^i(m)$ respectively. Similarly, we use $U^{i+1}(m), L^{i+1}(m)$ for the respective upper and lower bounds for q_{i+1}.

Let $s = T[i] = Q_i[0]$ be the first symbol in Q_i. Its probability is given by p_i. Also, let $s_{new} = T[i+m]$ be the new symbol that is shifted in. Its probability is given by p_{i+m}, and we also know it's position in the cdf. We first compute the intermediate bounds at step $k = m-1$ when using Q_{i+1}, namely:

[1]In practice, we only need the odd positions in T, which will mean less time and space. But here we describe the procedure for the entire T since the complexity remains the same.

$$U^{i+1}(m-1) = [U(m) - F_x(s)](\tfrac{1}{p_i})$$
$$L^{i+1}(m-1) = [L(m) - F_x(s-1)](\tfrac{1}{p_i})$$

Multiplying by $(\frac{1}{p_i})$ changes the probability space from the previous range of $[F_x(s-1)\ F_x(s))$ to the range $[0,1]$. After the computations, we can then perform the last step in the assignment procedure to determine the final range for the new tag:

$$U^{i+1}(m) = L^{i+1}(m-1) + [U^{i+1}(m-1) - L^{i+1}(m-1)]F_x(s_{new})$$
$$L^{i+1}(m) = L^{i+1}(m-1) + [U^{i+1}(m-1) - L^{i+1}(m-1)]F_x(s_{new}-1)$$

The tag q_{i+1} is then computed as the average of the two bounds as before. For the running example, the new value will be $q_{i+1} = \frac{1051}{2048}$. See Figure 3.6.

	a	c	a	b	d	c	a	b	d	a
U_k	1	$\frac{1}{4}$	$\frac{3}{16}$	$\frac{9}{64}$	$\frac{17}{128}$	1	$\frac{3}{4}$	$\frac{9}{16}$	$\frac{17}{32}$	$\frac{17}{32}$
L_k	0	0	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{33}{256}$	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{33}{64}$	$\frac{135}{256}$
$U_k - L_k$	1	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{64}$	$\frac{1}{256}$	1	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{64}$	$\frac{1}{256}$

Table 3.3: Upper and lower bounds on the current interval on the number line for each individual step in Figure 3.5 and Figure 3.6.

Table 3.3 shows the bounds for the two sequences. The bounds are obtained from the figures. Having determined q_i, which is fractional, we can then assign the final code for Q_i by mapping the tags to an integer in the range in $[0, n-1]$. This can be done using a simple formula:

$$c_i = \mathcal{C}(Q_i) = \lfloor (n-1)\frac{q_i - q_{min}}{q_{max} - q_{min}} \rfloor,$$

where $q_{min} = \min_i\{q_i\}$, and $q_{max} = \max_i\{q_i\}$. Notice that here, the c_i's computed will not necessarily be consecutive. But they will be ordered. The difference between c_i and c_{i+1} will depend on P_i and P_{i+1}. Alternatively, we can simply record the position where each Q_i fell on the number line. We then read off these positions from 0 to 1, and use the count at which each Q_i is encountered as its code. Since the q_i's are implicitly sorted, so are the c_i's. We have thus obtained an ordering of **all** the m-length substrings in T. This is still essentially a **partial ordering** of all the suffixes based on their first m symbols, but a total order on the m-length prefix of the suffixes.

The complexity of this procedure is again in $O(n + m + |\Sigma|) = O(n + \log n + |\Sigma|)$. The Σ component comes from the time needed to sort the unique symbols in T before

computing the *cdf*. This can be performed in linear time using counting sort. Since $|\Sigma| \leq n$, this gives a worst case time bound of $O(n)$ to compute the required codes for all the $O(n)$ m-length blocks. The space needed is also linear in $O(n+m+|\Sigma|) = O(n)$. We only need to maintain two extra $O(|\Sigma|)$ arrays, one for the number line at each step, and the other to keep the cumulative distribution function. We summarize the above results in the following theorems:

Theorem 2: Given a sequence $T = t_0 t_1 \ldots t_{n-1}$, from a fixed alphabet Σ, $|\Sigma| \leq n$, all the m-length prefixes of the suffixes of T can be ordered in linear time and linear space.

Theorem 3: Given a sequence $T = t_0 t_1 \ldots t_{n-1}$, with symbols from a fixed alphabet Σ, $|\Sigma| \leq n$, **Algorithm II** computes the suffix array of T in $O(n)$ worst case time and $O(n)$ worst case space.

Proof: Theorem 2 follows directly from the foregoing discussion. The last theorem follows, since the total time for computing the suffix arrays at all the recursion levels will be in $O(\frac{n \log n}{m})$ or $O(n)$ time. At the highest level, we can merge the suffix arrays SA_0 and SA_1 from T_0 and T_1 to form SA, the final suffix array for T in linear time, using the previous merging procedure. Although the space complexity is in $O(n)$, the actual space needed will, however, be $11n$ bytes of memory, since we will need to maintain the $n/2$ integers required for the integer codes for the m-blocks (recall, that we need to compute tags for only $n/2$ prefixes), and n integers SA_0 and SA_1, another n integers for the resulting suffix array, and n bytes for the original text. This assumes 4 bytes per integer, and 1 byte per character symbol. These are standard assumptions used by all the other algorithms. $\square$

We improve the space requirement using an alternative approach. Since we now have SA_0 and SA_1, we can use radix sort to merge the two suffix arrays, by using the sorted order in each array as the tie breaker. That is, we form an $n \times 2$ array TT, which contains the original sequence, defined as follows:

$TT[i, 0] = T[i]$, $\forall i$

$$TT[i,1] = \begin{cases} SA'_0[\frac{i}{2}] \text{ if } (i \mod 2) = 0 \\ SA'_1[\frac{i-1}{2}] \text{ if } (i \mod 2) = 1 \end{cases}$$

Then, we radix sort TT to produce SA, leading to an overall linear time construction of the suffix array of T. The algorithm requires n integers to store all the pre-computed codes for the m-length prefixes, n integers to store SA'_0 and SA'_1, and another n integers plus n bytes to store TT. However, at this point, we do not need the integer codes for the m-length prefixes, thus, we can reuse the space for the first half of the TT array. We can construct the TT array by first considering only entries from, say, SA_0. Since SA_0 is no longer needed after this, we can then re-use the space previously occupied by SA_0 to make up the second half or TT. And then, we map the SA_1 values into TT using the new space. Thus, the overall space needed is $7n$ bytes, assuming 4 bytes per integer, and 1 byte per character symbol.

3.4 Algorithm III: Generalized Case $1 : \eta$ Scheme

From the computational complexity analysis of Algorithm II, one of the primary disadvantages of Algorithm II is the binary partition scheme with limited flexibility. We propose Algorithm III which extends the partition scheme without losing the linear time sorting performance. We improve Algorithm II by modifying the selective partition ratio, namely, to the case of ratio $1 : \eta$, thus making Algorithm II a special case with $\eta = 1$. Systematically, we divide the input sequence of T into two subsequences of T_0 and T_1, where $T_0 = \{t_j, j \in [0, |T|) \mid j \bmod (\eta + 1) = 0\}$ and $T_1 = \{t_j, j \in [0, |T|) \mid j \bmod (\eta + 1) \neq 0\}$. We prove the computational complexity remains the same, being linear in the length of the input sequence no matter how the partition schemes are performed. However, the difficulty of practical implementation increases along with the increasing η partition blocks because of the increasing pre-processing block length of m.

3.4.1 Overview

In Algorithm II, we recursively perform the binary partition to the current input sequence. A non-symmetric treatment of the two partitions is applied at each recursion step. That

is, the second partition T_1 will be sorted recursively by successive partitioning. Meanwhile the first partition T_0 will be sorted according to the sorted results SA_1 from the second partition. The suffix array SA of sequence T is obtained in a bottom-up manner by the recursive merging step of its child suffix arrays SA_0 and SA_1 of T_0 and T_1 respectively. The improvement of Algorithm III is the generalized case of Algorithm II, using an adjustable partition ratio $1 : \eta$.

3.4.2 Partition Tree Height

Since the input sequence is partitioned at a $1 : \eta$ ratio for the left branch and the right branch, the longest partition branch will determine the maximum height h_{max} of the partition tree. After each partition, there will be two branches. The left branch contains a substring with the length of $\frac{\eta}{\eta+1} * |n'|$ with respect to its parent sequence n'. The right branch contains a substring with the length of $\frac{1}{\eta+1} * |n'|$. It's obvious that the right branch will exhaust the subsequence before the left branch does. Thus, the right branch will become the longest branch in the partition tree and consequently determine the height of the partition tree. Because $\frac{\eta}{\eta+1}$ of the parent sequence in length will be kept in the right branch, $n * (\frac{\eta}{1+\eta})^h$ will indicate how many symbols will be left at level h. At the bottom level, there will be at least one symbol left. For a sequence of length n with h_{max} as the total height of the partition tree, we will have:

$$n * \left(\frac{\eta}{1+\eta}\right)^{h_{max}} \geq 1$$
$$h_{max} \geq \log_{\frac{1+\eta}{\eta}}(n)$$

Similar to Algorithm II, the pre-computed information plays an essential role in the computational complexity reduction. Still, m is defined to be the size of the pre-computed block. We pre-compute a tag of each m-length prefix of each suffix from the sequence, which allows later comparison of the m-length blocks in $O(1)$ time. With the pre-computed information, it will reduce the the number of bucket sorting required at each level h from 2^{h-1} to $\frac{2^{h-1}}{m}$, each involving $2^{h_{max}-h}$ symbols. By assigning appropriate

m, we can reduce the complexity of the overall sorting procedure. If we assign m to be equal to h_{max}, the complexity would be

$$O\left(\frac{n\log n}{m}\right) = O\left(\frac{n\log n}{\log_{\frac{1+\eta}{\eta}}(n)}\right) = O(n).$$

In this scenario, the pre-computed block size is equal to $\log_{\frac{1+\eta}{\eta}}(n)$. Thus, this pre-computed block size will increase as the augment of the partition determinant η. Meanwhile, it also increases the program implementation difficulty in the arithmetic encoding part because it takes more steps to reach a decimal number. The compiler needs more floating point accuracy to perform the arithmetic encoding. In practice, m should not be arbitrarily large, since this will cause the pre-computed block size to increase indefinitely.

3.4.3 Conflict Resolution

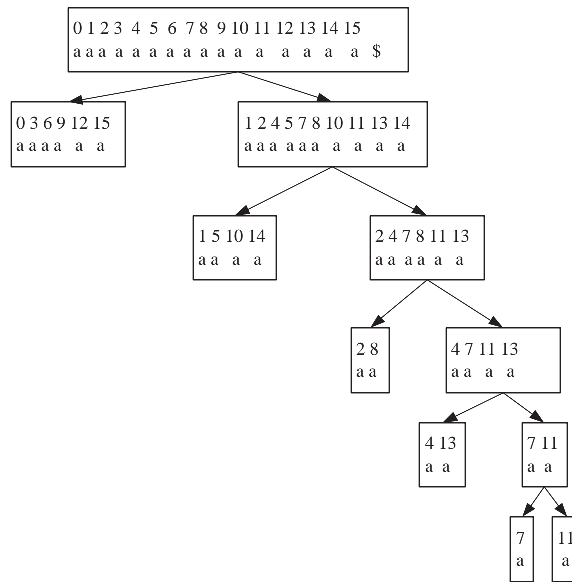

Figure 3.7: Ternary 1 : 2 partition tree with $\eta = 2$. Every $(1+\eta)$-th symbols are selected to create T_0. Solid arrows indicate partition procedure.

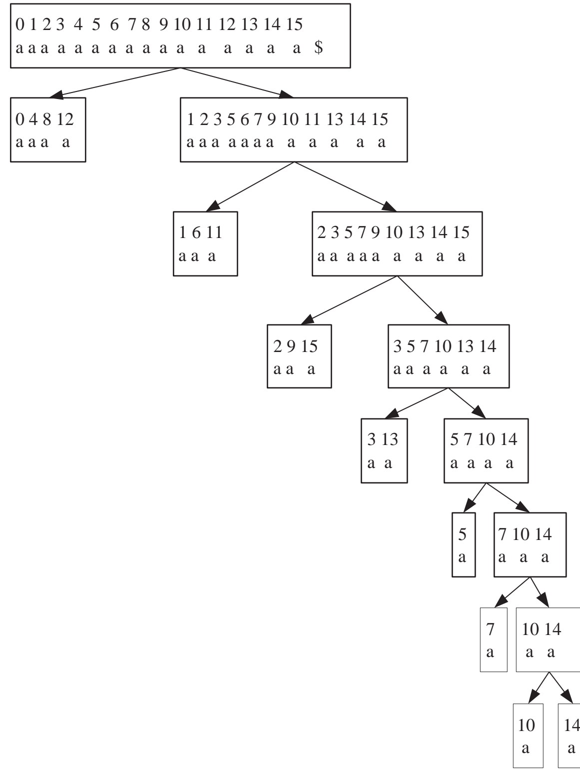

Figure 3.8: Quaternary 1 : 3 partition tree with $\eta = 2$. Every $(1 + \eta)$-th symbols are selected to create T_0. Solid arrows indicate partition procedure.

From Figure 3.7 and Figure 3.8, we know it takes no more than $(1 + \eta)^h$ steps to resolve the conflicts at level h with the partition scheme of $1 : \eta$. For example, it takes at most 3^h steps for 1 : 2 partition scheme and 4^h steps for 1 : 3 scheme . For example, at the first level of the partition tree with $\eta = 2$, the tuple pair $(t_0, SA(t_1))$ has to be modified to the triple $(t_0, t_1, SA(t_2))$ to accommodate the extended conflict cases. It can also be confirmed by the analysis in Table 3.4.

From Table 3.4 and Table 3.5, "Pos_x mod" indicates the modulus form representation of "Pos_1". The modulus form must be in the form of $\eta \cdot n + b$, where $n \in \mathbb{N}, b \in (0, 1, \ldots, \eta - 1)$. It's obvious that $(\eta \cdot n + b) \, mod \, \eta = b$ where $b \in (0, 1, \ldots, \eta - 1)$. For any $m \in \mathbb{N}, n \in \mathbb{N}$, we will know $(\eta \cdot m + b) \, mod \, \eta = (\eta \cdot n + b) \, mod \, \eta$. From Table 3.4 and Table 3.5, we know Pos_0 will take steps to reach the state of Pos_1, where the conflicts starting at Pos_0 will be at most resolved at position ending at Pos_1. By deducting the

Pos_0	Pos_0 mod	Pos_1	Pos_1 mod
3n+2	3n+2	3n+11	3(n+3)+2
3n+8	3(n+2)+2	3n+17	3(n+5)+2
3n+16	3(n+5)+1	3n+25	3(n+8)+1
3n+22	3(n+7)+1	3n+31	3(n+10)+1

Table 3.4: The merge routine for ternary partition scheme at level 2. Every *Pos* share the same modulus.

Pos_0	Pos_0 mod	Pos_1	Pos_1 mod
3n+4	3(n+1)+1	3n+31	3(n+10)+1
3n+13	3(n+4)+1	3n+40	3(n+13)+1
3n+25	3(n+8)+1	3n+52	3(n+17)+1

Table 3.5: The merge routine for ternary partition scheme at level 3. Every *Pos* share the same modulus.

difference between Pos_0 and Pos_1, you can calculate the upper bound of how many steps Algorithm III will take at most to resolve the conflict at each height level of h. The difference is given by $(1 + \eta)^h$. For example, in Table 3.4, $3n + 2$ will take $(1 + 2)^2 = 9$ steps to reach $3n + 11$ so as to resolve the conflicts when $h = 2$ with $\eta = 2$. In Table 3.5, $3n + 4$ will take $(1 + 2)^3 = 27$ steps to reach $3n + 31$ so as to resolve the conflicts when h is equal to 2 with $\eta = 2$.

3.5 Concluding Remarks

We propose a divide-and-conquer sort-and-merge algorithm for performing direct suffix sorting on a given input string. We analyze the practical performance of the proposed direct suffix sorting algorithm. We improve the practical space requirements and reduce the actual number of symbol-wise comparisons. Meanwhile, the current conflict resolution involves the arithmetic coding technique. As an inherent disadvantage of the arithmetic coding method, the division precision problem might not be handled properly. Thus, an

interesting future work would be to investigate other potentially more practical methods for computing the partial order, perhaps using *lcp* pre-computing.

Chapter 4

Multiple Sequence Alignment

4.1 Overview

Alignment of multiple biological sequences is one of the central problems in computational biology [14, 76]. Multiple sequence alignment has applications in various problems in biology, and can be applied respectively to DNA, RNA or protein sequences. Example applications can be found in whole-genome sequencing, identifying conserved regions and regulatory elements in related genomes [77], determining important regions for cite-directed mutatgenesis [78], construction of and analysis of phylogenic trees, phylogenic footprinting [5], etc.

Given m sequences, $(m \geq 2)$, $C = \{T_0, T_1, \ldots T_{m-1}\}$, an alignment of the sequences is obtained by inserting gaps at chosen positions in each of the m sequences, such that the resulting sequences all have the same length, say l. The sequences are thus arranged into an array of $m \times l$ symbols, whereby no column is allowed to contain only gaps. The goodness of an alignment is measured by *the alignment value*, which is determined using a scoring function. Let $C' = \{T'_0, T'_1, \ldots T'_{m-1}\}$, be the set of resulting sequences after an alignment. The value of the alignment is then given by: $V(C') = \sum_{i=1}^{l} v(T'_0(i), T'_1(i), \ldots T'_{m-1}(i))$, where $v(T'_0(i), T'_1(i), \ldots, T'_{m-1}(i))$ is the alignment value as given by a defined scoring function, and $T'_k(i)$ is the i-th symbol in the resulting aligned sequence T'_k. The problem is therefore to determine the optimal alignment, that is, a multiple sequence alignment

[1]Part of the work reported in this chapter has appeared in the papers [74, 75].

that will result in a minimal alignment value. The result is that the sequences are now arranged such that the maximal similarity between them are now exposed, and can then be exploited for various applications.

The problem of computing the optimal alignment for multiple sequences is known to be NP-complete [79, 80], and thus various heuristics and approximations have been proposed. Most methods are typically based on either global alignments [50], or local alignments [51]. Hybrid approaches aimed at integrating both local and global methods have also been reported [54,81]. Example popular alignment programs include CLUSTAL W and its family [3, 4], T-COFFEE [54], DIALIGN and its derivatives [1, 2], MUSCLE [55], Align-m [82], GAME [83], etc. A survey of alignment algorithms is given in [54], while empirical benchmarking of different algorithms have been performed in [84–86]. Gusfield [14] and Mount [16] provide a detailed discussion on the general problem of sequence alignment.

We propose a sort based method for constructing multiple sequence alignments. Our motivation is the recent result in linear time direct suffix sorting [33,47,48], whereby suffix arrays can be constructed *directly* in linear time and space in the worst case, without the need to first construct suffix trees. This has important implications for both the time required for computing the alignment, and the space needed for constructing the suffix arrays. We perform alignment by first identifying a set of anchor points, based on the suffix sorting output on the sequences. Final alignment and gap considerations can be made by a recursive application of the sort-based anchor point algorithm.

An important novelty in our approach is how biological mutation information is incorporated in each refinement step. Thus, the recursive alignment refinement is performed via sorting and alignment between anchor points using a mutated version of the original sequences. This sort-mutate-anchor paradigm separates our proposed sequence alignment algorithm from other previously proposed techniques.

4.1.1 Anchor Based Methods

Another approach that has been used to reduce the complexity of multiple sequence alignment is by the use of selected anchor points along the sequences. An anchor point

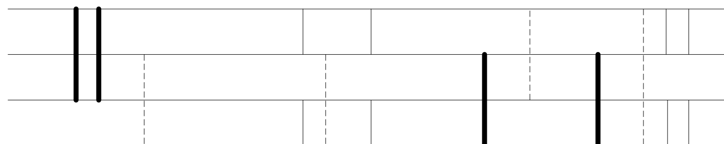

Figure 4.1: Schematic overview of anchor points along the sequences to be aligned. The horizontal lines represent the source sequences to be aligned. The vertical solid lines represent higher priority anchors with stronger constraints across different sequences. The dashed lines represent lower priority anchors with weaker constraints. The intersection between horizontal and vertical lines represent the prioritized anchor points, which play an essential role in multiple sequence alignment.

is defined as a group of symbols with high local similarity across multiple sequences. The anchor can be characterized by the sequences involved in the anchor, the starting position of the anchor in the sequences, and the length of the anchor. In some situations, especially with user-defined anchors, the anchors are given priorities [5], which can help the algorithm in resolving consistency between the anchors. Anchor priorities can also be used to determine which anchor points should be chosen first, where there are multiple anchor points in the same region. Anchors help to restrict the search required in performing the alignment, by constraining the required search to only regions between two anchor points. Clearly, the performance of anchor-based algorithms depends critically on how the anchor points are chosen, and to some extent, on how the alignment between anchor points is performed. Figure 4.1 shows a simple schematic diagram of anchor-based alignment. The sequences involved in the alignment are represented as horizontal lines. The vertical lines indicate where an anchor has been established between sequences. Solid vertical lines represent strong anchors, while the dotted lines depict weak anchors.

Various anchor-based methods have been proposed [10,83,87,88]. The methods differ in the way the anchor points are defined, in how they are computed, whether or not priorities are assigned, and in how the remaining sequences between anchors are aligned. In [89], the clusters observed in the sorted common prefix (SCP) data structure were proposed for determining regions for anchor point computations, using the notion of sphere of influence for the anchor points. In [10], anchor-based multiple sequence alignment was performed in three basic steps as shown in Algorithm 1.

Algorithm 1 Anchor Based Alignment Scheme

ANCHOR-BASED-ALIGN (T)
1 **Begin**
2 Compute the maximal unique matches (MUMs)
3 Compute the longest common subsequences based on the MUMs, using the longest increasing subsequence algorithm [14]
4 Close the gaps in the alignment by identifying local repeats or small mutations
5 **End**

A maximum unique match (MUM) is a subsequence that occurs only once in each of the sequences involved in the alignment, and it is not contained in any other MUM. Here, the MUMs are used to form anchor points for the alignment. The use of MUMs follows the idea used in popular alignment programs such as BLAST and FASTA. Conceptually, if there is an MUM between the input sequences, it is most likely to be part of the global alignment. One problem with MUM-based alignment is that it does not work well in aligning sequences from relatively distant species. The basic idea has been modified, for instance, using criteria different from MUMs. Example, GAME [83] used maximal exact matches (MEMs), rather than MUMs, while in LAGAN [87], Brudno et al used the idea of (q, k)-anchors, whereby anchor points are defined based on inexact matches. For two sequences, the (q, k)-anchor is defined as a pair of length q substrings matching with at most k errors between them.

In ACANA [88], anchors were determined by using the edit matrix produced during dynamic programming. By choosing near-optimal local alignments along the diagonals of the dynamic programming matrix, they recursively determined the best anchors for global alignment for each region, which are then fixed for the region. Final alignment is produced by connecting each region with the fixed anchor points.

Another way to view the anchors is in terms of constraints on the sequences to be aligned. Such constraints could be user-defined, or could be machine generated based on some application dependent information. Myers et al [90] proposed methods for progressive alignment incorporating sets of constraints, and studied the issue of consistency between constraints. Brown and Hudek [91] proposed a tree-based method for anchor-based progressive alignment using spaced or seeded anchors. Spaced anchors [92–94] are

constraints in the alignment defined in terms of a binary pattern over the sequences, such that matches or mismatches can be allowed at a given position along the sequences, based on whether we have say a **1** or a **0** at the given position. Sequence alignments using explicit user defined anchors were studied in [5].

While user-defined constraints can be quite important in some applications, this makes the methods less flexible, especially in using the algorithms on applications different from their original purpose. Moreover, for very large sequences, for instance, whole genomes, or when many sequences are involved, defining these manual anchor points may become difficult. Automated anchor point determination will thus be needed in such situations. Such automated methods should be able to provide options for augmenting the automatically determined anchor points with user-specified anchors. Similar ideas can be used to incorporate further expertise, for instance, knowledge of locally conserved protein secondary structures, as constraints in the alignment [95].

4.1.2 The Problem and Main Contribution

Three major problems in multiple sequence alignment can be identified: (1) gap induction; (2) ensuring biological relevance, for instance using known mutation (substitution) matrices; and (3) alignment scoring functions. Problem (3) often involves aspects of problem (2), in addition to considerations for gap costs. While (2) and (3) are needed for accurate and more sensitive alignment, the computational problem in multiple sequence alignment is often due to the model used for inducing gaps during the alignment. This is because the optimal alignment often requires an exhaustive evaluation of all the possibilities for gap insertion in the sequences being aligned. In fact, from one view point, the problem of sequence alignment can be viewed as that of appropriate gap induction within the sequences involved. For anchor-based alignment schemes, an additional problem is the selection of anchors points, and the development of appropriate anchor scoring functions to be used.

The major contribution of this work is a mechanism to cut down on the exhaustive search required for gap induction. The method we propose is anchor-based, whereby anchors are determined based on the sorted suffixes and position-based analysis. In

particular, we propose a sort-based alignment algorithm, using methods of linear-time worst-case suffix sorting as our basic building block. The sorts implicitly expose where alignments should be anchored, and which regions are not likely to form an alignment, thereby eliminating the need for exhaustive gap insertion and alignment score evaluation. We also develop a new anchor-point evaluation scheme based on the sorted suffixes, position-based sorts, and a proposed mutation mapping scheme.

After the initial sorting on the original sequences, we extend the sort-based approach using a novel mutation mapping scheme. Here, symbols are mutated based on a mutation map derived from standard mutation matrices, and the sorting procedures (suffix sorting and position-based sorts) on symbols between anchors are now performed on these mutated symbols. New anchor points are then determined within the previous anchors. Fixing the anchor points imply that subsequent steps cannot not change these anchors. Later alignment steps can only be performed on symbols within the same anchor region. Operations on symbols in two different anchor regions are forbidden. Sequence alignment is then performed as a recursive application of this sort-mutate-anchor paradigm on the symbols between each pair of anchor points.

4.2 Sort-based Alignment Algorithm

We propose a sort based multiple sequence alignment algorithm. Based on the observation that a well matched alignment is also a rearrangement of highly repetitive segments and non-repetitive region across all the source sequences, the algorithm transforms the alignment for multiple sequences into a long range repetitive pattern identification problem where the advantage of the automated suffix sorting technique can be fully exploited. Since suffix sorting could reveal the repetition information across all sequences, we can home in on the anchor points with the aid of suffix arrays. Figure 4.1 shows a schematic of sample anchor points among the source sequences. By constructing both strong and weak connections, the algorithm builds restraints to cross and align different sequences by their anchor points. By recursive repetition of the procedure using information from biological substitution matrices, the final alignment for multiple sequences can be obtained.

4.2.1 Notation

The algorithm performs multiple sequence alignment (MSA) on a set of m sequences Π from an alphabet Σ, where $\Pi = \{T_0, T_1, T_2, \ldots, T_{m-2}, T_{m-1}\}$. The input sequences Π which are passed to the algorithm are called source sequences. $|\Sigma|$ denotes the size of the alphabet and $|T_h|$ is the length of a specific sequence T_h. $S(T_h, t_k) = t_k t_{k+1} t_{k+2} \ldots t_{|T_h|-1}$ is defined as the suffix in sequence T_h, starting from position k to the end of sequence T_h, where $0 \leq k \leq |T_h|$ and $0 \leq h \leq (m-1)$. An example showing the relationship between T_h and $S(T_h, t_k)$ is listed below.

$$\begin{aligned} T_0 &= BIKQMVWHAQKCM \\ S(T_0, t_5) &= VWHAQKCM \end{aligned}$$

Concatenation is defined as the the operation of joining multiple character strings end to end by appending T_{i+1} to the end of T_i for any i, where $0 \leq i \leq (m-1)$, i.e., $T = \{T_0 \circ T_1 \circ T_2 \circ \cdots \circ T_{m-2} \circ T_{m-1}\}$, where $\circ$ is the concatenation operator. For example, the strings T_0, T_1 and T_2 may be concatenated to give T. We have embedded some special end-of-sequence symbols to each source sequence. The source sequences are joined together end to end to form a new sequence of T. **Example**.

$$\begin{aligned} T_1 &= BIKQMVWHAQKCM\#,\ T_2 = BEVRLYFNAXZZC\&, \\ T_3 &= XBCDDEFEENADVRZMC\$ \\ T &= BIKQMVWHAQKCM\#BEVRLYFNAXZZC\&XBCDDEFEE \\ & \quad NADVRZMC\$ \end{aligned}$$

In multiple sequence alignments, there will be some well aligned subsequences across different source sequences. The aligned symbols across multiple sequences are defined as anchor points. For each anchor point, there is an anchor score associated with it to indicate how well the anchor point can be used to align source sequences. A tightly

related group of anchor points is used to indicate a strong localized correlation. The anchor score is computed as the summation of the anchor scores for all the anchor points within the defined region of symbol-wise distance Q. A filled backbone sequence is the resulting alignment sequence, onto which the anchor points are mapped so as to present the final alignment for source sequences. The initial blank backbone sequence is the primitive structure prior to alignment, which is filled by anchor points in a step by step manner.

4.2.2 Overview

We provide a general outline of the proposed algorithm in Algorithm 2. It includes subroutines of the source sequence concatenation, suffix sorting, anchor points identification and biological substitution matrices decomposition. The proposed algorithm takes advantage of suffix sorting by converting the inter-sequence vertical alignment problem to one dimension horizontal repetitive pattern identification using suffix sorting. The general idea is to concatenate multiple source sequences $\Pi = \{T_0, T_1, T_2, \ldots, T_{m-2}, T_{m-1}\}$ into a single sequence T with special end-of-sequence symbols, then perform the suffix sorting on T. This sorting operation exposes the repetitive segments across the source sequences $\Pi = \{T_0, T_1, T_2, \ldots, T_{m-2}, T_{m-1}\}$. We evaluate the suitability of these repetitive segments for anchor points formation by computing their anchor scores. The segments with anchor scores above a threshold are marked as potential anchor regions. The starting position of each potential anchor region is a potential anchor point. The maximum valued anchor points are first selected and fixed onto the blank backbone sequences. Recursively, a less tightly correlated anchor point identification subroutine is performed between those previous identified anchor regions until all the anchor regions are exhausted. Details of each subroutine is given in the following sections.

Algorithm 2 Proposed Sort-Based Multiple Sequence Alignment

```
SORT-BASED-ALIGN(Π)
1  Begin
2     Concatenate Sequences in Π to form T
3     Suffix-Sort(T)
4     For each region T′ between two consecutive anchor points
5        Begin
6           Substitution-Decomp(T′)
7           Suffix-Sort(T′)
8        End
9  End

SUFFIX-SORT(T)
10 Begin
11    Perform Suffix Sorting on T
12    Perform position based sort on T
13    Compute anchor points on T
14    Return
15 End

SUBSTITUTION-DECOMP(T′)
16 Begin
17    Perform mutation mapping on T′ using Φ
18    Return
19 End
```

4.2.3 Anchor Points Selection

The sequence T is constructed by appending T_{i+1} to the end of T_i for each i, where $0 \leq i \leq (m-1)$, i.e., $T = T_0\$T_1\# \ldots T_{m-1}\%$, where \$, #, and % are special end-of-sequence symbols. A one-to-one mapping relation between Π and T has been noted from the sequence concatenation operation, i.e., $T_s \Leftrightarrow (T_h, t_k)$ for any $T_s \in T$ and $(T_h, t_k) \in \Pi$, where $0 \leq h \leq (m-1), 0 \leq k \leq |T_h|, s = k + \sum_{i=0}^{h-1}(|T_i|+1)$. A set Π' of four source protein sequences used in [5] are listed in Figure 4.2. The suffix sorting routine is then applied on the simple sequence of T. This will produce the suffix S which contains all the suffixes in T, sorted according to their alphabetical orders in Σ. Essentially, T will be

partitioned into $|\Sigma|$ groups, whereby each group contains suffixes in T that start with the same symbol. The sorted suffixes , starting with symbol σ from the four sample source sequences Π', are denoted as σ-block. The N group is listed as well as (h, k) and suffix array information in Figure 4.3. Particularly, only the first r symbols of suffixes within the N-block are shown, $r = 5$ in Figure 4.3. We mark k value as ID for all suffixes and re-sort them by k within the group. This will create the re-sorted suffixes as the second column in Figure 4.3. Given the one-to-one mapping between T and (T_h, t_k), each suffix S_s in S can be marked with its corresponding (T_h, t_k) pair.

```
1r69 :SISSRVKSKRIQLGLNQAELAQKVGTTQQSIEQLENGKTKRPRFLPELASALGVSVDWLLNGT
1au7A:GMRALEQFANEFKVRRIKLGYTQTNVGEALAAVHGSEFSQTTICRFENLQLSFKNACKLKAILSKWLEE
      AEQKRRTTI
1neq :CSNEKARDWHRADVIAGLKKRKLSLSALSRQFGYAPTTLANALERHWPKGEQIIANALETKPEVIWPSR
1a04A:ERDVNQLTPRERDILKLIAQGLPNKMIARRLDITESTVKVHVKHMLKKMKLKSRVEAAVWVHQERIF
```

Figure 4.2: A set of four DNA-binding protein residues Π using the same data set used in [5]. They are concatenated to form a single sequence of T by appending T_{i+1} to the end of T_i for each i, where $0 \le i \le (m-1)$.

A sliding window Q is introduced to elaborate the anchor point selection process. Q is defined as a covered region of k such that all suffixes that fall into this region can be considered in determining the anchor point. This region is marked between Q_{start} and Q_{end}, which indicates the boundary of the potential anchor points. Thus, it makes the algorithm a local alignment scheme. Q will cover the inclusive region $[Q_{start}, Q_{end}]$, namely $|Q| = Q_{end} - Q_{start} + 1$. The default value of Q is set to $|\Sigma|$ since this reflects the most general case of uniform distribution for symbols in the biological sequences. The anchor score for the anchor point within the region Q is computed as shown in the downward sliding window of Figure 4.3. Let r denote the sliding window size. Initially, Q covers the suffix of $S(h_i, k_i)$, $S(h_{i+1}, k_{i+1})$, $\ldots$, $S(h_{i+(r-1)}, k_{i+(r-1)})$, where $Q_{start} = k_i$ and $k_{i+j} \in [Q_{start}, Q_{end}]$ for all j, $0 \le j \le (r-1)$. Since $Q_{start} = k_i$, we can use the first suffix to represent the region Q. $Q(h, k)$ denotes the region of covered suffixes, which starts from suffixes $S(h, k)$. In Figure 4.3, $Q_{start} = 3$ because $k_0 = 3$. $Q_{end} = Q_{start} + |Q| = 23$. Thus, the region $Q(2, 3)$ will cover all the suffixes

(h,k)	Sorted Suffixes	Suffix Arrays	ID
(1,54)	NACKL	24	54
(2,40)	NALER	29	40
(2,55)	NALET	14	55
(1,9)	NEFKV	69	9
(2,2)	NEKAR	67	2
(0,37)	NGKTK	28	37
(0,60)	NGT$$	3	60
(3,23)	NKMIA	44	23
(1,46)	NLQLS	31	46
(0,15)	NQAEL	48	15
(3,4)	NQLTP	63	4
(1,24)	NVGEA	54	24

(a)

(h,k)	Re-sorted Suffixes by k
(2,2)	NEKAR
(3,4)	NQLTP
(1,9)	NEFKV
(0,15)	NQAEL
(3,23)	NKMIA
(1,24)	NVGEA
(0,37)	NGKTK
(2,40)	NALER
(1,46)	NLQLS
(1,54)	NACKL
(2,55)	NALET
(0,60)	NGT$$

(b)

Figure 4.3: The N-block for sequences used in Figure 4.2. (a) The panel table is sorted suffix according to their alphabetical order. (b) The panel table is sorted by their position k. After the sorted symbols are obtained, a sliding window of size Q is scanned vertically down the sequence segments. For each sliding window, the number of distinct symbols across sequence segments are computed, which is used to compute the anchor score for the corresponding anchor points. $ is the end-of-sequence marker.

with a k value, where $k \in [Q_{start}, Q_{end}]$, i.e., suffixes of $(2,3)$, $(3,5)$, $(1,10)$ and $(0,16)$ will be covered by $Q(2,3)$ in the first step because all the k values for these suffixes are within the region of $[Q_{start}, Q_{end}] = [3, 23]$. Let $\varphi(h_i, h_{i+1}, \ldots, h_{i+(r-1)})$ denote the number of distinct values in the array $[h_i, h_{i+1}, \ldots, h_{i+(r-1)}]$. After the determination of sliding window $Q(2,3)$, the anchor score for this potential $Q(2,3)$ can be computed using Equation 4.1:

$$\begin{aligned} C(h,k) &= \alpha_1 \cdot C_1 + \alpha_2 \cdot C_2 + \alpha_3 \cdot C_3, \text{where } \alpha_1 \gg \alpha_2 \gg \alpha_3 \qquad (4.1) \\ C_1 &= \varphi(h_i, h_{i+1}, h_{i+2}, \ldots, h_{i+(r-1)}) \\ C_2 &= \sum_{j=0}^{r-1} ((r-j)^2 \cdot k_{i+j}) \\ C_3 &= \sum_{j=0}^{r-1} (|k_{i+j} - \bar{k_i}|), \text{where } \bar{k_i} = \frac{\sum_{l=0}^{r-1} k_{i+l}}{r} \end{aligned}$$

In the first sliding window $Q(2,3)$, $\varphi(h) = 4$ for all $h \in Q(2,3)$. $Q(h,k)$ is then shifted to

the the next suffix by incrementing Q_{start} to $k+1$, namely sliding down the window by one position in Figure 4.3. The new $Q(3,5)$ covers the regions $[Q_{start}, Q_{end}] = [5, 25]$. Thus, suffixes of $(3,5)$, $(1,10)$, $(0,16)$, $(3,24)$ and $(1,25)$ are covered by $Q(3,5)$. In the sliding window of $Q(3,5)$, $\varphi(h) = 3$ for all $h \in Q(3,5)$. A similar computation of anchor score $C(h,k)$ can be obtained from Equation 4.1 as well. $\varphi(h)$ values are listed to the right of each sliding window in Figure 4.3. The anchor score computation steps are repeated until the sliding window Q reaches the bottom of the N-block for the example in Figure 4.3. The algorithm will proceed to the next group after it finishes the N-block until all groups are processed.

In reality, the computation of C_2 and C_3 might not be performed if

$$\varphi(h_i, h_{i+1}, \ldots, h_{i+(r-1)})/m \geq \beta$$

where β is a threshold. The algorithm emphasizes the impact of C_1 more than C_2 and C_3. It also weights on C_2 more than C_3. This allows the algorithm to surpass C_2 and C_3 if C_1 suffices to determine the significance of the anchor point. In a tie situation with C_1, the value of C_2 still suffices to circumvent the computation of C_3 in order to determine the anchor point. C_3 will be computed only if using C_1 and C_2 leads to a tie and cannot tell the significance of the anchor point.

Figure 4.4 shows an anchor point determination process, where the source sequences are on the top of the diagram and the aligned output sequence are at the bottom of the diagram. The suffix sorting routine is applied on sequence T. Suffix S is marked as being sorted, based on the alphabetical order in Σ and (h,k). S_s is marked with its corresponding (h,k). The intermediate results are shown on the three vertical columns of Figure 4.4 in decreasing order of $\varphi(h)$. In the first vertical column, $\varphi(h) = 3$, which makes them good candidates for the potential anchor points. Since $\varphi(h)/m = 1$, the computation of C_2 and C_3 need not be performed for the first vertical column. B group, A group and C group are marked as strong anchor points. In the second vertical column, $\varphi(h) = 2$. Thus the algorithm has to check if $\varphi(h)/m$ passes a certain threshold, which makes them good candidates for the potential anchor points. The computation of C_2 and C_3 will be introduced in this scenario to further measure these potential anchor

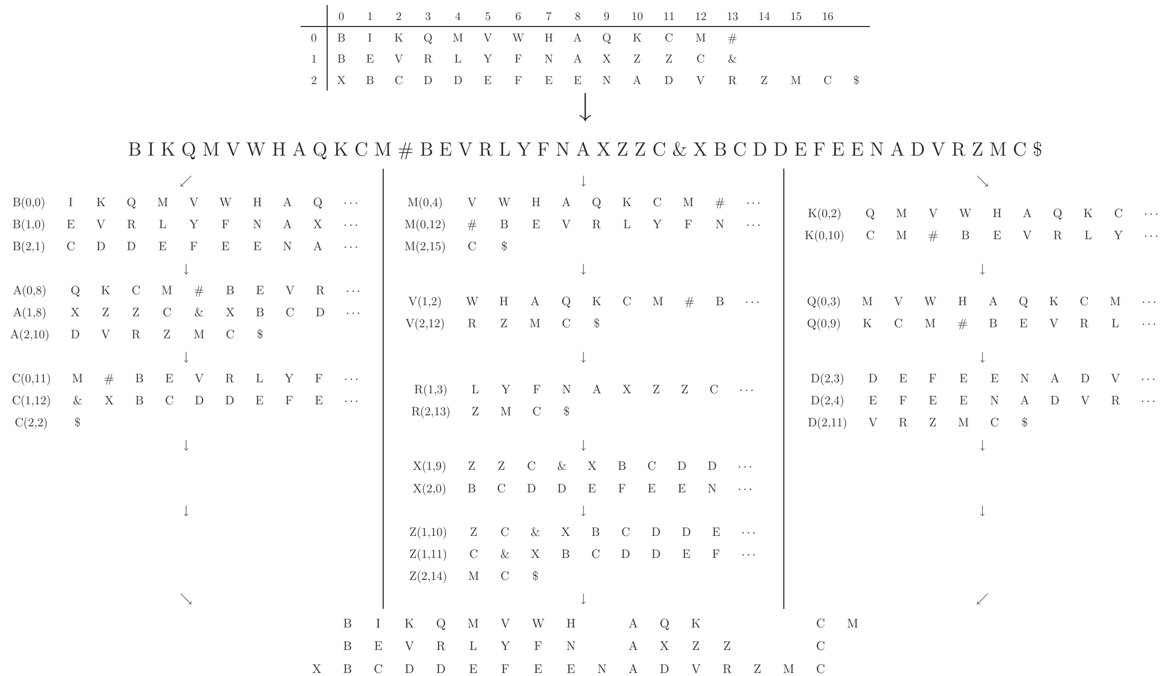

Figure 4.4: Suffix sorting routine. The direct suffix sorting is applied on the source sequences. The strongest anchor points $B - A$ and $A - C$ are identified. The source sequences are fixed by the strongest anchor points. $\varphi(h) = 3$ for the left most column. $\varphi(h) = 2$ for the central column. $\varphi(h) = 1$ for the right most column.

points. The M-block, V-block, R-block, X-block and Z-block are then marked as weak anchor points in Figure 4.4. In the third vertical column of Figure 4.4, $\varphi(h) = 1$. Thus the algorithm discards them as candidates for the potential anchor sites because no subsequence can be aligned. The aligned sequences at the bottom of Figure 4.4 correspond to the result after the strong anchor point determination, where B-block, A-block and C-block are anchored.

4.3 Mutation Mapping

4.3.1 Motivation

Figure 4.4 is the result after direct anchor point determination process. One of the disadvantages of direct anchor point sorting is that it only identifies the strongest anchor points across the source sequences. The implicit structural and functional significance of biological sequences cannot be fully exploited. We introduce the biological mutation mapping matrices of BLOSUM [7] and PAM [6] to further refine the multiple sequence alignment results between two consecutive strong anchor points, for instance the regions between B-block and A-block or A-block and C-block in Figure 4.4. These matrices calculate a log-odds score for all possible substitutions of the 20 standard amino acids with typical applications in sequence alignment and multiple sequence alignment. Both BLOSUM62 and PAM250 are two widely used biological mutation matrices. BLOSUM62 is calculated by using the observed substitutions between proteins which have 62 or more sequence identity. PAM250 means 250 substitutions per hundred amino acids. which gives the different rates of mutations between pairs of amino acids.

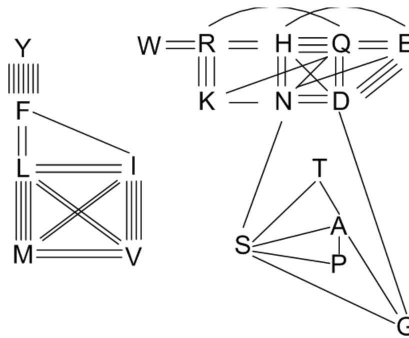

Figure 4.5: Biological mutation graph for PAM250 matrix [6]. More lines between symbols imply a stronger probability of mutation from one symbol to the other.

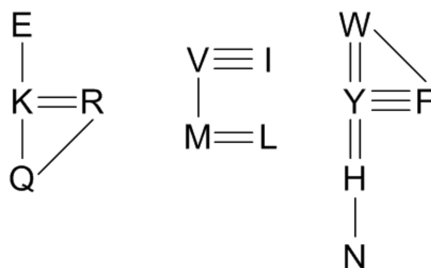

Figure 4.6: Biological mutation graph for BLOSUM62 matrix [7]. More lines between symbols imply a stronger probability of mutation from one symbol to the other.

4.3.2 Computing the Mutation Map

After obtaining the preliminary results in Figure 4.4, the mutation-based suffix sort will continue to further refine the segmental alignment between two consecutive pre-determined anchor points. The biological mutation matrix M will be introduced at this stage. $M(a, b)$, a $|\Sigma|$ by $|\Sigma|$ matrix where $0 \leq a, b \leq |\Sigma|$, denotes the different rates of mutations between pairs of amino acids. Notice M is a symmetric matrix, i.e., $M(a, b) = M(b, a)$ for any a and b where $0 \leq a, b \leq |\Sigma|$. V is defined as a subset of the alphabet Σ, where there exists an $M(a, b)$ and $M(a, b) > 0$ for any $a \in V$ and $b \in \Sigma$. We map all the symbols in V to vertices V' in graph G'. We also create $M(a, b)$ parallel edges between vertices a and b, where $M(a, b) > 0$ and $a, b \in V'$. Figure 4.5 shows the graph G' for PAM250 matrix. Figure 4.6 shows the graph G' for BLOSUM62 matrix. Graph G' can also be represented by value κ, which indicates how many parallel edges between two vertices in V'. We can group the edges by κ. Figure 4.7 shows the corresponding κ table for BLOSUM62 matrix. Because the symmetric property of matrix M, we define the mutation from alphabetically greater symbol to alphabetically smaller symbol as shown in Figure 4.7. We define κ group to contain all mutated vertices where there are κ parallel edges between each other. For example, $\kappa = 3$ group lists all the mutated vertices where there are 3 parallel edges between them in Figure 4.6. The κ grouping table is to be used in the mutation-based suffix sorting routine. This κ grouping table depends on the biological mutation table only and it is independent of the input source sequences. The κ table will be produced once during the alignment procedure, or can be pre-stored before the alignment starts.

Triple Reduction			Double Reduction			Single Reduction		
κ	$=$	3	κ	$=$	2	κ	$=$	1
V	$\mapsto$	I	Y	$\mapsto$	H	H	$\mapsto$	N
Y	$\mapsto$	F	K	$\mapsto$	R	E	$\mapsto$	K
			M	$\mapsto$	L	K	$\mapsto$	Q
			W	$\mapsto$	Y	Q	$\mapsto$	R
						V	$\mapsto$	M
						W	$\mapsto$	F

Figure 4.7: κ mutation table for BLOSUM62 matrix. κ group is defined to contain all mutated vertices where there are K parallel edges between each other, i.e., $\kappa = 3$ group lists all the mutated vertices where there are 3 parallel edges between them in Figure 4.6. The κ grouping table is to be used in the mutation-based suffix sorting routine and depends on the biological mutation table only and it is independent of the input source sequences. It will be produced once during the alignment procedure, or can be pre-stored before the alignment starts.

4.3.3 Mutation Procedure

Because the strong anchor points have been fixed in the first suffix sorting routine, the mutation based suffix sorting will be applied on the regions between two consecutive strong anchor points in a decreasing order of κ. The suffix sorting routine is applied on the regions after each κ group mutation operation. The mutation information will be written to a supplemental table in order to recover the mutated sequences back to the original symbols during the finalizing steps.

Figure 4.8 and Figure 4.9 shows the mutation based suffix sorting routine after the determination of mutation table. The maps $V \mapsto I$ and $Y \mapsto F$ ar $\kappa = 3$ group are first applied on the region between pre-determined anchor points B and A as shown in Figure 4.8. A suffix sorting routine is performed after the mutation. This will anchor the N-block in the regions. Then $K \mapsto R$, $M \mapsto L$, $W \mapsto E$ and $H \mapsto F$ in $\kappa = 2$ group are applied on the region. A suffix sort after the mutation then produces the anchor points of L and F. Continuously perform the mutation, the algorithm will reach a stage when no more anchor sites can be found to achieve a positive profit gain. At this stage, for any regions between two adjacent anchor sites, the blank gap spaces are always at the

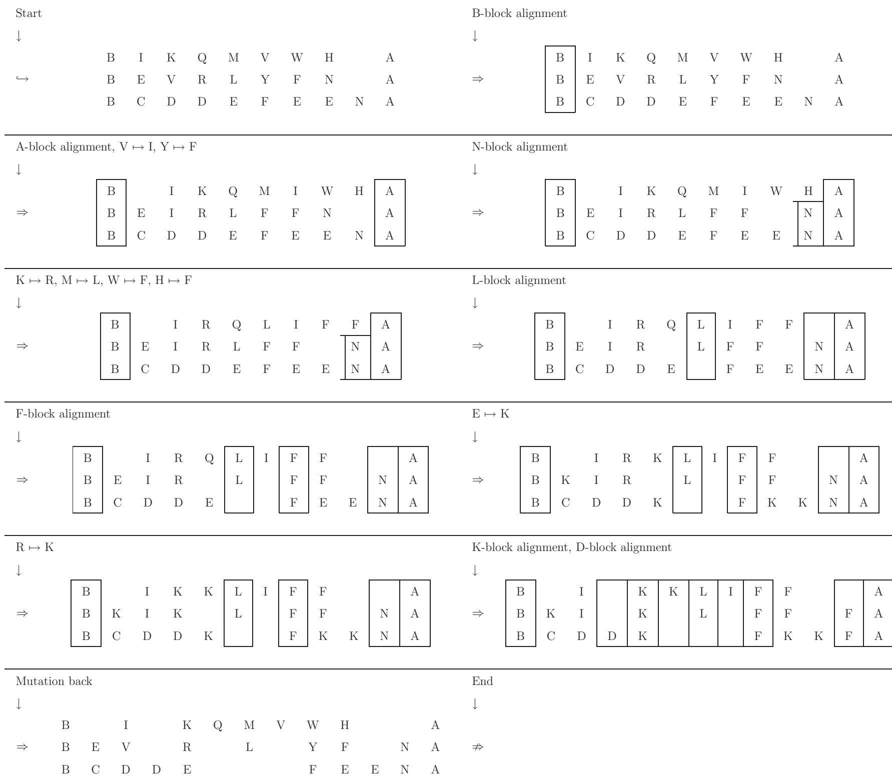

Figure 4.8: Intermediate mutation based suffix sorting alignment results between $B - A$ anchor point. The biological mutation matrix mapping is performed in a descending order of κ.

end of the segment because whenever the algorithm tries to align a local anchor site, it always shifts the anchor sites to the left. Following the mutation information in Figure 4.7, the algorithm performs the mutation once for each unique κ value and stores the mutation information as well as (h, k) in the complementary table. The complementary table will be used to recover the alignment to the original input source sequences after all the κ-level mutations have been completed. The mutation is performed in a decreasing order of κ such that the symbols with stronger affinity will be anchored first.

Each subsequence is recursively aligned within its individual anchor region following the mutation based suffix sorting routine. After each regional alignment is achieved, they are concatenated based on the strongest anchor points. Figure 4.10 shows a final result for the source sequences on top of Figure 4.4. The final results are obtained through concatenation of the regional alignment results from anchor points $B - A$ and $A - C$.

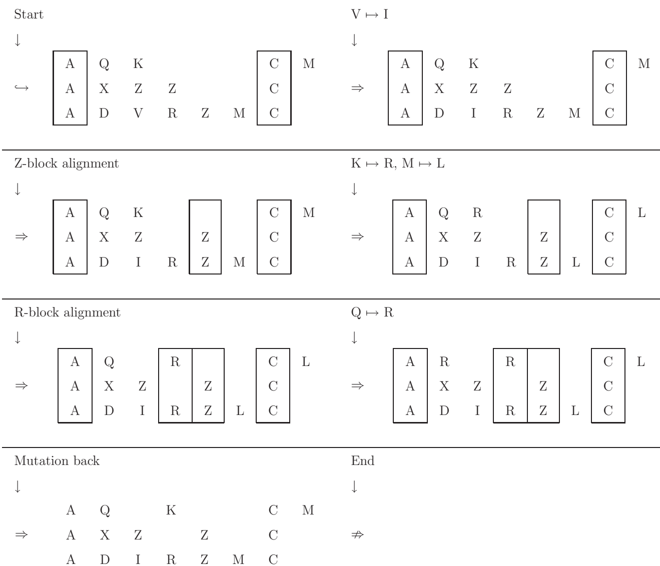

Figure 4.9: Intermediate mutation based suffix sorting alignment results between $A - C$ anchor point. The biological mutation matrix mapping is performed in a descending order of κ.

	B		I		K	Q	M	V	W	H			A	Q		K			C	M
	B	E	V		R		L		Y	F		N	A	X	Z		Z		C	
X	B	C	D	D	E				F	E	E	N	A	D	I	R	Z	M	C	

Figure 4.10: The final concatenated results with biological mutation matrix information. Notice how the current method aligns more biologically similar protein on the same column.

4.3.4 Gap Handling

The traditional bottleneck in sequence alignment is the search for gap positions. Notice that we do not perform direct search for best positions for gap insertion at each stage. With our approach to gap induction, it means that the traditional problem of exhaustive search for best positions for gap insertion has now been transformed to the problem of searching for best anchor points. The problem of best positions for gap insertion is handled implicitly by the sorting stage and the mutation-mapping procedure.

As the sliding window Q moves along the anchor sites, some anchor points might be found to be closer to some previously fixed anchor sites on the consensus sequence. The proposed algorithm will meet the creation of both anchor sites by insertion of gaps.

E		E	-
E		E	-
G	$\Longrightarrow$	-	G
G		-	G
E		E	-

Table 4.1: The proposed algorithm introduces gaps to align two anchor points which were originally on the same vertical location on the backbone sequence.

The induction of gaps avoid the possibility of assigning two anchor points on the same location of the consensus sequence. Whenever two anchor points are identified on the same location of the consensus sequence, the alphabetically smaller anchor point is fixed at the current location and the alphabetically greater anchor point is fixed at the location next to the current one to the right. The algorithm fills the corresponding space with gaps across different sequences.

4.4 Complexity Analysis

The time complexity for the proposed algorithm is $O(mq^2QN)$, where m is the total number of sequences in Π, q is the length of the horizontal sliding window, Q is the size of the vertical sliding window and N is the total length of the concatenated sequences. It costs $O(N)$ time to perform the suffix sorting on the concatenated sequence T and another $O(N)$ to sort them again by position. It takes $O(mq^2Q)$ to compute the anchor score for each sliding window. There are at most $N = \sum_{h=0}^{m-1} |T_h|$ such potential anchor points to be computed. It takes $O(N)$ to plot the anchor score with position sliding window reference. It takes another $O(N)$ time to map the anchor sites back to the final consensus sequence for multiple sequence alignment. In total, the algorithm needs $O(mq^2QN + 4N) = O(mq^2QN)$ time to determine the required anchor points.

4.5 Results

The program takes the standard FASTA format protein data as input and performs the sort-based multiple sequence alignment. The following protein pairs are from putative *E. coli* which shares homology with the enteropathogenic E. coli (EPEC). A pair of such protein sequences are chosen to compare the results.

```
>gi|304362|gb|AAA23097.1| shares homology with the enteropathogenic
E. coli (EPEC) eae (E. coli attaching and effacing) gene; putative
akyrefkiihkqfnepqrstvwwaklllmwhssdhhhyiewepncdiscndscss
>gi|304363|gb|AAA23097.2| shares homology with the enteropathogenic
E. coli (EPEC) eae (E. coli attaching and effacing) gene; putative
wahvneymmvhwkrfeiyrtylmcwsklllletrhlkdhkgymykfhncsdiecnsccs
```

Table 4.2: Pairwise protein sequences of enteropathogenic E. Coli (EPEC) in FASTA format to be aligned.

```
.akyrefkiiH.KqFNEpqRSTv...wwak..lllmwhssdhHhyiewepncdiscnd.SCSS
wahvneymmvHWKrF.EiyR.TylmcwsklllletrhlkdhkgYmykfh.n.c.sdiecnSCCS
```

Table 4.3: Final alignment results by proposed algorithm on pairwise protein sequence alignment of enteropathogenic E. Coli (EPEC).

The current implementation can correctly display the optimal alignment layout. It's very fast in practical running time. The result above takes no more than 1 second. Compared with results of DiAlign [1, 2] and Clustal W [3, 4], our implementation can successfully align "H.K/HWK" block, "FNE/F.E" block and "RST/R.T" block. DiAlign is unable to align any of these 3 blocks. Clustal W can only align "H.K/HWK" block, but not "FNE/F.E" block, or "RST/R.T" block. These blocks are capitalized in Table 4.6 and Table 4.4. The mutation pre-processing was performed before the anchor-based vertical alignments. The BLOSUM62 mutation matrix was used in this alignment. The sort-based alignment was then applied after the mutation pre-processing. An important novelty in our approach is how biological mutation information is incorporated in

```
>DiAlign
-akyrefkiiHKqFNEpqRSTvw-waklllmw-hssdhhHyiewepnc-discnDSCSS
wahvneymmvHWKrFEiyRTylmcwsklllletrhlkdhkGymykfhncsdiecnSCCS-

>Clustal W
-akyrefkiiH-KqFNEpQRSTvwwaklllm-whssdhhHyiewepnc-discnDSCSS
wahvneymmvHWKrFEiyRTylmcwsklllletrhlkdhkGymykfhncsdiecnSCCS-
```

Table 4.4: Final alignment results by DiAlign [1, 2] and Clustal W [3, 4] on pairwise protein sequence alignment of enteropathogenic E. Coli (EPEC).

each refinement step. Thus, the recursive alignment refinement is performed via sorting and alignment between anchor points using a biological mutated version of the original sequences. Recall from that for BLOSUM62 mutation matrix, “H” has the closest biological mutation distance to “Y” than any other protein, see Figure 4.6. Using the biological mutation matrix, our proposed algorithm can successfully align “H/Y” for the protein sequences. However both DiAlign and Clustal W fail to align “H/Y”, marked by capitalized letters in Table 4.6 and Table 4.4. Towards the end of the input protein sequences, our algorithm can better align “SCSS/SCCS” block with only one mismatch. However, both DiAlign and Clustal W can align the block as “DSCSS/SCCS-”, which creates three mismatches. This sort-mutate-anchor paradigm separates our proposed sequence alignment algorithm from other previously proposed techniques.

The program works very well on pairwise sequence alignment. Meanwhile, it also gives a good alignment on multiple sequence alignments. The program constructs suffix arrays directly in linear time and space, without the need to first construct suffix trees. The program then performs alignment by first identifying a set of anchor points, based on the suffix sorting output on the sequences. Final alignment and gap considerations are made by a recursive application of the sort-based anchor point algorithm.

For multiple sequence alignment, the program takes the standard FASTA format protein data as input and performs the sort-based multiple sequence alignment. The following protein sequences are from putative *E. coli* which shares homology with the enteropathogenic *E. coli* (EPEC). A group of such protein are chosen to compare the

results. The practical running time is within 1 second.

```
>gi|304357|gb|AAA23099.1| shares homology with the enteropathogenic
E. coli (EPEC) eae (E. coli attaching and effacing) gene; putative
SGIKQMVYAHQKCM
>gi|304358|gb|AAA23093.1| shares homology with the enteropathogenic
E. coli (EPEC) eae (E. coli attaching and effacing) gene; putative
GEVRLWFNHIPPC
>gi|304359|gb|AAA23087.1| shares homology with the enteropathogenic
E. coli (EPEC) eae (E. coli attaching and effacing) gene; putative
IGCDDEFEENHDVTPMCY
```

Table 4.5: Multiple protein sequences of enteropathogenic E. Coli (EPEC) in FASTA format to be aligned.

```
SGIKQMVYA..HQK...CM
.GEVRLWFN..HIPP..C
IGCDDE.FEENHDVTPMCY
```

Table 4.6: Final alignment results by proposed algorithm on multiple protein sequence alignment of enteropathogenic E. Coli (EPEC).

4.6 Concluding Remarks

We presented a sort-based alignment algorithm for multiple sequences. Our proposed algorithm particularly works well on sequences with long range tandem duplications because of computational feasibility given linear time suffix arrays and reduced coincidence from long range sequences. It converts the multiple sequence alignment problem to a simple well studied linear suffix sorting problem and thus circumvents the computational overhead. Our proposed approach distinguishes itself by recursively identifying the anchor sites and fixing the master sequence structure. Using a two-passing sorting routine,

the algorithm stable-sorts the concatenated sequence by alphabet and in-sequence position. All the potential anchor points are selected by anchor points scoring criteria according to their distinct occurrence, differentiated weight and distance standard deviation. The highest scoring anchor sites are first selected and mapped back to the consensus sequence as the fixed anchor sites. We simplify the alignment score functions, namely C_1, C_2 and C_3 so as to better reflect the actual implication. We analyze the complexity requirements of the proposed algorithm. We are able to correctly scale the proposed algorithm from smaller-size benchmark sequence data to practical protein sequences. We verify the performance of the proposed alignment algorithm particularly on real protein benchmark sequence data. Meanwhile, we embed the mutation based alignment scheme into the current algorithm. The current algorithm treats each individual mismatch according to the biological mutation matrix information. We believe the mutation mapping will produce a more biologically-relevant alignment for the scientific environment.

Chapter 5

Repetitive Structures and Data Compression

5.1 Overview

5.1.1 Challenge of Protein Sequence Compression

In the following, we use the term "protein sequence" to refer to the entire collection of all the proteins in an organism as was done in [34]. This is different from the use of the terminology in other related work such as [17] to refer to the sequence of amino acids in one polypeptide.

Even with the degeneracy of the genetic code, it is well established that protein sequences are particularly difficult to compress. One reason is the problem of multiplicity and individuality of protein sequences [98]. Given the relation between sequence complexity and complexity of organisms, one can expect that for complex higher organisms, their protein sequence should be difficult to model, and hence compress. Another major difficulty lies in the apparent randomness of the symbols in a protein sequence. In Figure

[1] Part of the work reported in this chapter has appeared in the papers [96, 97].

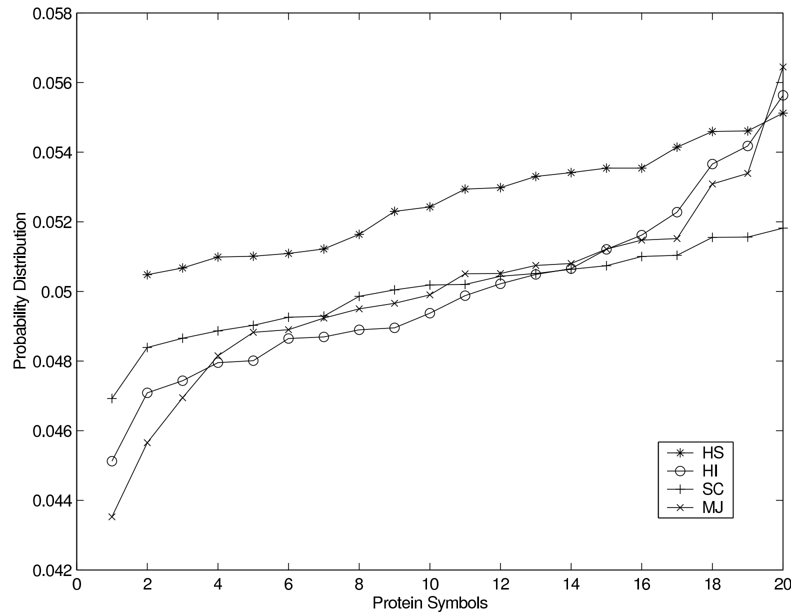

Figure 5.1: Sorted probabilities and higher order entropy. The protein sequences are taken from four organisms: HI: H. influenzae; MJ: M.jannaschii; HS: H. sapiens; SC: S. cerevisiae.

5.1, we show the individual probabilities for the four protein sequences in the Protein Corpus used by Nevill-Manning and Witten [34]. For each sequence, the symbols in x-axis are sorted in increasing order of their probability. The first observation is the very small range of values for the probabilities from around $0.043 - 0.057$ for any given sequence. The amino acid M. *Tryptophan* (Trp, symbol W) did not occur in the HS sequence. A smaller range for the probabilities imply more closeness to the uniform distribution, and hence less compressibility. It can be observed that HS (*H. Sapiens*) and SC (*S. Cerevisiae*) seem to have a higher complexity. This difficulty in compression is further illustrated in Figure 5.2, which shows the higher order entropy for the sequences.

We can also consider the problem of compressibility of protein sequences by looking at the performance of current state of the art compression algorithms on such sequences. Table 5.1 and Table 5.2 show that protein is in fact difficult to compress. Traditional compression algorithms (BWT, PPM, WinZip (LZ-based)) all expand, rather than compress the sequences. That is, they require more than 4.32 bits per symbol (bps) to represent

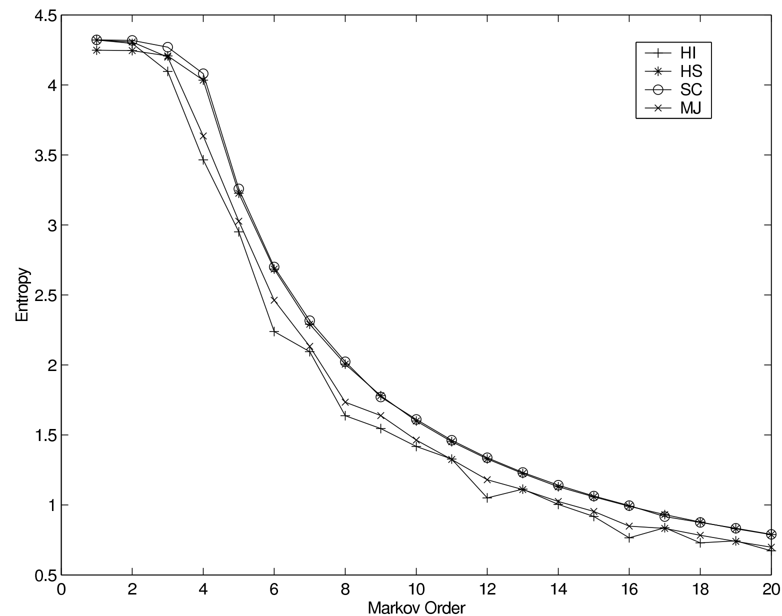

Figure 5.2: Sorted probabilities for the four sequences in the protein corpus. The protein sequences are taken from four organisms: HI: H. influenzae; MJ: M.jannaschii; HS: H. sapiens; SC: S. cerevisiae.

the protein sequences. A better result is produced by specific algorithms for protein sequence compression, but none was able to produce a universally good compression across all the sequences. Most of them still require more than 4bps. (The biological sequence compression algorithms in the table are GenCompress [60], CP [34], lzaCTW [62] and blockCode [67]).

5.1.2 Compressibility of Protein Sequences

Initial empirical evidence that protein sequences could indeed be significantly compressed can be seen in the partial success of the block coding algorithm of Sampath [67] and the context-tree weighting methods of Matsumoto et al [62]. See Table 5.1 and Table 5.2. The methods were able to compress some protein sequences in the corpus below the 4.0 bps mark. In particular, the block coding method was able to compress the HI sequence to about 3.66 bps, although the method performed very poorly on all the other sequences.

General Compression Algorithms					
	PPM	BWT	WinZip	Gzip	PPMD
H. Influenzae	4.881	4.49	4.671	4.672	4.151
M. Jannaschii	4.734	4.45	4.589	4.588	4.061
S. Cerevisiae	4.854	4.49	4.638	4.64	4.157
H. Sapiens	4.639	4.43	4.605	4.605	4.119

Table 5.1: Failure of general compression algorithms on protein sequences. Results in bits/symbol (smaller values imply better performance).

Biological Sequence Compression Algorithms							
	Gen Compress	CP(0)	CP(1)	CP(2)	CP(3)	Lza CTW-8	Block Code
H. Influenzae	4.156	4.156	4.149	4.146	4.143	4.118	3.665
M. Jannaschii	4.062	4.068	4.06	4.056	4.051	4.028	5.102
S. Cerevisiae	3.97	4.163	4.158	4.152	4.146	3.952	5.175
H. Sapiens	4.972	4.133	4.126	4.12	4.112	3.920	5.087

Table 5.2: Failure of biological sequence compression algorithms on protein sequences. Results in bits/symbol (smaller values imply better performance).

This is still a very significant result. It dispels the view that protein is incompressible, and shows that a carefully constructed algorithm could work well on a restricted set of input sequences. As was suggested by the author, this ability to compress only the sequences from HI could mean that H.Influenzae is a unique organism. We note that, from the viewpoint of compressibility, neither the higher-order entropy, nor the simple probability distribution of the symbols supports this view that HI is unique. Although generality of the algorithm is still a problem, especially when viewed from the definition of "incompressibility" in [34], the fact that such a relatively low compression can be achieved on a protein sequence is significant in its own right. This raises the hope that some unknown constraints may be at play in such sequences, which if identified could have implications not only for the compaction of the sequences, but on general analysis

of such sequences.

5.2 Long-Range Correlation in Protein Sequences

The coding regions in genomic sequences are generally less repetitive than the non-coding regions [17, 65]. Thus, it is widely agreed that protein sequences generally have more entropy, and hence exhibit more randomness than general DNA sequences. A repetition-based complexity profile has recently been proposed for prokaryotic sequences [99]. In fact, protein has been touted as being incompressible [34]. In studying the SCP (sorted common prefix) statistics of genomic sequences, we bumped unto an unusual observation: an unprecedented redundancy in protein sequences! (See Table 5.4, compare with Table 5.3). The protein sequences were the same as those used by Nevill-Manning and Witten in the "Protein is Incompressible" paper [34]. Each of the four files in the corpus contains a concatenated sequence of all the proteins in the genome of one single organism (except for that of H.Sapiens, which is not complete). The observation is the unusually high values of Kmax in the SCP statistics for the protein sequences. (Kmax is the maximum common prefix for a given sequence). This means that, certain set of genes in some area of the genome are repeated in exactly the same order at some other points, further down the genome. Apart from the mere size of the repeated subsequences as given by Kmax, an equally important aspect is the relatively long range of separation between the repeats. For H. Sapiens and S. Cerevisiae, the occurrences with $Kmax$ are separated by more than $350,000$ protein symbols (or greater than $1,050,000$ base pairs). Since the protein sequences are concatenated, this means that the repetition could involve genes located in different chromosomes. For protein sequences (see Table 5.4), we can observe the overlapping nature: $Kmax$ tends to be larger than the difference between the starting indices.

To our knowledge, this scale of redundancy has never been observed in protein sequences of a genome. Although multiple gene copies and repeated histone clusters are known to be present in most eukaryotic genomes [100], their number and their size are not enough to explain the above observation. More importantly, the orders in which the

genes are arranged in the genome tend to be conserved as they are being repeated at a different location along the genome. Apparently, this redundancy has not been previously observed with computational methods, because protein sequences are not usually considered in the way it was considered by Nevill-Manning and Witten [34]. The reason could come from the following.

1. Most computational analysis of protein sequences treat each gene independent of the other, and not in the concatenated form as was done in [34].

2. They are usually based on protein sequences from different species, and not the complete set of protein sequences from a large genome, such as the human genome.

3. They do not expect to find such long range repetitions in protein, and hence, choice of parameters may exclude such unusual cases.

4. Most repeat finding algorithms do not consider overlapping repeats.

5. Large complete genomes have only recently become available.

Thus, simple methods to find repeats would fail to identify such long-range redundancies that could span chromosomal boundaries.

This observation goes against the grain of conventional assumptions about protein sequences in the data compression community. The biological implications are numerous, for instance in the regulation and control of gene expressions. Protein sequences from the same genome could exhibit very long-range periodicities. But these are interrupted and masked by the introns - the non-coding areas in the genome. One could use this as a basis for a check for correct splice sites of subsequent genes, (after the correct identification of some initial genes). There could also be some evolutionary implications: nature not only conserves the important genes by replications in the same genome, but it also conserves their orders. Biological and evolutionary evidence of the prevalence of genome-wide gene duplications and their implications have been studied in [28, 101].

SCP Statistics for Common DNA Sequences						
Sequence	Size u	Kmax	$\frac{Kmax}{u}$	Start Index 1	Start Index 2	Difference
humEpsBarr	172,280	32,442	0.188	12,145	15,217	3,072
HUMGHCSA	66,495	408	0.00614	8,572	1,958	6,614
MitoMPOMTCG	186,609	191	0.00102	121,038	38,876	82,162
YSCCHRIII	316,613	1,682	0.00531	12,335	199,487	187,152
YSCCHRIV	1,531,929	3,573	0.00233	528,048	531,933	3,885
E. Coli	4,638,690	2,815	0.00061	4,165,800	4,207,196	41,396

Table 5.3: SCP statistical information for six common DNA sequences of humEpsBarr, HUMGHCSA, MitoMPOMTCG, YSCCHRIII, YSCCHRIV and E. Coli.

SCP Statistics for Concatenated Protein Sequences							
Sequence	Size u	Number of Genes	Kmax	$\frac{Kmax}{u}$	Start Index 1	Start Index 2	Difference
H. Influenzae	448,770	1,740	220,685	0.492	53,200	8	53,192
M. Jannaschii	509,508	1,680	343,105	0.673	34,899	3	34,896
S. Cerevisiae	2,900,346	8,220	886,531	0.306	480,296	29	480,267
H. Sapiens	3,295,749	5,733	392,004	0.119	358,676	24	358,652

Table 5.4: SCP statistical information for four concatenated DNA sequences of H. Influenzae, M. Jannaschii, S. Cerevisiae and H. Sapiens.

5.2.1 Efficient Analysis of Sequence Correlation

We use the SCP data structure described in [102] as the basic structure to analyze the protein sequences. Let T be an input sequence of length $u = |T|$. Suppose we precompute the longest-common-prefix (LCP) between all pairs of the sorted suffixes from a sequence, T. Using the relationship between the BWT [41] and the suffix tree, in addition to the auxiliary arrays previously described in [40, 103], we can obtain these sorted suffixes in linear time. We store this information in a table. Since the table is based on the sorted suffixes, we call this the sorted common prefix (SCP). Although the SCP and traditional LCP store the same basic information, the structure of the SCP is

completely different. Also, the sorted nature of the suffixes for the SCP has implications in the computation of this table, and its diverse uses. Given the i-th and j-th sorted suffixes, (SS_i and SS_j respectively), if $SCP(i,j) = k$, it means that the first k positions in the i-th suffix and the j-th suffix are exact matches (i.e., $SS_i[1 \dots k] = SS_j[1 \dots k]$ and $SS_i[1 \cdots (k+1)] \neq SS_j[1 \cdots (k+1)]$). We observe the following properties about the SCP structure. Let i, j, k, be indices for the sorted suffixes, such that $i < j$, and $j < k$. Then, $SCP(i,j) \geq SCP(i,k), \forall k > j$. Also, if $SCP(i,j) \geq 0$ and $SCP(j,k) = 0$, then $\forall k > j$, $SCP(j,k) = 0$. More generally, let $x = SCP(i,k)$. Then, $\forall k > j$, $SCP = (j,k) = SCP(i,k) + SCP(SS_j[(x+1) \dots u], SS_k[(x+1) \dots u]) = x + SCP(SS_j[(x+1) \dots u], SS_k[(x+1) \dots u])$. The SCP is symmetric: $SCP(j,k) = SCP(k,j)$. It is also usually sparse, although not always. Example, with $T = AAAAA$, we will have a full SCP table, where the row entries are of the form $1111; 222; 33; 4$. Essentially, any pair of suffixes with an SCP value different from 0 has some repetition which may or may not lead to a compression gain. Further, the block nature of the SCP means that we can compute the SCP for each symbol block, without reference to the other symbol blocks, leading to a lower complexity in the calculations.

Figure 5.3 shows the different forms of large scale repetitions (gene duplications) we observed in the protein sequences. $S1$ is the repeated pattern. Some repetitions are overlapped with previous repetitions (case A), or are completely contained in some previous repetition (case B), while others could occur thousands of amino acids down the protein sequence (case C). In some other cases, the repeated subsequences could span two different adjourning repetitions (case D). We observed that in all the sequences in the protein corpus, the large scale repetitions tend to be tandem duplications.

In [89], it was shown that, for a fixed alphabet, the SCP can be computed using an $O(n)$ number of comparisons on average, where n is the length of the sequence. We, however, note that the compression results are independent of how we extract the long range tandem duplications or other repetitions in the sequence. The important issue is that we extract and exploit them in the compression. But the compression time clearly depends on the method we use to extract these.

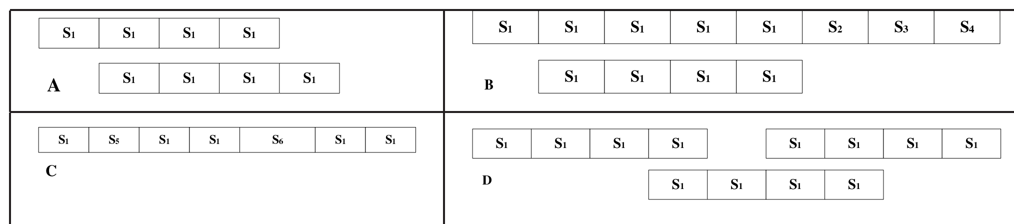

Figure 5.3: Different forms of repetitions observed in the protein sequences, as exposed by the SCP. A: Overlapping repeats (pattern S_1 occurred 5 times). B: One subsequence covers the other subsequence. We break them down into smaller sequences S_1. C: Repeated sequences are separated by possibly long stretches of amino acids between them. This is the most common case. D: The triple overlap case. We break the overlaps down into subsequence S_1.

5.3 Compressing Protein Sequences

Figure 5.4 is a flowchart of the proposed compression algorithm. After each parse, we obtain two outputs - the dictionary items and the remaining sequences. The dictionary consists of information about the repeated patterns, such as repetition length, position, number of occurrence, etc, needed for later decoding. Since both of these may contain redundancy, to achieve maximum compression gain, we pass these two to the core algorithm again until no compression could be achieved. For two repeating sequences that overlap (cases A, B and D in Figure 5.3), we cannot replace one of them without cutting the length of the second. In this scenario, we will construct a shorter subsequence to represent these two, as shown in Figure 5.3. After identifying the various forms of correlation in the sequence, the next problem becomes how to use these to compress the protein sequence. This will typically involve performing some form of substitution using the discovered repetition structures. Thus, we need to parse the input sequence to indicate the positions of the repeats and where and how they are to be substituted. We use an off-line dictionary based approach. First, we remove the repeats from the original sequence, and move them into an off-line dictionary (or index) of repeats. Then we code all occurrences of the repeat with reference to the position of the repeat in the dictionary. The size of the pointers is likely to be generally smaller than when we use on-line dictionary (especially with absolute references), since the number of repeats should be

much smaller than the size of the input sequence. But we have to compress and send the off-line dictionary as part of the compressed string. To guarantee compression, we can enforce a condition that whenever an item is inserted in the dictionary, it should not lead to an expansion of the data. We also have to consider whether referencing using the pointers should be relative or absolute.

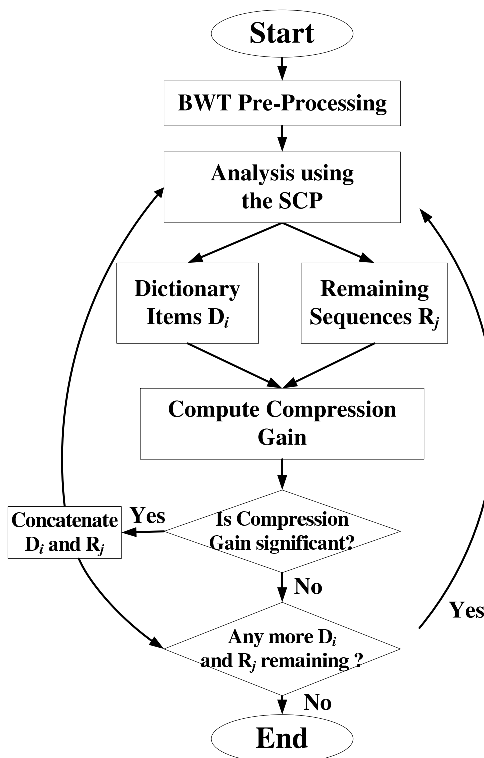

Figure 5.4: Flowchart for proposed compression algorithm. Dictionary and remaining sequences are recursively passed to the compression program until the overall compression gain is not significant.

5.3.1 Parsing and Encoding

Our approach here is to use the vocabulary parsing scheme (vps) proposed in [40]. For simplicity we use the $VPS1$ algorithm, which provides off-line dictionary compression with pointers in the dictionary. The compression performance will depend on a number

of factors, such as: the size of the pointers and the way they are coded; the type of referencing used - absolute or relative; the size of the dictionary (i.e., number of distinct repetition structures); and the type of referencing used in the dictionary (if any).

Under algorithm $VPS1$, we remove each repeated substring from the input sequence, and move it to an external dictionary. In the dictionary, we record the positions in the sequence where each repetition occurred, along with the repetition type. Thus there is no reference or pointer information in the original sequence. The parsed string is just a concatenation of the remaining subsequences after the repeats have been removed. To capture the case of different repetition types, we include repetition codes for each type of repetition in the dictionary.

Example Consider the following sample sequence, S:

	P_1		P_2		P_3		P_4		P_5		P_6		P_7
x_1	MMCTGTCMM	x_2	MM	x_3	GTCMM	x_4	TG	x_5	MMCTG	x_6	TTGMCMGTT	x_7	MM

When we move the repeated sequences into the dictionary, the remaining parsed sequence will be:

$$Parse(S) : x_1x_2x_3x_4x_5x_6x_7$$

The x_i's represent some other parts of the sequence that are not included in the repetition structure. Thus, the parsed sequence from Algorithm $VPS1$ is very simple. The dictionary however is a little more complicated. Using the following notations: $r_i = i$-th repetition pattern, $l(r) = |r|$ =length of repeat pattern r, (r) =total number of occurrences of r, $t(r)$ =repetition code for current occurrence of r, we can represent the dictionary structure for the sample sequence as follows:

The performance of the above scheme depends critically on the internal representations used for the dictionary. We use the term vocabulary to refer to the ensemble of repeat structures without reference to their specific locations in the sequence. For simplicity, we consider only the case with one type of repeat. The analysis can be extended to the general case with different types of repeats. We analyze the expected compression gain for the offline scheme using the above dictionary organization. We note that we do not need to explicitly encode $l(r)$ and $\eta(r)$ since these can be computed on the

Index	Repeat Pattern	$l(r)$	$t(r)$	$\eta(r)$	Positions
1	MMCTGTCMM	9	1	1	p_1
			3	1	p_6
2	GTCMM	5	1	1	p_3
			2	1	p_5
3	MM	2	1	2	p_2, p_7
4	TG	2	1	2	p_4

Table 5.5: Different types of tandem repeats. $t(r) = 1$ indicates direct repeat. $t(r) = 2$ indicates reverse repeat. $t(r) = 3$ indicates complemented palindrome.

fly. By using self-delimiting and uniquely decodable codes, we avoid direct coding of the delimiters in the above representations.

We use the following additional notations: $P_{r,j}$ = position of the j-th occurrence of repeat pattern r; κ = dictionary size (i.e., number of distinct repetitions); $\mu = |S|$ = size of the sequence, Σ = input alphabet, ($|\Sigma| = 20$ for protein sequences); $\beta = \lceil \log |\Sigma| \rceil$ = number of bits required to code each amino acid; $b(n) = \lceil \log n \rceil$ = number of bits required to represent an integer n; $C(X)$ cost of coding sequence X; $L = max\{l(r_i)\}$ = the maximum length of the repeats; 0 and 1 are flag bits.

Vocabulary Encoding: Vocabulary:$r_1 0 r_2 0 \ldots 0 r_\kappa 0$

Positions: $P_{1,1}, P_{1,2}, \ldots P_{1,\eta(1)} 0 P_{2,1}, \ldots P_{2,\eta(2)}, 0 \ldots 0 P_{\kappa,1}, P_{kappa,2}, \ldots P_{\kappa,\eta(\kappa)}, 0$

Using the above representation, we have the following: Cost of original sequence: $C(S) = |S|.\lceil \log |\Sigma| \rceil = \mu.\beta$ Cost of parsed sequence (i.e., remaining sequence after removing repeats): $C(parse(S)) = \mu\beta - \beta \sum_{i=1}^{\kappa} l(r_i)\eta(r_i)$

Cost of vocabulary: $C(V_A(S)) = \beta_1 \sum_{i=1}^{\kappa}(l(r_i) + 1) = \kappa\beta + \beta_1 \sum_{i=1}^{\kappa} l(r_i)$, where $\beta < \beta_1 \leq \beta + 1$. Essentially, the addition of the flag bit 0 forms a new alphabet with the original alphabet σ. Cost of positions: $= C(Pos_n(S)) = \sum_{i=1}^{\kappa} \sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil$ The cost of dictionary representation is then the combined cost of the vocabulary and the positions: $C(D(S)) = C(V_A(S)) + C(Pos_n(S))$

Compression Gain:

$$
\begin{aligned}
G(S) &= C(S) - (C(D(S)) + C(Parse(S))) \\
&= \mu\beta - ((\kappa\beta + \beta_1 \sum_{i=1}^{\kappa} l(r_i) + \sum_{i=1}^{\kappa} \sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil) + (\mu\beta - \beta \sum_{i=1}^{\kappa} l(r_i)\eta(r_i)))
\end{aligned}
$$

With $\beta_1 = \beta + 1$, we have an underestimation of the gain:

$$
G(S) = \beta \sum_{i=1}^{\kappa} l(r_i)(\eta(r_i) - 1) - \kappa(\beta + 2) - \sum_{i=1}^{\kappa} l(r_i) - \sum_{i=1}^{\kappa} \sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil
$$

Using the compression gain above, we can reduce the time required for parsing and computation of compression gain by performing the required computations based on the length of the repeat. We see that with $\kappa = 1$, the minimal length that can guarantee a positive compression gain will be given by:

$$
G(S) \geq \frac{\eta(r)\lceil \log \mu \rceil + \beta + 2}{\beta(\eta(r) - 1) - 1}
$$

Thus, for a protein sequence with 1 million symbols, say, using the simplest case of repeats that appear at least 2 times, we get a minimal length for the repeat sequence to be 14.

5.3.2 Multi-level Hierarchical Decomposition

To further exploit the potential redundancy, we perform the parsing and decomposition in a hierarchical manner. That is, after the first parse, the remaining sequence is used again as the input to the parsing algorithm to discover other potential correlations that may have been missed by the previous parsing stages. Similarly, given that the dictionary entries could be very long, potentially tens of thousands of amino acids, see Table 5.4. we also feed the dictionary entries for further decomposition and parsing. The result

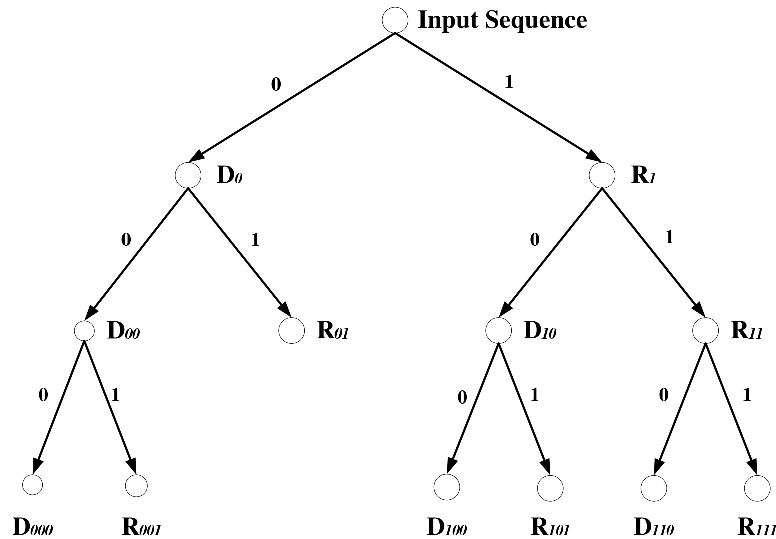

Figure 5.5: Multi-level hierarchical decomposition. Dictionary and remaining sequences are recursively passed to the compression program until the overall compression gain is not significant.

is a recursive multi-level decomposition strategy, see Figure 5.4. At each level, the input sequence is decomposed into two components, the dictionary D_i and the remaining sequence R_j. Both D_i and R_j are further passed to the parsing algorithm for further decomposition. Decomposition stops when the compression gain $G(S)$ for the input sequence is negative or less than a threshold. Each internal node except the root in the tree is a provisional output. Each leaf in the tree is part of the final output. For instance, R_{01} in Figure 5.4 is terminated as a leaf in the tree because we cannot achieve any more compression gain by decomposing it further. Thus, it is coded without further parsing. It is obvious that the remaining sequences R typically require more rounds of passing than the dictionary items D, since the remaining sequences normally have longer lengths.

The compression results typically improve with higher levels of decomposition. Also, the time required for parsing and compression gain analysis, decreases very fast as the levels of decomposition increase, although the overall time increases with more decomposition levels.

5.3.3 Results

Table 5.6 and Table 5.7 show the statistics of maximal repeats for the protein sequence and compression results using the above parsing scheme, and the observed long-range correlations in the sequences. Table 5.6 and 5.7 show a comparison with the results from Table 5.1 and Table 5.2 for the biological sequence compression algorithms on the same sequence. We have ignored the bits required to represent the special position symbols used to indicate gene region separators in the protein sequences (usually less than 100 bytes per sequence).

Sequence	Size	Kmax	Repeat Length $l(r)$	Number of Occurrence $\eta(r)$	Compression Results (bps)
H. Influenzae	509,508	220,685	34,896	7	2.546
M. Jannaschii	448,770	343,105	53,192	5	2.273
S. Cerevisiae	2,900,346	886,531	406,239	3	3.111
H. Sapiens	3,295,749	392,004	338,359	3	3.435

Table 5.6: Statistics of maximal repeats and compression results using the observed long-range correlations. Compression results in bits/symbol (small values imply better performance).

Sequence	Gen Compress	CP(0)	CP(1)	CP(2)	CP(3)	LZa CTW-8	Block Code	Proposed Method
H. Influenzae	4.156	4.156	4.149	4.146	4.143	4.118	3.665	2.546
M. Jannaschii	4.062	4.068	4.06	4.056	4.051	4.028	5.102	2.273
S. Cerevisiae	3.97	4.163	4.158	4.152	4.146	3.952	5.175	3.111
H. Sapiens	4.972	4.133	4.126	4.12	4.112	3.920	5.087	3.435

Table 5.7: Comparative compression performance using the observed long-range correlations. Compression results in bits/symbol (small values imply better performance).

The superior performance of the proposed method is evident in the above tables. We notice that the improved performance is consistent across all the different protein sequences. This is expected, given the consistently high values of K_{max} in Table 5.4.

It was observed that the long range correlations that produced the highest compression gains are often due to large tandem duplications, with thousands of symbols in the primitive pattern.

In terms of compression time, after the SCP is computed, the parsing and compression stage take about 7 minutes for MJ sequence, on a dual-Athlon 1.8 GHz processor running Ubuntu Linux 5.10.

5.4 Concluding Remarks

We have proposed a direct suffix sorting based protein sequence compression algorithm. We tested the method on the protein corpus often to evaluate protein sequence compression algorithms. A comparison between our proposed algorithm with some other existing data compression algorithms was made. Our main method is based on an analysis of the long-range repetition structures in the protein sequences. Most of these were observed to be tandem duplications which are then exploited for compression using an off-line dictionary strategy.

Chapter 6

Discussion

6.1 Summary

We have proposed two algorithms for solving the direct suffix sorting problem. The first algorithm runs in linear time and linear space on average, but in $O(n \log n)$ time in the worst case, using $O(n)$ space. Using ideas from Shannon-Fano-Elias codes used in information theory, the second algorithm improved the first to an $O(n)$ worst case time and space complexity. We extended the approach to the generalized $1 : \eta$ partitioning schemes. The algorithms proposed perform direct suffix sorting on the input sequence, circumventing the need to construct the suffix tree first.

The proposed algorithms are generally independent of the type of alphabet, Σ. The only requirement is that Σ be fixed during the run of the algorithm. Any given fixed alphabet can be mapped to a corresponding integer alphabet. Also, since practically the number of unique symbols in $|T|$ cannot be more than n, the size of T, it is safe to say that $n \geq |\Sigma|$.

For practical implementation, one will need to consider the problem of practical code assignment, since the procedure described may involve dealing with very small fractions, depending on m, the block length. This is a standard problem in practical data compression. Periodic re-scaling and re-normalization schemes can be used to address the

problem. Moffat et al [104] provide some ideas on how to address such practical problems.

The proposed direct suffix sorting is then applied to two different applications. One is protein sequence compression and the other is multiple sequence alignment. The thesis has considered the problem of compressibility of protein sequences based on the observation of long-range genome scale correlations in protein sequences, and proposed a simple scheme to exploit such correlations. Using a method based on the sorted common prefix, the algorithm identifies the correlated protein sequences, and then uses a simple dictionary-based parsing and encoding scheme to provide compression. The results show that the approach can provide a consistent compression of the protein sequences, at times down to less than 2.3 bps. The improved performance was observed on all the protein sequences in the Protein Corpus.

A sort based algorithm for sequence alignment is then presented. It's based on the observation that a well matched alignment is a re-arrangement of both highly repetitive regions and non-repetitive regions. The method fixes those repetitive regions through direct suffix sorting. A biological mutation mapping procedure is introduced to further exploit the structural and functional relatedness between the peptides. A seamless mutation based suffix sorting routine accomplished the processing of those non-repetitive regions. The algorithm bypasses the typical gap insertion difficulties by implicitly placing gaps after each anchor point determination. It conveniently converts the multiple sequence alignment problem to smaller problems involving the well-studied linear suffix sorting problem and thus circumvents the computational overhead. The proposed approach distinguishes itself by recursively identifying the anchor sites and fixing the master sequence structure. Using a two-step sorting routine, the algorithm stable-sorts the concatenated sequence lexicographically and in-sequence position. The potential anchor points are selected by the number of their distinct occurrence, differentiated weight and distance standard deviation. It's expected that the algorithm will improve the practical running time for large scale multiple sequence alignment problems.

6.2 Contributions

In this dissertation, we propose a linear time direct suffix sorting algorithm to construct the suffix array data structures. It does not employ the intermediate suffix tree data structures. We address an open problem in suffix sorting [46] by reducing the gap between the actual space requirement of current suffix sorting algorithms, and the minimal requirement of $5n$ bytes - for storing both the original string, and its suffix array. It is a divide-and-conquer sort-and-merge algorithm for performing direct suffix sorting on a given input string. Given a string of length n, our algorithm runs in $O(n)$ worst case time and space. The algorithm recursively divides an input sequence into two parts, performs suffix sorting on the first part, then sorts the second part based on the sorted suffix from the first.

Our algorithm differs from previous approaches in the use of a simple partitioning step, and how it exploits this simple partitioning scheme for conflict resolution. The space requirement for the proposed algorithm is $7n$ bytes. The method is also unique in its use of Shannon-Fano-Elias codes in efficient construction of a global partial order for the suffixes. To our knowledge, this is the first time information-theoretic methods have been used as the basis for solving the suffix sorting problem.

The direct suffix sorting algorithm is then applied to solve the multiple sequence alignment using a biological mutation matrix. Gaps are determined by the repetitive regions and consequently inserted back to the input sequences to attain an improved alignment. The protein symbols are mutated according to the biological mutation matrix to reflect their biological distance between each other with pre-determined mutation score as determined by the mutation matrix. We also apply the direct suffix sorting algorithm to the problem of compressibility of protein sequences. We propose a method to exploit the unusual redundancy in concatenated protein sequences in compressing such sequences. The result is a significant reduction in the number of bits required for representing the sequences. We report results in bits per symbol (bps) of 2.27, 2.55, 3.11 and 3.44 for the protein sequences from *M. Jannaschii*, *H. Influenzae*, *S. Cerevisiae*, and *H. Sapiens* respectively, the same protein sequences used by Nevill-Manning and Witten in the "Protein is incompressible" paper [34]. The observed long-range correlations could

have significant implications beyond sequence compression and complexity analysis.

Appendix A

List of Publications

1. Donald Adjeroh and Fei Nan, “Direct Suffix-Sorting via Shannon-Fano-Elias Codes”, Theoretical Computer Science, 2008 (submitted).

2. Fei Nan, “Bioinformatic Study of γ-Secretase and its Substrates”, Master Thesis, West Virginia University, 2008.

3. Donald Adjeroh and Fei Nan, “Suffix Sorting via Shannon-Fano-Elias Codes”, 2008 IEEE Data Compression Conference (DCC 2008), Snowbird, UT.

4. Fei Nan and Donald Adjeroh, “A Sort-based Algorithm for Multiple Sequence Alignment”, 2007 IEEE Computational Systems Bioinformatics (CSB 2007), San Diego, CA.

5. Fei Nan and Donald Adjeroh, “An Algorithm for Suffix Sorting and Its Applications”, 2006 IEEE Computational Systems Bioinformatics (CSB 2006), Stanford, CA.

6. Donald Adjeroh and Fei Nan, “On Compressibility of Protein Sequences” 2006 IEEE Data Compression Conference (DCC 2006), Snowbird, UT.

7. Fei Nan and Donald Adjeroh, “On Complexity Measures for Biological Sequences” 2004 IEEE Computational Systems Bioinformatics (CSB 2004), Stanford, CA.

References

[1] B. Morgenstern, A. Dress, and T. Werner, "Multiple DNA and protein sequence alignment based on segment-to-segment comparison," *Proceedings of the National Academy of Sciences*, vol. 93, pp. 12 098–12 103, 1996.

[2] B. Morgenstern, "DIALIGN 2: improvement of the segment-tosegment approach to multiple sequence alignment," *Bioinformatics*, vol. 15, pp. 211–218, 1999.

[3] D. G. Higgins and P. M. Sharp, "CLUSTAL: A package for performing multiple sequence alignment on a microcomputer," *Gene*, vol. 73, no. 1, pp. 237–244, 1988.

[4] J. Thompson, D. Higgins, and T. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, pp. 4673–4680, 1994.

[5] B. Morgenstern, S. J. Prohaska, D. Pöhler, and P. F. Stadler, "Multiple sequence alignment with user-defined anchor points," *Algorithms for Molecular Biology*, vol. 15, pp. 1–6, 2006.

[6] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A model for evolutionary change in proteins. atlas of protein sequence and structure," *National Biochemical Research Foundation*, vol. 5, pp. 345–352, 1978.

[7] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," in *Proceedings of the National Academy of Sciences*, vol. 89, 1992, pp. 10 915–10 919.

[8] D. Haussler, "Computational genefinding," *Trends in Biochemical Sciences, Supplementary Guide to Bioinformatics*, pp. 12–15, 1998.

[9] N. Volfovsky, B. J. Hass, and S. Salzberg, "A clustering method for repeat analysis in DNA sequences," *Genome Biology*, vol. 2, no. 8, pp. 1–11, 2001.

[10] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Research*, vol. 27, pp. 2369–2376, 1999.

[11] J. Aach, M. Bulyk, G. Church, J. Comander, A. Derti, and J. Shendure, "Computational comparison of the two draft sequences of the human genome," *Nature*, vol. 409, no. 1, pp. 856–859, 2001.

[12] S. F. Altschul, T. L. Madden, A. A. Schffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[13] W. R. Pearson, "Comparison of methods for searching protein sequence databases," *Protein Science.*, vol. 4, pp. 1145–1160, 1995.

[14] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge, UK: Cambridge University Press, 1997.

[15] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge, UK: Cambridge University Press, 1998.

[16] D. M. Mount, *Bioinformatics: Sequence and Genome Analysis, 2nd ed.* Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, 2004.

[17] L. Gatlin, *Information Theory and the Living System.* New York, NY: Columbia University Press, 1972.

[18] R. N. Mantegna, S. V. Buldyrev, A. L. Goldberger, C. K. Peng, M. Simons, and H. E. Stanley, "Linguistic features of noncoding DNA sequences," *Physical Review Letters*, vol. 73, no. 23, pp. 3169–3172, 1994.

[19] R. I. Richards, K. Holman, S. Yu, and G. R. Sutherland, "Fragile X syndrome unstable element, p(CCG)n, and other simple tandem repeat sequences are binding sites for specific nuclear proteins," *Human Molecular Genetics*, vol. 2, pp. 1429–1435, 1993.

[20] B. S. Majewski, N. C. Wormald, G. H. Havas, and Z. J. Czech, "A family of perfect hashing methods," *Computer Journal*, vol. 39, no. 6, 1996.

[21] O. Bat, M. Kimmel, and D. E. Axelrod, "Computer simulation of expansions of DNA triplet repeats in the Fragile X Syndrome and Huntington's disease," *Journal of Theoretical Biology*, vol. 188, pp. 53–67, 1997.

[22] R. R. Sinden, V. N. Potaman, E. A. Oussatcheva, C. E. Pearson, Y. Lyumchenko, and L. S. Shlyakhtenko, "Triplet repeat DNA structures and human genetic disease: dynamic mutations from dynamic DNA," *Journal of Biosciences*, vol. 24, no. 1, pp. 53–65, 2002.

[23] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[24] P. Clote and R. Backofen, *Computational Molecular Biology*. Wiley, 2002.

[25] Z. Gu, L. Steinmetz, X. Gu, C. Scharle, R. Dacis, and W. Li, "Role of duplicate genes in genetic robustness against null mutations," *Nature*, vol. 421, pp. 63–66, 2003.

[26] Z. Gu, A. Cavalcanti, F. Chen, P. Bouman, and W. Li, "Extent of gene deplication in the genomes of drosophila, nematode, and yeast," *Molecular Biology and Evolution*, vol. 19, no. 3, pp. 256–262, 2002.

[27] S. Cutler and P. McCourt, "Dude, where's my phenotype dealing with redundancy in signaling networks," *Vision Statement of Plant Physiology*, vol. 138, pp. 558–559, 2005.

[28] T. J. Gibson and J. Spring, "Genetic redundancy in vertebrates: polyploidy and persistence of genes encoding multidomain proteins," *Trends in Genetics*, vol. 14, pp. 46–49, 1998.

[29] J. Normanly and B. Barteltl, "Redundancy as a way of life - IAA metabolism," *Current Opinion in Plant Biology*, vol. 2, pp. 207–213, 1999.

[30] M. A. Nowak, M. C. Boerlijst, J. Cooke, and J. M. Smith, "Evolution of genetic redundancy," *Nature*, vol. 388, pp. 167–171, 1997.

[31] D. Maier, B. M. Marte, W. Schafer, Y. Yu, and A. Preiss, "Drosophila evolution challenges postulated redundancy in the E(spl) gene complex," *Proceedings of the National Academy of Sciences*, vol. 90, pp. 5464–5468, 1993.

[32] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmca*, vol. 40, no. 1, pp. 33–50, 2004.

[33] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *Journal of the Association for Computing Machinery*, 2006.

[34] C. G. Nevill-Manning and I. H. Witten, "Protein is incompressible," in *IEEE Data Compression Conference*, 1999, pp. 257–269.

[35] P. Weiner, "Linear pattern matching algorithms," in *Bulletin of Mathematical Biology*, vol. 46, 1984, pp. 567–577.

[36] E. M. McCreight, "A space-economical suffix tree construction algorithm," in *Journal of the Association for Computing Machinery*, vol. 23, 1976, pp. 262–272.

[37] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249–260, 1995.

[38] P. Fenwick, "The Burrows-Wheeler transform for block sorting text compression," *Computer Journal*, vol. 39, no. 9, pp. 731–740, 1996.

[39] J. Seward, "On the performance of BWT sorting algorithms," in *IEEE Digital Compression Conference*, vol. 17, 2000, pp. 173–182.

[40] D. Adjeroh, Y. Zhang, A. Mukherjee, M. Powell, and T. C. Bell, "DNA sequence compression using the Burrows-Wheeler Transform," in *IEEE Computational Systems Bioinformatics Conference*, Palo Alto, CA, August 2002, pp. 303–313.

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

[42] D. Adjeroh, T. Bell, and A. Mukherjee, *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*, 1st ed. Springer, 2008.

[43] N. J. Larsson and K. Sadakane, "Faster suffix sorting," Lund University, Tech. Rep., 1999.

[44] S. J. Puglisi, W. F. Smyth, and A. Turpin, "The performance of linear time suffix sorting algorithms," in *Digital Compression Conference*, Snowbird, UT, pp. 358–367.

[45] M. Farach-Colton, "Optimal suffix tree construction with large alphabets," in *IEEE Symposium on Foundations of Computer Science*, Miami Beach, FL, October 1997, pp. 137–143.

[46] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, "On the sorting-complexity of suffix tree construction," *Journal of the Association for Computing Machinery*, vol. 47, pp. 987–1011, 2000.

[47] D. K. Kim, J. S. Sim, H. Park, and K. Park, "Constructing suffix arrays in linear time," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 126–142, 2005.

[48] P. Ko and A. Aluru, "Space-efficient linear time construction of suffix arrays," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 143–156, 2005.

[49] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays," in *String Processing and Information Retrieval Symposium and International Workshop on Groupware*, Cancun, Mexico, September 1999, pp. 81–88.

[50] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 444–453, 1970.

[51] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[52] M. Waterman and M. Perlwitz, "Line geometries for sequence comparison," *Journal of Computational Biology*, vol. 1, no. 4, 1994.

[53] D. F. Feng and R. F. Doolittle, "Progressive sequence alignment as a prerequisite to correct phylogenetic trees," *Journal of Molecular Evolution*, vol. 25, no. 4, pp. 351–360, 1987.

[54] C. Notredame, D. Higgins, and J. Heringa, "T-Coffee: a novel algorithm for multiple sequence alignment," *Journal of Molecular Biology*, vol. 302, pp. 205–217, 2000.

[55] R. Edgar, "MUSCLE: Multiple sequence alignment with high score accuracy and high throughput," *Nucleic Acids Research*, vol. 32, pp. 1792–1797, 2004.

[56] S. R. Eddy, "Multiple alignment using hidden markov models," *ISMB intelligent systems for molecular biology*, vol. 3, pp. 114–120, 1995.

[57] M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1993.

[58] S. Grumbach and F. Tahi, "A new challenge for compression algorithms: genetic sequences," *Information Processing and Management*, vol. 30, pp. 875–886, 1994.

[59] E. Rival, O. Delgrange, J. P. Delahaye, M. Dauchet, M. Delorme, A. Henaut, and E. Ollivier, "Detection of significant patterns by compression algorithms: the case

of approximate tandem repeats in DNA sequences," *Computer Applications in the Biosciences*, vol. 13, pp. 131–136, 1997.

[60] X. Chen, S. Kwong, and M. Li, "A compression algorithm for DNA sequences and its applications in genome comparison," in *Proceedings of the 10th Workshop on Genome Informatics (GIW'99)*, 1999, pp. 52–61.

[61] J. K. Lanctot, M. Li, and E. Yang, "Estimating DNA sequence entropy," in *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2000)*, San Francisco, CA, January 2000, pp. 409–418.

[62] T. Matsumoto, K. Sadakane, and H. Imai, "Biological sequence compression algorithms," Department of Information Science, University of Tokyo," Research report, 2000.

[63] G. Korodi and I. Tabus, "An efficient normalized maximum likelihood algorithm for DNA sequence compression," *ACM Transactions on Information Systems*, vol. 23, no. 1, pp. 3–34, 2005.

[64] A. Apostolico and S. Lonardi, "Offline compression by greedy textual substitution," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1733–1744, 2000.

[65] D. Loewenstern and P. Yainilos, "Significantly lower entropy estimates for natural DNA sequences," in *Proceedings of IEEE Data Compression Conference*, Snowbird, UT, March 1997, pp. 151–161.

[66] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communication*, vol. 32, no. 4, pp. 396–402, 1984.

[67] G. Sampath, "A block coding method that leads to significantly lower entropy values for the proteins and coding sections of Haemophilus Influenzae," in *Proceedings of IEEE Bioinformatics Conference*, Palo Alto, CA, August 2003, pp. 287–293.

[68] F. Nan and D. Adjeroh, "An algorithm for suffix sorting and its applications," in *IEEE Computational Systems Bioinformatics*, Palo Alto, CA, August 2006.

[69] D. Adjeroh and F. Nan, "Suffix sorting via Shannon-Fano-Elias codes," in *IEEE Data Compression Conference*, 2008, pp. 502–502.

[70] ——, "Direct suffix sorting via Shannon-Fano-Elias codes," *Theoretical Computer Science*, 2008.

[71] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath, "Order-preserving minimal perfect hash functions and information retrieval," *ACM Transactions on Information Systems*, vol. 9, pp. 281–308, 1991.

[72] A. K. Garg and C. C. Gotlieb, "Order-preserving key transforms," *ACM Transactions on Database Systems*, vol. 11, pp. 213–234, 1986.

[73] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley Interscience, 1991.

[74] F. Nan and D. Adjeroh, "A sort-based algorithm for multiple sequence alignment," in *IEEE Computational Systems Bioinformatics*, San Diego, CA, August 2007.

[75] F. Nan, "Bioinformatic study of γ-secretase and its substrates," Master's thesis, West Virginia University, 2008.

[76] I. Dubchak and L. Pactcher, "The computational challenges of applying comparative-based computational methods to whole genomes," *Briefings in Bioinformatics*, vol. 3, pp. 18–22, 2002.

[77] M. Kellis, N. Patterson, M. Endrizzi, B. Birren, and E. Lander, "Sequencing and comparison of yeast species to identify genes and regulatory elements," *Nature*, vol. 423, pp. 241–254, 2003.

[78] R. Mutsumi, H. Atomi, and T. Imanaka, "Identification of the amino acid residures essential for proteolytic activity in an archeal signal peptide peptidase," *Journal of Biological Chemistry*, vol. 281, no. 15, pp. 10 533–10 539, 2006.

[79] B. DasGupta and L. Wang, *Selected Topics in Computational Biology.* John Wiley & Sons, Inc, 1998.

[80] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of Computational Biology*, vol. 1, no. 4, 1994.

[81] M. Sammeth, B. Morgenstern, and J. Stoye, "Divide-and-conquer alignment with segment-based constraints," *Bioinformatics*, vol. 19, pp. 89–95, 2003.

[82] I. V. Walle, I. Lasters, and L. Wyns, "Align-m: A new algorithm for multiple alignment of highly divergent sequences," *Bioinformatics*, vol. 20, no. 9, pp. 1428–1435, 2004.

[83] J. H. Choi, H. G. Cho, and S. Kim, "GAME: A simple and efficient whole genome alignment method using maximal exact match filtering," *Computational Biology and Chemistry*, vol. 29, pp. 244–253, 2005.

[84] J. D. Thompson, F. Plewniak, and O. Poch, "BAliBASE: A benchmark alignment database for the evaluation of multiple sequence alignment programs," *Bioinformatics*, vol. 15, pp. 87–88, 1999.

[85] J. D. Thompson, P. Koehl, R. Ripp, and O. Poch, "BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark," *Proteins: Structure, Function, and Bioinformatics*, vol. 61, pp. 127–136, 2005.

[86] D. A. Pollard, C. M. Bergman, J. Stoye, S. E. Celniker, and M. B. Eisen, "Benchmarking tools for the alignment of functional noncoding DNA," *Bioinformatics*, vol. 5, 2004.

[87] M. Brudno, C. Do, G. Cooper, and M. Kim, "LAGAN and multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA," *Genome Research*, vol. 13, pp. 721–731, 2003.

[88] W. Huang, D. M. Umbach, and L. Li, "Accurate anchoring alignment of divergent sequences," *Bioinformatics*, vol. 22, pp. 29–34, 2006.

[89] D. Adjeroh and J. Feng, "Locating all tandem repeat families in a sequence," in *IEEE Computational Systems Bioinformatics Conference*, Palo Alto, CA, August 2003, pp. 676–681.

[90] G. Myers, S. Selznick, Z. Zhang, and W. Miller, “Progressive multiple alignment with constraints,” *Journal of Computational Biology*, vol. 3, 1996.

[91] D. G. Brown and A. K. Hudek, “New algorithms for multiple DNA sequence alignment,” in *Workshop on Algorithms in Bioinformatics*, Bergen, Norway, September 2004, pp. 314–325.

[92] B. Ma, J. Tromp, and M. Li, “Patternhunter: Faster and more sensitive homology search,” *Bioinformatics*, vol. 18, no. 3, 2002.

[93] U. Keich, M. Li, B. Ma, and J. Tromp, “On spaced seeds for similarity search,” *Discrete Applied Mathematics*, vol. 138, pp. 253–263, 2004.

[94] B. Brejova, D. Brown, and T. Vinar, “Optimal spaced seeds for homologous coding regions,” in *Journal of Bioinformatics and Computational Biology*, 2004, pp. 595–610.

[95] X. Zhang and T. Kahveci, “A new approach for alignment of multiple proteins,” in *Pacific Symposium on Biocomputing*, vol. 11, Maui, HI, January 2006, pp. 339–350.

[96] F. Nan and D. Adjeroh, “On complexity measures for biological sequences,” in *IEEE Computational Systems Bioinformatics*, Palo, Alto, CA, August 2004, pp. 522–526.

[97] D. Adjeroh and F. Nan, “On compressibility of protein sequences,” in *IEEE Data Compression Conference*, Snowbird, UT, March 2006, pp. 422–434.

[98] J. S. Fruton, *Protein, Enzymes, Genes: The Interplay of Chemistry and Biology.* Ithaca, NY: Yale University Press, 1999.

[99] O. G. Troyanskaya, O. Arbell, Y. Koren, G. M. Landau, and A. Bolshoy, “Sequence complexity profiles of prokaryotic genomic sequences: A fast algorithm for calculating linguistic complexity,” *Bioinformatics*, vol. 18, no. 5, pp. 679–688, 2002.

[100] J. D. Hawkins, *Gene Structure and Expression.* Cambridge, UK: Cambridge University Press, 1985.

[101] M. Hurles, "Gene duplication: The genomic trade in spare parts," *PLoS Biology*, vol. 205, no. 2, pp. S13–S19.

[102] D. Adjeroh and J. Feng, "The SCP and compressed domain analysis of biological sequences," in *IEEE Computational Systems Bioinformatics Conference*, Palo Alto, CA, August 2004, pp. 587–592.

[103] T. Bell, M. Powell, A. Mukherjee, and D. A. Adjeroh, "Searching BWT compressed text with the Boyer-Moore algorithm and binary search," in *IEEE Data Compression Conference*, Snowbird, UT, March 2002, pp. 112–121.

[104] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, pp. 256–294, 1995.