Development of Alignment-free Algorithms for Various Post-genomic Data
多様なポストゲノムデータのためのアラインメントフリーなアルゴリズムの構築

by

Taku Onodera
小野寺拓

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on June 12, 2015
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Tetsuo Shibuya　渋谷哲朗
Associate Professor of Computer Science

ABSTRACT

DNA, RNA and proteins are basic biopolymers that are used universally among almost all biological systems. Inferring the functions of these molecules from their sequential or structural information is one of the most important problems in computational biology. As the so-called next generation sequencing (NGS) technologies and molecular dynamics simulation technologies are developed, it has become realistic to comprehensively investigate the space of sequences/structures that have never been observed. To advance such studies further, we need to develop efficient data analysis methods that are not dependent on alignment and more sophisticated (*de novo*) genome assemblers that can handle large and complex genomes such as metagenomes or eukaryotic genomes. We study these problems in this thesis.

In the study of alignment-free data analysis methods, we investigate the possibilities of alignment-free annotation methods for sequences and structures. Classification by composition-based string kernels and support vector machines (SVMs) is an existing alignment-free method for function prediction. Particularly, the spectrum kernel achieves relatively high classification accuracy and highly efficient computation based on the suffix tree. Thus, we start from this method and design better methods by applying more advanced data structures developed in string algorithms community.

In Chapter 3, we propose the b-suffix array data structure to make it possible to compute the kernel function that we introduce in Chapter 4 in time independent of the dimension of the feature space. This data structure is a generalization of the suffix array and it can also be applied as a string index that supports the search of patterns with wildcards in predetermined positions. Such pattern matching problems arise in spaced seed-based sequence homology search. We also propose non-trivial construction algorithms for the b-suffix array.

In Chapter 4, we propose the gapped spectrum kernel, a string kernel that is based on the frequency of substrings as well as the spectrum kernel. Different to the spectrum kernel, we introduce gaps (wildcards in pattern matching) to the substrings. A similar idea is used in an existing method called wildcard kernel. The wildcard kernel achieves high prediction accuracy by considering all gap patterns of a given pattern length and weights, but it was not known what happens when multiple but not all gap patterns are used. Applying the results from Chapter 3, we propose an efficient algorithm to calculate the gapped spectrum kernel and an algorithm to make a SVM prediction in time independent of the size of the support vectors. We also show that the sum of the gapped spectrum kernels corresponding to a given length and weights matches the wildcard kernel. From this relationship, efficient methods for wildcard kernel computation and prediction are derived. We also experimentally show that the sum of a few gapped spectrum kernels corresponding to randomly chosen parameters can predict protein families comparatively accurately as the wildcard kernel.

In Chapter 5, we study protein structure analysis. Protein structures are more directly related to functions than sequences are and thus, if available, they can be important clues to infer functions. On the other hand, structural alignment, the *de facto* standard method to measure structural similarities, is more computationally expensive than sequence alignment. Thus, in Chapter 5, we give an alignment-free kernel for protein structures applying the techniques for alignment-free kernels for sequences we saw in Chapter 4. Also, we propose an efficient method for kernel computation, which is based on an existing data structure called the two-dimensional suffix tree, and prediction method that takes time independent of the size of support vectors. We experimentally show that, compared to the most accurate similar existing method, the proposed method can achieve comparative accuracy while it runs more than 300 times faster.

We consider genome assembly problem in Chapter 6. Most existing genome assemblers first construct a graph that represents the overlaps of reads and try to recover the original sequence or long substring of it by following a path of the graph. While graph construction has been studied extensively, there is no established method for the part of recovering the original sequence from the graph. Particularly, one big open problem is how to process substructures introduced into the assembly graph by sequencing errors, repeat regions, diploidy/polyploidy or possibly other reasons. Existing methods detect such substructures by using simple motifs. Sometimes, however, complex substructures that cannot be detected by simple motifs considered in previous work do appear. What is required ultimately is to determine how to process these substructures but, different to simple motifs, detecting such complex substructures is already non-trivial. We, therefore, give a graph theoretic characterization of these complex substructures, which we name superbubbles, and clarify several properties of them. We also propose an efficient algorithm to detect all superbubbles in a given graph. The algorithm takes time quadratic to the number of vertices in the worst-case, but it runs very efficiently in practice. We show the algorithm runs in linear time in expectation under a probabilistic model.

As a whole, in this thesis, we develop alignment-free algorithms to facilitate comprehensive studies of biological sequences and structures.

論文要旨

DNA, RNA, タンパク質といった高分子は全ての生物において基本的な役割を果たしている. これらの機能や性質をその配列や構造の情報から予測することは計算生物学の最も重要な問題である. 次世代シーケンサー（NGS）や分子動力学シミュレーション技術の発展にともない今までは観測されてこなかった配列・構造空間の網羅的な探索が可能となってきている. そのような研究をさらに推し進めるには, アラインメントに基づかない, より効率的なデータ解析手法の設計および, メタゲノムや真核生物のゲノムなど巨大で複雑なゲノムにも対応できるより高度な (*de novo*) ゲノムアセンブリ手法が求められている. 本論文では網羅的な配列・構造データのアラインメントフリーな解析を実現するために, これらの問題に取り組む.

アラインメントに基づかないデータ解析手法の研究では, アラインメントフリーな配列・構造データのアノテーション手法の可能性を探る. 機能予測におけるアラインメントフリーな既存手法として部分文字列などの頻度に基づく文字列カーネルと support vector machine (SVM) を使った分類手法がある. とくに spectrum kernel という既存手法では比較的高い予測精度に加え, 接尾辞木を用いた高速な計算が可能である. そこで, この手法をベースに, 文字列アルゴリズムの分野で発展してきたより高度なデータ構造も援用しつつ, さらに発展的な手法の開発を行う.

まず３章では, ４章で提案するカーネル関数を特徴空間の次元によらない計算量で求めるために, 接尾辞配列の一般化である b 接尾辞配列というデータ構造を提案する. このデータ構造は接尾辞配列の一般化であり, 特定の位置にワイルドカードを含むパターンの検索にも応用可能である. このようなパターンマッチングは実際に spaced seed を用いた配列相同性検索において必要とされている. ３章ではこの他に b 接尾辞配列の非自明な構築アルゴリズムも提案する.

４章では spectrum kernel と同様に部分文字列の頻度に基づく gapped spectrum kernel という文字列カーネルを提案する. Spectrum kernel との違いは部分文字列の中にギャップ（パターンマッチにおけるワイルドカード）を導入することである. 類似のアイディアは wildcard kernel という既存手法でも使われている. Wildcard kernel は特定のサイズと重みのギャップパターンを全て同時に考慮することで spectrum kernel よりも高い予測精度を実現しているが, 必ずしも全てではない複数のギャップパターンを考慮した場合にどのようなことが起きるのかは知られていなかった. ３章の結果を応用し, gapped spectrum kernel を求める効率的なアルゴリズムを提案する他, SVM を使った予測に必要な計算をサポートベクターのサイズによらない計算量で行う手法も提案する. また, 特定のサイズと重みに対応した gapped spectrum kernel の和が wildcard kernel に等しいことを示す. この関係により, wildcard kernel の計算と SVM のための効率的なアルゴリズムが得られる. また, 計算機実験により, 少数のランダムに選んだ gapped spectrum kernel の和によって, wildcard kernel を用いるのと遜色のない精度でタンパク質の family を予測できることを示す.

５章ではタンパク質立体構造に対するアノテーションに取り組む. タンパク質の構造は配列よりも直接的に機能に関係しており, 既知であれば機能推定のための重要な手がかりになる. 一方立体構造の類似度の指標として標準的に使われている構造アラインメントとよばれる手法は配列アラインメントよりもさらに多くの計算量を必要とする. そこで５章では４章において考えた部分文字列の頻度に基づくアラインメントフリーな文字列カーネルのテクニックを立体構造に応用し, 構造に対するアラインメントフリーなカーネルを与える. また, 二次元接尾辞木とよばれる既存のデータ構造を用いた効率的なカーネルの計算法と, サポートベクターのサイズによらない予測法を提案する. タンパク質 superfamily の予測実験では, 提案手法が類似の既存手法で最も精度のよいものと同程度の予測精度を, 300 倍以上高速に達成できることを示す.

ゲノムアセンブリに関しては６章で扱う. 既存のゲノムアセンブリ手法ではまずリードのオーバーラップを表すグラフを作り, そのグラフのパスをたどることで元の配列, またはその中に含まれる長い部分文字列を復元するというアプローチが主流である. グラフの構築に関してはすでに多くの研究が行われているが, グラフから元の配列を復元する部分に関してはまだ確立された方法は存在しない. とくにシーケンスエラー, リピート配列, 多倍体ゲノムなどによってグラフの中に導入される部分構造をどのように処理するかが一つの大きな問題となっている. 既存手法ではグラフに含まれるシンプルなモチーフを使ってこのような部分構造を検出していた. しかし場合によっては既存手法で考えられていたようなシンプルなモチーフでは検出できない複雑な部分構造が出現することがある. 最終的にはこれらの部分構造をどう処理するかが問題になるが, それ以前にこれらの部分構造はよりシンプルなモチーフと異なり検出自体が非自明である. そこで我々はこのような複雑な部分構造のグラフ理論的な特徴づけとして superbubble という部分グラフのクラスを定義し, その性質を明らかにする. また, 与えられたグラフに含まれる全ての superbubble を検出するための効率的な手法を提案する. 提案手法はグラフの頂点数に対して最悪で二乗の計算量を必要とするが, 実際にはひじょうに効率的に動作する. 我々は確率的なモデルのもと, 提案手法が平均的に線形時間で動作することを示す.

全体として, 本論文では網羅的な生物学的配列・構造研究にむけたアラインメントフリーアルゴリズムを開発する.

Acknowledgements

I express my deepest gratitude to my academic advisor Prof. Tetsuo Shibuya. Throughout my doctoral study, I got all sorts of lessons too numerous to mention here from him but probably, the most critical one is the importance of enjoying the work. Working with him was an immense joy.

I am also grateful to my coauthors Alex Bowe and Prof. Kunihiko Sadakane. Prof. Sadakane also kindly gave me an opportunity to give a talk at NII Shonan meeting in September 2013.

In 2013, I had a chance to work as an research assistant at JST, ERATO, Kawarabayashi Large Graph Project. I thank the project leader Prof. Ken-ichi Kawarabayashi and our group leader Dr. Kohei Hayashi for letting me join the project. The contents of Chapter 6 of this thesis were derived from this project.

I was very lucky to have many influential researchers around me. Prof. Hiroshi Imai, kindly allowed me to participate in his laboratory seminars at Hongo campus of our university. By giving talks at the seminar, I could get lots of helpful feedback and lessons. Occasionally, I visited the Laboratory of DNA Information Analysis in our Shirokanedai campus and had some exciting discussion with members there on various topics from research to social problems to the charisma of their boss Prof. Satoru Miyano. In particular, exchange with Dr. Atsushi Niida and Dr. Paul Sheridan stimulated my interests in biology very much. I appreciate Prof. Miyano for creating such a community.

I extend my gratitude to the stuffs of institutions I had a chance to work in, including Kimiko Otani, Akiko Shimada, and Keiko Yamaguchi from the Computer Science Department, Graduate School of Science, the University of Tokyo, Asako Suzuki and Ayako Tomiyasu from the Laboratory of DNA Information Analysis, Satoko Tsushima from Global Research Center for Big Data Mathematics, National Institute of Informatics. Dr. Ayumu Saito from Laboratory of DNA Information Analysis kindly helped me whenever I faced hardware troubles though it was not his job.

I also express my gratitude to all the people with whom I had a chance to have helpful discussions on the contents of this thesis, including Dr. Travis Gagie, Kentaro Honda, Prof. Seiya Imoto, Prof. Jesper Jansson, Daisuke Kimura, Prof. Gregory Kucherov, Prof. Shin-ichi Minato, Dr. Yuichi Shiraishi and Dr. Rui Yamaguchi.

Lastly, I thank my family for everything.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

Nucleic acids (DNA and RNA) and *proteins* are responsible for almost all biological functions. Understanding these molecules is the central problem of molecular biology and has a practical impact on medicine.

The *sequential* or *structural* aspects of these molecules are intimately related to their meanings and functions. For example, DNA sequences[1] contain blueprints of proteins and programs to control protein production processes and the structures of enzyme proteins specifically determine which enzyme acts on which substrate. While it is usually difficult and time-consuming to determine the function of molecules directly by experiments, there are various technologies to obtain the sequence or structure data of biological molecules. Thus, it is important to automatically annotate these sequences and structures.

What underlies most sequence analysis methods is the assumption that sequential similarities imply functional similarities. This assumption is not always but often true. The standard method to measure sequence similarities is sequence alignment such as global alignment [97] or local alignment [125]. Alignment is a type of generalization of the edit distance and intuitively, it represents 'alignment' of sequences such as the one in Figure 1.1. Given a newly found sequence, one way to predict its function is database search, i.e., to find sequences in the database that are similar to the new sequence. Since exhaustive comparison by alignment is too slow, heuristic database search methods such as FASTA [78] and BLAST [3] were developed. A more accurate annotation method is to extract highly conserved patterns, called motifs, from sequences that are already associated to some function. Upon a newly derived sequence is given, the prediction is made by checking if the motif matches to the new sequence. The types of motifs include consensus sequences [120, 121], position specific scoring matrices [127] and hidden Markov models [63]. Even more accurate annotation is possible by deriving discrimination rules using both positive data (associated to the function) and negative data (not associated to the function). The combination of *support vector machines(SVMs)* [133] and *string kernels* is a major class of methods in this category.

Protein structures are more directly related to the functions than sequences are. Also, there are proteins that are not similar at sequential level but share

[1] In biology and related fields, the term ' sequence ' is used to mean string. We use these terms exchangeably in this thesis.

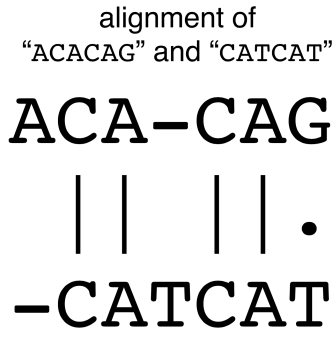


Figure 1.1: An example of the sequence alignment. `ACACAG` can be transformed to `CATCAT` by deleting the first `A`, inserting `T` just before the second `C` and modifying the last character from `G` to `T`.

similar structures and functions.[2] Therefore, if the structure of a protein with unknown function is obtained, structure analysis is an important tool to infer its function. Alignment is also considered for protein structures, where correspondence between atoms (usually α-carbons) is considered [129, 40, 118, 146]. However, different from sequence alignment, there is no *de facto* standard method for the alignment of protein structures. A commonality is the high computational cost compared to sequence alignment.

1.2 Alignment-free Methods for Comprehensive Studies of Biological Molecules

The recent advent of the so-called next generation sequencing (NGS) technologies made it possible to obtain massive amount of sequence data very quickly at low cost. Also, because of the progress in the methods for protein structure identification, especially molecular dynamics (MD) simulation, unprecedented amount of protein structure data are being produced these days. These technologies enabled us to collect large-scale data that are less biased. Previously, due to the high cost of genome sequencing, sequence analyses, especially whole genome analyses, were focused on few model organisms. Similarly, currently known protein structures are biased towards those proteins that are amenable to experimental structure identification methods such as X-ray crystallography or nuclear magnetic resonance. Advancements in NGS and MD simulation are opening the possibility of more comprehensive studies of the molecular biology. However, in order to realize such comprehensive studies, several problems must be addressed.

First, data analysis methods geared to large-scale and less biased data are needed. Most existing analysis methods are based on alignment. While alignment is suitable for the close examination of few sequences, it has several drawbacks. For one thing, calculating the optimal alignment is often computationally expensive. It takes $O(n^2)$-time to calculate the edit distance (the most simple case of alignment) between length n sequences by dynamic programming [137, 97, 114, 115, 138, 139] and there is an evidence that this time complexity is (modulo logarithmic factors) optimal [6]. Also, aligning objects with

[2] It is also possible that structurally similar proteins do not share functions.

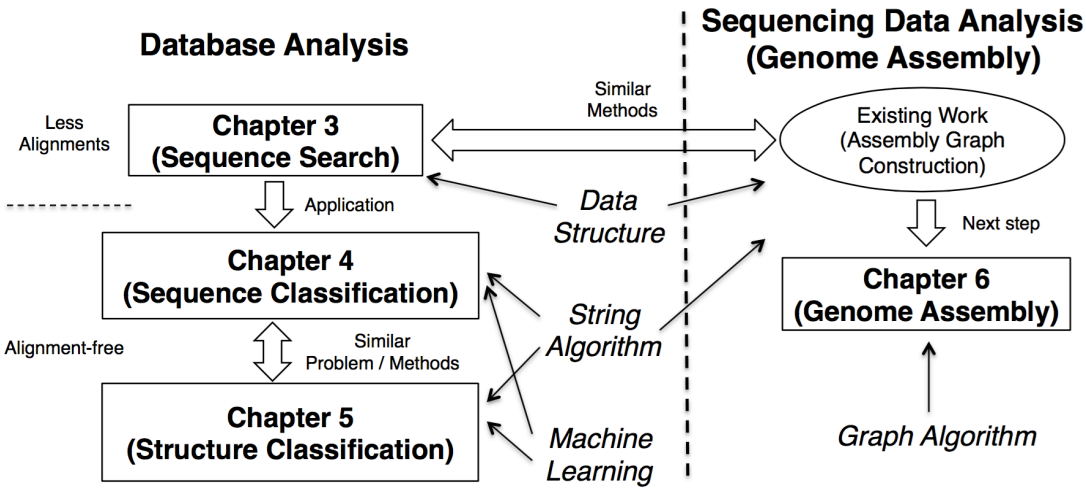


Figure 1.2: The relationship between problems and techniques related to this thesis.

low similarities often results into meaningless outputs.

Second, *de novo* genome assemblers that can handle reads from comprehensive studies such as metagenome analysis are needed. Although NGS made it possible to collect massive amount of DNA sequences at a very low cost, existing DNA sequencing technologies cannot read a DNA sequence without cutting it into short fragments called reads. To take the full advantage of NGS, we need a method to reconstruct the whole genome sequence from reads. This problem is called *genome assembly*. When a reference genome, the whole genome sequence of an organism from the same or close species as the sequenced individual, is available, one usually assembles reads by mapping them to the reference genome. When there is no reference genome, one needs to recover the whole genome only from reads, which is called *de novo* genome assembly. Although mapping is much easier than *de novo* genome assembly, it is not applicable for comprehensive studies such as metagenome analysis (analysis of genomes from multiple organisms in various environments such as soils, seawaters or human gut.) because reads are derived from a mixture of possibly unknown species. Even for the study of intra species-level phenomena of some organisms of particular interest, in order to perform whole genome analysis, one needs to do *de novo* assembly at least once. Because we consider only *de novo* assembly in this thesis, we omit the term '*de novo*' for brevity. Although genome assembly have been studied extensively in bioinformatics community, currently, it is still difficult to assemble complex and large-scale genomes such as eukaryotic genomes or metagenomes.

In this thesis, we study alignment-free methods for functional annotation of biological sequences/structures and genome assembly problem to pave the road towards more effective and comprehensive studies of biological molecules. The high level overview of our work is summarized in Figure 1.2.

Function prediction. Most existing annotation methods are based on alignment and thus, inherit the aforementioned drawbacks of alignment. To solve the root of this problem, it is necessary to design sequence similarity measures that are free from alignment.

Previously, alignment-free sequence analysis has been studied in the context

of phylogenetics, the study of evolutionary histories. In phylogenetic problems where evolutionary distant sequences need to be compared, sequence alignment often results into meaningless outputs. Thus, string similarity measures based on substring composition, average common substring length, information theory and iterated maps have been developed [135, 136]. However, there were almost no similar methods for function annotation.

A notable exception is the string kernel proposed by Leslie et al. [73], called the spectrum kernel, and related string kernels [126, 71, 72]. In general, string kernels are similarity measures for strings that can be plugged into SVMs. The spectrum kernel and the related kernels are based on the composition of substructures such as substrings of some length and in particular, they are free from alignment. One thing that makes these sequence similarity measures particularly interesting is that these measures can sometimes be calculated very efficiently by applying the string index data structures. For example, the spectrum kernel [73] for two strings of length n can be calculated in optimal $O(n)$-time by applying the suffix tree [142, 25] data structure. It is in stark contrast to the time complexity of $O(n^2)$ for sequence alignment.

Because SVMs accept any similarity measures as long as it satisfies a property called positive semi-definiteness, there can be many possible ways to further improve and extend these composition-based string kernels. Also, mainly since 1990s, in string algorithm community, many advanced suffix tree-like data structures and related algorithms have been developed [142, 51, 83, 16, 25, 26, 36, 55, 61, 99]. Therefore, it is very promising to, starting from the spectrum kernel, design more advanced string kernels applying the advanced index data structures.

Genome assembly. The state-of-the-art genome assemblers today are based on some graphs encoding the overlap information of reads. Each edge of such a graph is labeled by a string and, if constructed successfully, the original sequence (or at least a long substring of it) corresponds to a tour. In the past several years, many studies have been made on the step to construct assembly graphs and this part is now well established [13, 131]. On the other hand, how to recover the original whole genome sequence from the assembly graph remains open. In particular, it is known that if there exist many similar (but not the same) reads, which are possibly introduced by sequencing errors, repeat regions and diploidy/polyploidy, then, genome assembly usually becomes very difficult. Another factor that complicates assembly is that the graph usually contains spurious edges introduced by sequencing errors. We study this problem through graph theoretic approach.

1.3 Overview of Our Contributions

In this section, we briefly explain each of the four main contributions of this thesis.

1.3.1 Text Indexing with Gaps.

We introduce a new string kernel the gapped spectrum kernel in Chapter 4. Although this kernel and the combination of multiple instances of it can be used

to predict protein families more accurately than the spectrum kernel [73], the dimension of feature vector depends exponentially on a parameter. On the other hand, the spectrum kernel can be computed in time linear to the input size (the size of strings) by applying the suffix tree [126]. While the spectrum kernel is based on the frequencies of substrings of some particular length, the gapped spectrum kernel is based on the frequencies of the substrings of some particular length with gaps in particular positions. Therefore, it is natural to apply data structures that are similar to the suffix tree but supports search of gapped patterns to the computation of the gapped spectrum kernel. In pattern matching terminology, gaps here are called wildcards. In literature, several suffix tree-based data structures that support the search of patterns with wildcards have been studied. However, these data structures either takes super-linear space or search time involves terms that depends exponentially to the parameter. They are also very complex. We, instead, propose the *b-suffix array*, a very simple variant of the suffix array that supports the search of patterns with wildcards in predetermined positions. Although our main motivation for this data structure is the application to the computation of the gapped spectrum kernel, we treat it independently in Chapter 3 because it may have other applications. In particular, aside from the application to Chapter 4, the b-suffix array can also be used as a text index that supports the search of patterns with wildcards in predetermined positions. Such a pattern matching problem occurs, e.g., in the context of spaced seed search in sequence homology search. The technical contributions in Chapter 3 are construction algorithms for the b-suffix array. The b-suffix array can be constructed naïvely by radix sort. At each bucket sort in the radix sort, suffixes are sorted by single character. We show how to modify radix sort so that multiple characters can be taken into account at each bucket sort.

The results in Chapter 3 were originally published in [102].

1.3.2 The Gapped Spectrum Kernel for Support Vector Machines

In the context of comprehensive studies of 'protein universe', computationally efficient protein sequence annotation is needed. The spectrum kernel [73] is a particularly promising approach to this problem. A string kernel is a similarity measure for strings that can be plugged into support vector machines (SVMs). The resulting SVMs can learn the discrimination rules from annotated sequences and can be applied to automatically annotate newly found sequences. The spectrum kernel is known to be able to achieve computational efficiency and prediction accuracy both at high levels.

Though the spectrum kernel characterizes strings by the composition of contiguous patterns included in the string, a simple calculation indicates that using gapped patterns instead of contiguous patterns leads to better prediction. Indeed the wildcard kernel [71] achieves higher prediction accuracy than the spectrum kernel by taking into account all possible gap patterns of particular length and weights. However, it was not known whether it is necessary to consider *all* gap patterns for that purpose. In Chapter 4, we investigate it.

We first introduce the gapped spectrum kernel and an efficient algorithm to compute it. The gapped spectrum kernel is based on the composition of non-contiguous patterns of particular gap positions. The kernel computation

algorithm is based on the b-suffix array data structure introduced in Chapter 3 and runs in time independent of the dimension of the feature space. Also, we show how to perform the prediction phase of SVMs based on this kernel function in time that is independent of the size of the support vectors. Then, we show that the sum of the gapped spectrum kernels for all gap patterns of a particular length and weights matches the wildcard kernel. Thus, we can derive an interpolation between single gapped spectrum kernel and the wildcard kernel. Together with the aforementioned prediction algorithm, this gives a prediction algorithm for the wildcard kernel with time complexity independent of the size of the support vectors (the existing algorithm depends linearly). Also, we experimentally show that the sum of the gapped spectrum kernels corresponding to a few randomly selected gap patterns can predict protein families comparatively accurately as the wildcard kernel.

The results in Chapter 4 were originally published in [103].

1.3.3 Fast Classification of Protein Structures by an Alignment-free Kernel

Though protein structure data are more difficult to obtain than sequences are, analyzing structures is important because structures are more directly related to the protein functions. In order to extract useful information from structure data, especially those derived by molecular dynamics simulation studies, not only accurate but also computationally efficient structure annotation methods are needed. Structural alignment is the *de facto* standard method for protein structure comparison. However, structural alignment is computationally very expensive. The time complexity depends on the formulation but even the best known bound based on heuristics is still $O(n^4)$ where n is the size of the protein [108]. Therefore, the merit of alignment-freeness should be even bigger for protein structures than it is for sequences. In spite of that, there was no research focusing on the alignment-free analysis of protein structures.

In Chapter 5, we propose an alignment-free kernel for protein structures that is based on a novel use of *protein contact maps*, a notion introduced in the context of structural alignment. One can plug the proposed kernel into SVMs and apply it to classification. The proposed kernel can be computed in quadratic time by using the two dimensional suffix tree [32, 59]. Furthermore, the prediction based on the proposed kernel can be made in time independent of the size of the support vectors (existing methods depends linearly). Also, we experimentally demonstrate that the proposed classification method is comparatively accurate as the most accurate existing method, which is based on structural alignment, while it runs more than 300 times faster. This result indicates that alignment-free methods are promising for the analysis of protein structures in general.

The results in Chapter 5 are not published yet.

1.3.4 Detecting Superbubbles in Assembly Graphs

The standard approach to genome assembly today is graph-based [96, 107]. In this approach, one first constructs a graph encoding the overlap information of reads and then reconstructs the original sequence by 'linearizing' the graph. While researchers have been studying the graph construction phase vigorously,

relatively few studies have been made on the second phase of reconstructing the original sequence from the graph. Also, most existing work on the second phase is *ad hoc*. In Chapter 6, we broaden the realm of systematic studies a step further towards the second phase.

The difficult part of the reconstruction phase is the linearization of complex substructures of the graph, which presumably originate from repeat regions of the original sequence, sequencing errors or diploidy/polyploidy. Though identifying the real origins of these substructures is important, just finding such substructures is already highly non-trivial. On the other hand, once found, it seems possible to apply elaborate analysis algorithms such as multiple alignment to these complex substructures even if such elaborate algorithms are not applicable to the entire graph. Therefore, we define *superbubbles*, a graph theoretic characterization of the complex substructures, presumably introduced by repeats, errors or diploidy/polyploidy, and clarify their properties. We also give an efficient algorithm to enumerate all superbubbles in a given assembly graph. Though this algorithm takes $\Theta(n^2)$-time in the worst case where n is the number of vertices, it runs very fast in practice. We explain this behavior of the proposed algorithm by using a probabilistic model.

The results in Chapter 6 were originally published in [101].

1.4 Organization of the Thesis

The rest of this paper is organized as follows. We introduce the preliminary knowledge in the next chapter. In Chapter 3, we describe the b-suffix array data structure and related algorithms. In Chapter 4, we explain the new string kernel for protein sequences. In Chapter 5, we account for the alignment-free kernel for protein structures. Then in Chapter 6, we describe our study on genome assembly. We conclude in the last chapter.

Chapter 2

Preliminaries

2.1 Basic Assumptions

In this section, we introduce the model of computation and an assumption on strings that we use throughout this thesis. We do not need to be aware of these topics in most parts of this thesis, but we rely on them, explicitly or not, in the linear time construction of the suffix array, the linear time construction of the Isuffix tree and the constant time computation of range minimum query after linear time preprocessing in Section 2.3.

Machine model. Throughout this thesis, we use the *word RAM model* of computation. In this model, the computer has an infinite size random access memory consisting of words. Each word consists of a fixed number of bits. When we write, e.g., $O(n)$-space or the size of something is $O(n)$, we mean $O(n)$ words, not bits. Let w be the number of bits of each word. The computer can perform algebraic operations such as $<, +, -, \times, /, \mathsf{mod}$ on w bits integers all in constant time. Also, it is assumed that $w = \Theta(\log n)$ where n is the size of the input.[1] In particular, this assumption means that the address of a word used in any polynomial time computation fits in a constant number of words. This is a reasonable model of modern computers and is widely accepted in the algorithms community.

Integer alphabet. In this thesis, we assume that each character of a string or an entry of a matrix is encoded as an integer that fits in a constant number of words and that both encoding and decoding can be done in constant time. For example, the characters `A`, `T`, `G`, and `C` of DNA sequences (cf. Section 2.5) can be encoded as integers 0, 1, 2 and 3 respectively on computer memories. The set of code words in this assumption is called *integer alphabet*. We call this assumption the *integer alphabet assumption*. Integer alphabet assumption is a reasonable way to treat character encodings used in modern computers such as ASCII code in a general fashion and thus, it is widely used by string algorithms community. The strings we consider in this thesis, namely, DNA sequences and protein sequences have constant size alphabet (cf. Section 2.5)

[1] This assumption may sound weird because it means that the machine will change according to the size of the input. Another way to think about it is there are different machines for different sizes of the input.

and thus, integer alphabet assumption may look overly strong.[2] There are, however, two reasons it is preferable in this thesis too. First, we sometimes introduce additional characters to strings for technical reasons. For example, string algorithm researchers often treat a collection of strings as a single string by concatenating all member strings. When doing this, they delimit different strings by additional unique characters. In such a case, the alphabet size increases as the database grows. Second, problems for strings on integer alphabet often have as efficient solutions as the corresponding problems for strings on constant size alphabet. If this is the case, an integer alphabet is clearly preferable to constant alphabet because it is more general.

2.2 Notations

In this section, we introduce notations that are used in the rest of this chapter and the following chapters.

We denote the alphabet by Σ and the size of Σ by σ. The symbol $\circ$ represents the concatenation of strings and characters. For example, when S_1 and S_2 are strings, $S_1 \circ S_2$ is the string derived by appending S_2 to S_1. Similarly, when S is a string and c is a character, $S \circ c$ is the string derived by appending c to S. We denote the set of length k strings on Σ by Σ^k and the set of finite length strings on Σ by Σ^*.

For a string T, we denote the i-th character by $T[i]$ and the substring from the i-th to the j-th character inclusive by $T[i:j]$. We denote the length of T by $|T|$. For a pair of strings S and T, $lcp(S,T)$ denotes the *longest common prefix* of S and T. For a pair of strings T of length n and P of length m with $m \leq n$, an *occurrence* of P in T is an integer $i \in [1, n-m+1]$ such that $T[i:i+m-1] = P$. For an $m \times n$ matrix M and a $p \times q$ matrix P with $p \leq m$ and $q \leq n$, an *occurrence* of P in M is a pair of integers $(i,j) \in [1, m-p+1] \times [1, n-q+1]$ such that $M[i:i+p-1, j:j+q-1] = P$.

We denote the submatrix from the i_1-th row to the i_2-th row and the j_1-th column to the j_2-th column of a matrix M as $M[i_1:i_2, j_1:j_2]$. We also write $M[i, j_1:j_2]$ to mean $M[i:i, j_1:j_2]$ and write $M[i_1:i_2, j]$ to mean $M[i_1:i_2, j:j]$.

We use bold face symbols such as $\mathbf{x}$ to denote vectors. The expression $\mathbf{x}^\top$ means the transpose of vector $\mathbf{x}$.

For a finite set S with a total preorder $\preceq$, the *leftmost rank* of an element $x \in S$ is defined to be $\#\{y \in S : y \preceq x, x \npreceq y\}$. For brevity, we sometimes use the word rank to mean leftmost rank.

2.3 The Suffix Tree and the Suffix Array

The *suffix tree* [142] and the *suffix array* [83] are data structures that are very useful in problems related to strings, especially the search problem. These data structures are directly related to the contents of Chapter 3 and Chapter 4. It has also an indirect relationship with the contents of Chapter 6. We also need a variant of the suffix tree called the Isuffix tree in Chapter 5. Thus, in this section, we introduce these data structures.

[2] In Chapter 5, we consider binary matrices.

2.3.1 The Suffix Tree

For exposition, we first introduce the trie and the compressed trie.

Trie. The *trie* [28] data structure is a tree that stores a collection $\mathcal{C}$ of strings. Each edge is labeled by a character. For any u with children v and w, the label of edge (u, v) is not equal to the label of edge (u, w). For each string S in $\mathcal{C}$, there is a vertex v such that the concatenation of the edge labels of the edges in the path from the root to v is S. There are several ways to store the children of each vertex. We choose to store them in a sorted list (sorted by the label). Each vertex has at most σ children. Thus, at each vertex u and character c, the children v of u s.t. (u, v) is labeled by c can be found, if any, in $O(\log \sigma)$-time by binary search.

Given a trie, and a query string P of length m, one can find if P is in $\mathcal{C}$ by Algorithm 2.1. Because the while loop is executed at most m times, this algorithm takes $O(m \log \sigma)$-time.

Algorithm 2.1 Search algorithm for trie

Ensure: If $P \in \mathcal{C}$ then return true; otherwise return false

```
1:  u ← root of the trie
2:  i ← 1
3:  while true do
4:      if i > |P| then
5:          return true
6:      else if u has a child v s.t. (u, v) is labeled by P[i] then
7:          u ← v
8:          i ← i + 1
9:      else
10:         return null
```

Compressed trie. The *compressed trie* [93][3] is another data structure representing a collection of strings $\mathcal{C}$. It is defined to be the tree derived from the trie for $\mathcal{C}$ by contracting each path without any branching node to a single edge labeled by the concatenation of the labels of the original edges. The compressed trie has the following properties:

1. The first characters of the labels of the edges between an internal node and its children are different;

2. Each internal node has more than 1 children;

At each internal node, we store the children in a list sorted by the first characters of their labels.

We define some notation and terminology for the compact trie. We denote the label of edge e, which is a string, by $label(e)$. For brevity we sometimes write $label(u, v)$ to mean $label((u, v))$. For a node v, let $child(v, c)$ denote the child u of v such that the first character of $label(u, v)$ is c if such a child exists. A

[3]The compressed trie is called PATRICIA in the original paper.

locus is a concept that represents a node of the trie underlying the compressed trie. Formally, it is a triplet (v, c, i) where v is a node, c is a character and i is a non-negative integer. For a locus to be valid, either $i = 0$ or $child(v, c)$ exists and i is smaller than the length of $label(v, child(v, c))$. In the following, we only consider valid locuses. The *path label* of a locus $\ell = (v, c, i)$, which we denote by $str(\ell)$, is the concatenation of the labels of the edges in the path from the root to v and length i prefix of $label(v, child(v, c))$. We use locus $(v, c, 0)$ and node v exchangeably and sometimes write, say, $str(v)$ to mean $str((v, c, 0))$.

Let $\mathcal{C}$ be the set of strings stored in the compressed trie. We assume that any string in $\mathcal{C}$ is not a prefix of another string in $\mathcal{C}$. Note that, in this case, for any string $s \in \mathcal{C}$, there is a leaf ℓ such that $str(\ell) = s$. One can find all leaves corresponding to strings in $\mathcal{C}$ prefixed by a query P of length m by Algorithm 2.2. In line 5 v can be found in $O(\log \sigma)$-time by binary search. The while loop is executed for at most m times. In the string comparison in line 7, if $P[i]$ is matched to some character in S, then $P[i]$ is never used in the later comparisons. Thus, the contribution of line 7 to the total runtime is $O(m)$. Thus, the while loop takes $O(m \log \sigma)$-time. Because every internal node has more than one children, for any vertex v, the size of the subtree rooted at v is linear to the number of the leaves of the subtree. Therefore, it takes $O(occ)$-time to report all leaves of the subtree rooted at r in line 18 where occ is the number of the outputs.

Algorithm 2.2 Prefix search algorithm for the compressed trie

Require: Any string in $\mathcal{C}$ is not a prefix of another string in $\mathcal{C}$
Ensure: Output all strings in $\mathcal{C}$ prefixed by query P of length m

```
 1: u ← root of the compressed trie
 2: i ← 1
 3: while true do
 4:     if child(u, P[i]) exists then
 5:         v ← child(u, P[i])
 6:         S ← label(u, v)
 7:         h ← |lcp(P[i : m], S)|
 8:         if h = m − i + 1 then
 9:             if h = |S| then
10:                 ℓ ← (v, P[i], 0) and exit from while loop
11:             ℓ ← (u, P[i], h) and exit from while loop
12:         if h < |S| then
13:             abort
14:         (u, i) ← (v, i + |S|)
15:     else
16:         abort
17: r ← vertex v s.t. ℓ = (v, c, i) for some c and i
18: return all leaves of the subtree rooted at v
```

Definition of the suffix tree. Let T be a string of length n. We assume no suffix of T is a prefix of another suffix. One can guarantee that this assumption is met by appending to T a character that does not appear in T, which is denoted

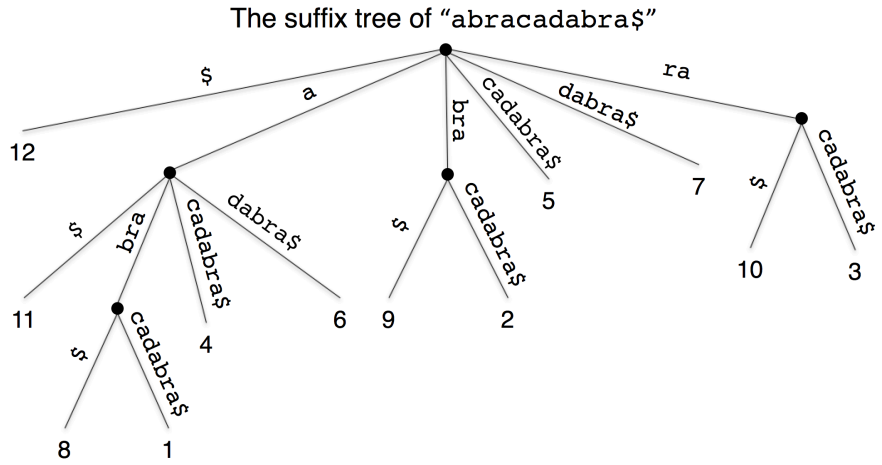


Figure 2.1: An example of the suffix tree. Edges under each internal node are sorted from left right by the first characters of the labels. The character \$ is assumed to be smaller than any other characters.

by \$. The *suffix tree* of T is defined to be the compressed trie storing all suffixes of T. The leaf with path label $T[i : n]$, which we denote by $leaf(i)$, is labeled by i; See Figure 2.1 for an example of the suffix tree. The suffix tree has the following properties:

1. Each edge is labeled by a substring of T;
2. The first characters of the labels of the edges between an internal node and its children differ;
3. Each internal node has more than 1 children;
4. It has n leaves $leaf(1), leaf(2), \ldots, leaf(n)$ and they are aligned from left to right by the lexicographic order of the corresponding suffixes.

Because the number of internal nodes of a tree that satisfies the third property is equal to or less than the number of its leaves minus one, the suffix tree has at most $n - 1$ internal nodes. Because the number of leaves of a tree is the number of vertices minus one, the number of leaves of the suffix tree is at most $n+(n-1)-1 = 2n-2$. Each edge label, for example, $T[i : j]$, can be represented by the tuple (i, j). Thus, the size of the suffix tree is $\Theta(n)$.

Search. Let P be another string of length $m \leq n$. The expression $T[i : i + m - 1] = P$ holds if and only if $T[i : n]$ is prefixed by P. Therefore, all occurrences of P in T can be enumerated in $O(m \log \sigma + occ)$-time by the prefix search algorithm for the compressed trie (Algorithm 2.2).

Construction. The construction algorithms for the suffix tree have a long history. Weiner proposed an $O(n\sigma)$-time algorithm to construct the suffix tree for a length n string on size σ alphabets [142]. McCreight improved it to $O(n \log \sigma)$-time [87]. Ukkonen proposed an algorithm with the same time complexity as McCreight's algorithm but that can run online [130]. Farach proposed an $O(n)$-time algorithm under integer alphabet assumption [25]. Although we do not

describe the details of these algorithms here because they are not directly related to the current thesis, let us mention that the most efficient algorithm by Farach [25] shares the essential ideas with the suffix array construction algorithm by Kärkkäinen and Sanders [55], which we will cover in the next subsection. It is also worth mentioning that, if necessary, the suffix tree can be constructed in $O(n)$-time from the suffix array.

The generalized suffix tree. The suffix tree is a data structure defined for a single string. It is useful when one wants to perform many instances of string search on a single large text. However, there are many applications where one wants to perform search over a collection of texts instead of just one text. One can use *the generalized suffix tree* in such cases.

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a collection of strings. Let T be $T_1 \circ \$_1 \circ T_2 \circ \$_2 \circ \cdots \circ \$_{n-1} \circ T_n \$_n$ where $\$_i$ is a unique character that does not appear anywhere else in T. The generalized suffix tree of $\mathcal{T}$ is defined to be the suffix tree of T with a caveat that the leaf corresponding to $T_i[j : |T_i] \circ \$_i \circ T_{i+1} \circ \cdots \circ T_n \circ \$_n$ be labeled by tuple (i, j). One can use the generalized suffix tree in almost the same manner as the suffix tree to search all occurrences a pattern of length m in the collection in $O(m \log \sigma + occ)$-time.

2.3.2 The Suffix Array

Definition. Let T be a string of length n. The *suffix array* of T, which we denote by SA_T, is an array containing $1, 2, \ldots, n$, sorted by the lexicographic order of $T[1 : n], T[2 : n], \ldots, T[n : n]$, respectively. It was introduced as a compact analogue of the suffix tree. Though both the suffix tree and the suffix array takes $O(n)$-space, the constant factor of the suffix array is smaller. The suffix array takes $4n$ bytes assuming that each entry is represented by a 32 bit integer while the suffix tree takes $20n$ bytes even if implemented carefully [66]. To simulate the functionality of the suffix tree, it is often useful to prepare another data structure called the *height array*. The height array for T, which we denote by HGT_T, is an array of size n such that

$$HGT_T[i] = \begin{cases} 0 & \text{if } i = 0 \\ |lcp(T[SA_T[i-1] : n], T[SA_T[i] : n])| & \text{otherwise.} \end{cases}$$

See figure 2.2 for an example. The suffix array coincides with the labels of the leaves of the suffix tree aligned from left to right. We usually omit the subscript T of SA_T and HGT_T because it is obvious from the context. Even if it is not, we usually choose to state what the underlying string is explicitly and avoid writing the subscript for brevity.

Search. Let P be a string of length m. Let $left(P)$ be $\min\{i : P \preceq T[SA[i] : n]\}$ if $\{i : P \preceq T[SA[i] : n]\}$ is non-empty and $n+1$ otherwise. Also, let $right(P)$ be $\max\{T[SA[i] : n] \preceq P\}$ if $\{T[SA[i] : n] \preceq P\}$ is non-empty and 0 otherwise. $P = T[i : i + m - 1]$ iff $i = SA[j]$ for some $j \in [left(P), right(P)]$. One can find each of $left(P)$ and $right(P)$ by binary search. The pseudocode is shown in Algorithm 2.3. Each comparison of this binary search (line 9) is between

The suffix array and the height array for "abracadabra$"

i	SA[i]	T[SA[i]:12]	HGT[i]
1	12	$	0
2	11	a$	0
3	8	abra$	1
4	1	abracadabra$	4
5	4	acadabra$	1
6	6	adabra$	1
7	9	bra$	0
8	2	bracadabra$	3
9	5	cadabra$	0
10	7	dabra$	0
11	10	ra$	0
12	3	racadabra$	2

Figure 2.2: An example of the suffix array and the height array. The character $ is assumed to be smaller than any other character.

strings and takes $O(m)$-time. Thus, the whole binary search takes $O(m \log n)$-time. Once $left(P)$ and $right(P)$ are derived, one can report all occurrences in $O(occ)$-time by scanning the interval.

Algorithm 2.3 $O(m \log n)$-time algorithm to find $left(P)$

Ensure: $left = left(P)$
1: **if** $P \preceq T[SA[1] : n]$ **then**
2: $left \leftarrow 1$
3: **else if** $P \succ T[SA[n] : n]$ **then**
4: $left \leftarrow n + 1$
5: **else**
6: $(L, R) \leftarrow (1, n)$
7: **while** $R - L > 1$ **do**
8: $M \leftarrow \lfloor (L + R)/2 \rfloor$
9: **if** $P \preceq T[SA[M] : n]$ **then**
10: $R \leftarrow M$
11: **else**
12: $L \leftarrow M$
13: $left \leftarrow R$

In the paper that introduced the suffix array, Manber and Myers showed how to find $left(P)$ and $right(P)$ in $O(m + \log n)$-time with a slight modification to Algorithm 2.3 [83]. The algorithm to find $left(P)$ is shown in Algorithm 2.4. We start from the whole array and successively halve the intervals that contain $left(P)$. Let L and R be the index of the left and right boundaries of the current

interval respectively and let M be $\lfloor \frac{\ell+r}{2} \rfloor$. We maintain $\ell := |lcp(P, T[SA[L] : n])|$ and $r := |lcp(P, T[SA[R] : n])|$. If $\ell \geq r$, the interval is halved as follows. We compute $llcp := lcp(T[SA[L] : n], T[SA[M] : n])$ in constant time. We explain how to do this shortly. If $llcp > \ell$ then $left(P)$ is in $[M, R]$ and ℓ is unchanged. If $llcp < \ell$ then $left(P)$ is in $[L, M]$ and r will be $llcp$. If $llcp = \ell$ then we need to compare P and $T[SA[M] : n]$ to determine if $left(P)$ is in $[L, M]$ or $[M, R]$. Because $lcp(P, T[SA[L] : n]) = \ell = llcp = lcp(T[SA[L] : n], T[SA[M] : n])$, P and $T[SA[M] : n]$ share the first ℓ characters. Thus, it suffices to compare $T[SA[M] + \ell : n]$ and $P[\ell + 1 : m]$. If $P[i]$ is matched to some character of $T[SA[M] + \ell : n]$ in this comparison, then, $P[i]$ is never used in the comparison made in future. Thus, if $\ell \geq r$, either the number of the characters in P that can be used for comparison decreases or the interval $[L, R]$ is halved in constant time. The same is true for the case when $\ell < r$. Therefore, this algorithm takes $O(m + \log n)$-time. The value $right(P)$ can also be found in $O(m + \log n)$-time similarly. Thus, all occurrences of P can be listed in $O(m + \log n + occ)$-time.

Now we explain how to compute $llcp = lcp(T[SA[L] : n], T[SA[M] : n])$ in constant time. Manber and Myers proposed to prepare a table of all possible values of L and M. Here we show a method based on the height array. Note that

$$|lcp(T[SA[i] : n], T[SA[j] : n])| = \min\{HGT[k] : i < k \leq j\}$$

for all $1 \leq i < j \leq n$. Therefore, to compute the length of the longest common prefix of any two suffixes, it suffices to perform the *range minimum query (RMQ)* over the height array. In RMQ problem (on array), the input is an integer array A of size n and two indices i and j with $i \leq j$. The output is the index of minimum element in the subarray $A[i, \ldots, j]$. Bender and Farach-Colton proved the following:[4]

Theorem 2.1 (Bender and Farach-Colton [9])**.** *After $O(n)$-time preprocessing, one can answer range minimum query on array in $O(1)$-time.*

In order to implement Algorithm 2.4, we construct the suffix array from the text, and construct the height from the suffix array and then construct the range minimum data structure for the height array. We show the construction algorithm for the suffix array and the height array below.

Construction. One can construct the suffix array in linear time by constructing the suffix tree and output the leaf labels in the order of depth first search. However, this is not a satisfactory solution, because the use of the suffix tree nullifies the space efficiency of the suffix array. For this reason, many algorithms to construct the suffix array without constructing the suffix tree have been proposed. Here, we explain the algorithm by Kärkkäinen and Sanders [55] because it is relevant to this thesis, especially Chapter 3.

This algorithm takes a string T of length n on integer alphabet. We assume that n is divisible by 3 for brevity. Other cases can be handled with slight modifications. The algorithm is as follows:

[4]Strictly speaking, they showed a reduction from RMQ to lowest common ancestor (LCA) problem. They also proposed a simple algorithm to solve LCA problem in $O(1)$-time after $O(n)$-time preprocessing but this bound was first proven by Harel and Tarjan [38].

Algorithm 2.4 $O(m + \log n)$-time algorithm to find $left(P)$

Ensure: $left = left(P)$
1: $\ell \leftarrow lcp(P, T[SA[1] : n])$
2: $r \leftarrow lcp(P, T[SA[n] : n])$
3: **if** $\ell = m$ or $P[\ell + 1] \leq T[SA[1] + \ell]$ **then**
4: $\quad left \leftarrow 1$
5: **else if** $r < m$ and $P[r + 1] > T[SA[n] + r]$ **then**
6: $\quad left \leftarrow n + 1$
7: **else**
8: $\quad (L, R) \leftarrow (1, n)$
9: $\quad$ **while** $R - L > 1$ **do**
10: $\quad\quad M \leftarrow \lfloor (L + R)/2 \rfloor$
11: $\quad\quad$ **if** $\ell \geq r$ **then**
12: $\quad\quad\quad llcp \leftarrow lcp(T[SA[L] : n], T[SA[M] : n])$
13: $\quad\quad\quad$ **if** $llcp \geq \ell$ **then**
14: $\quad\quad\quad\quad mid \leftarrow \ell + lcp(T[SA[M] + \ell : n], P[\ell + 1 : m])$
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad mid \leftarrow llcp$
17: $\quad\quad$ **else**
18: $\quad\quad\quad rlcp \leftarrow lcp(T[SA[R] : n], T[SA[M] : n])$
19: $\quad\quad\quad$ **if** $rlcp \geq r$ **then**
20: $\quad\quad\quad\quad mid \leftarrow r + lcp(T[SA[M] + r : n], P[r + 1 : m])$
21: $\quad\quad\quad$ **else**
22: $\quad\quad\quad\quad mid \leftarrow rlcp$
23: $\quad\quad$ **if** $mid = m$ or $P[mid + 1] \leq T[SA[M] + mid]$ **then**
24: $\quad\quad\quad (R, r) \leftarrow (M, mid)$
25: $\quad\quad$ **else**
26: $\quad\quad\quad (L, \ell) \leftarrow (M, mid)$
27: $\quad left \leftarrow R$

1. Radix sort tuples $\{(T[i], T[i + 1], T[i + 2]) : 1 \leq i \leq n\}$;
2. Let $r(i)$ be the rank of $(T[i], T[i + 1], T[i + 2])$. Construct the table for r from the sorted array derived at previous step;
3. Prepare strings $T_1 := r(1) \circ r(4) \circ \cdots \circ r(n - 2), T_2 := r(2) \circ r(5) \circ \cdots \circ r(n - 1), T_{12} := T_1 \circ 0 \circ T_2$;
4. Recursively construct SA', the suffix array for T_{12}. This is equivalent to sorting $\{T[i : n] : i \equiv 1 \text{ or } 2 \pmod 3\}$;
5. Let $r'(i)$ be the rank of $r(3i + 1) \circ r(3i + 4) \circ \cdots \circ r(n - 2)$ among the suffixes of T_1. Construct the table for r' from SA';
6. Radix sort tuples $\{(T[3i], r'(i)) : 1 \leq i \leq n/3\}$. This is equivalent to sorting $\{T[i : n] : i \equiv 0 \pmod 3\}$;
7. At this point, we essentially have two sorted arrays: one for $\{T[i : n] : i \equiv 1 \text{ or } 2 \pmod 3\}$ and the other for $\{T[i : n] : i \equiv 0 \pmod 3\}$. Prepare the

table for the ranks of $\{T[i:n] : i \equiv 1 \text{ or } 2 \pmod 3\}$. Merge the sorted arrays. The comparison between $T[i:n]$ with $i \equiv 1$ or 2 (mod 3) and $T[j:n]$ with $j \equiv 0 \pmod 3$ is done as follows:

- If $i \equiv 1 \pmod 3$, compare $T[i]$ and $T[j]$. If they match, compare $T[i+1:n]$ and $T[j+1:n]$. The latter comparison can be done by looking up the rank table because $i+1 \equiv 2 \pmod 3$ and $j+1 \equiv 1 \pmod 3$;
- If $i \equiv 2 \pmod 3$, compare $T[i:i+1]$ and $T[j:j+1]$. If they match, compare $T[i+2:n]$ and $T[j+2:n]$. The latter comparison can be done by looking up the rank table because $i+2 \equiv 1 \pmod 3$ and $j+2 \equiv 2 \pmod 3$.

Every step but step 4 takes $O(n)$-time. Because T_{12} consists of $2n/3+1$ characters from integer alphabet, step 4 takes time at most $f(\frac{2}{3}n+1)$ where $f(n)$ is the maximum time needed in total. Thus, $f(n) \leq f(\frac{2}{3}n+1)+c_1 n$ for some constant c_1. Also, $f(c_2) = O(1)$ for any constant c_2. Therefore, $f(n) = O(n)$.

Height Array Construction. Kasai et al. gave an $O(n)$-time algorithm to construct the height array from the suffix array [56]. For brevity, we assume the last character of T, $T[n]$, is smaller than any other character in T. One can guarantee that this assumption is met by appending a character with this property to T. The algorithm is described in Algorithm 2.5. The logic is as follows. At the i-th iteration of the while loop, the algorithm computes the length of lcp between $T[i:n]$ and its predecessor in the suffix array, for example, $T[j:n]$. If $|lcp(T[i:n], T[j:n])| = h > 0$, $|lcp(T[i+1:n], T[j+1:n])| = h-1$ and $T[j+1:n] < T[i+1:n]$. In this case, the length of lcp between $T[i+1:n]$ and its predecessor in the suffix array is at least $h-1$. Thus, at the $i+1$-th iteration, it suffices to start comparing strings from the h-th characters. Each of line 2,7 and 11 takes constant time and thus, all operations except line 9 take $\Theta(n)$-time. Because h is the length of lcp, it cannot be larger than n. Also, h is decremented (at line 7) for at most n times. Thus, the number of times line 9 is executed is bounded by $2n$. Therefore, this algorithm takes $\Theta(n)$-time in total.

Algorithm 2.5 Construction of the height array.

Ensure: HGT is the height array of T

1: **for** $i = 1$ to n **do**
2: $\quad R[SA[i]] = i$
3: $HGT[1] \leftarrow 0$
4: $h \leftarrow 0$
5: **for** $i = 1$ to $n-1$ **do**
6: $\quad h \leftarrow \max\{h-1, 0\}$
7: $\quad$ **while** $T[SA[R[i]]+h] = T[SA[R[i]-1]+h]$ **do**
8: $\quad\quad h \leftarrow h+1$
9: $\quad HGT[R[i]] \leftarrow h$

The generalized suffix array. As well as the suffix tree, one can also apply the suffix array to a collection of strings. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a collection of strings. The *generalized suffix array* is defined to be the suffix array of the string $T_1 \circ \$_1 \circ T_2 \circ \$_2 \circ \cdots \circ \$_{n-1} \circ T_n$ where $\$_i$ is a character that does not appear in any other place in $\mathcal{T}$. Because this is just the suffix array for a long string, the height array is also defined for the generalized suffix array and algorithms for search and construction for the suffix array can also be applied to the generalized suffix array. When $\sum_{k=1}^{j-1}(|T_k|+1) < i \leq \sum_{k=1}^{j}(|T_k|+1) - 1$, we say that the index i belongs to T_j.

2.3.3 The Isuffix Tree

Definition. Next, we describe a generalization of the suffix tree for matrices. There are several data structures for this purpose generically called the two dimensional suffix trees [31, 32, 59]. Here, we focus on the *Isuffix tree* by Kim et al. [59] because it can be constructed in linear time.

An *Istring* is a string of strings defined for a square matrix. For an $n \times n$ square matrix M, the Istring of M, denoted by $I(M)$ is a length $2n-1$ string of strings. The characters of $I(M)$ are defined as follows:

$$I(M)[i] = \begin{cases} M[1:(i+1)/2, (i+1)/2] & \text{if } i \text{ is odd} \\ M[i/2+1, 1:i/2] & \text{if } i \text{ is even.} \end{cases}$$

The string $I(M)[i]$ for each i is called an *Icharacter*. An Istring Is_1 is defined to precede another Istring Is_2 if Is_1 is a null Istring or $Is_1[1]$ precedes $Is_2[1]$ or $Is_1[2:|Is_1|]$ precedes $Is_2[2:|Is_2|]$. Note that it is equivalent to $I_1[1] \circ I_1[2] \circ \cdots \circ I_1[|I_1|]$ precedes $I_2[1] \circ I_2[2] \circ \cdots \circ I_2[|I_2|]$. The suffix of an $m \times n$ matrix M at position (i,j), denoted by $M_{i,j}$, is $M[i:i+k, j:j+k]$ where $k = \min\{m-i, n-j\}$. In other words, $M_{i,j}$ is the largest square submatrix of M whose upper left corner is (i,j).

The Isuffix tree of an $m \times n$ matrix M is a compressed trie storing Istrings of all suffixes of M; see Figure 2.3 for an example. Note that while the suffix tree stores strings, the Isuffix tree stores Istrings, i.e., strings of strings. Therefore, each edge label is an Istring instead of a string. Because each edge label is equal to $I(M_{i',j'})[i:j]$ for some suffix $M_{i',j'}$ and i,j, it can be represented by tuple (i',j',i,j). Thus, the Isuffix tree takes $\Theta(mn)$-space. The path label of a node is the concatenation of the edge labels of edges in the path from the root to the node. The depth of a node is the number of Icharacters in the path label of the node. To guarantee the one-to-one correspondence between leaves and suffixes, one can append a row at the bottom of M and a column on the right of M consisting of unique elements.

Search. Let M be an $n_1 \times n_2$ matrix and P be an $m \times m$ matrix. To find all occurrences of P in M, it suffices to find suffixes of M whose Istring is prefixed by $I(P)$. Because the Isuffix tree is a compressed trie, the prefix search algorithm for the compressed trie shown in Algorithm 2.2 is also applicable to the Isuffix tree.

The difference is that edge labels of the Isuffix tree are Istrings and thus, to find the edge to follow at each internal node, we need to perform binary search

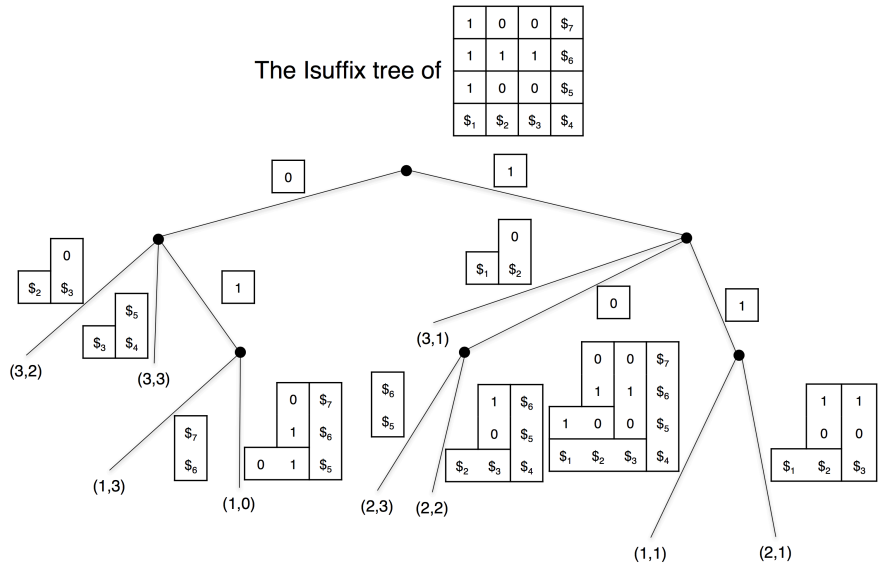


Figure 2.3: An example of the Isuffix tree. Length 1 suffixes (those submatrices consisting only of 1 entry $\$_i$ for $1 \leq i \leq 7$) are not shown.

on strings instead of characters. The length of each string is at most m. Also, the number of children of each internal node can be larger than σ. At any rate, it is never greater than the number of leaves $n_1 n_2$. Thus, the binary search at each node takes $O(m \log n_1 n_2)$-time. Because $I(P)$ contains $2m-1$ Icharacters, the search takes $O(m^2 \log n_1 n_2)$-time in total.

Construction. Kim et al. [59] proposed an algorithm to construct the Isuffix tree for an $m \times n$ matrix on an integer alphabet in $O(mn)$-time. Instead of covering the detail of this algorithm, we just mention that the proposed algorithm follows the recursive strategy similar to the suffix tree construction algorithm by Farach [25] and the suffix array construction algorithm by Kärkkäinen and Sanders [55].

The generalized Isuffix tree. Let M_i be an $m_i \times n_i$ dimensional matrix for $1 \leq i \leq \ell$. Let $m := \max\{m_i\}+1$. Let M be the matrix derived by the following procedure:

1. Let M'_k be an $m \times (n_k + 1)$ matrix s.t.

$$M'_k[i:j] = \begin{cases} M_k[i:j] & \text{if } 1 \leq i \leq m_k, 1 \leq j \leq n_k \\ \mathsf{dummy} & \text{otherwise;} \end{cases}$$

2. Concatenate $M'_1, M'_2, \ldots, M'_\ell$ from left to right;

3. Replace each dummy by a unique character.

We define the generalized Isuffix tree of $M_1, M_2, \ldots, M_\ell$ as the Isuffix tree of M. M is an $m \times (\sum_{i=1}^{\ell} n_i + \ell)$-dimensional matrix. Thus, the generalized Isuffix tree of M takes $\Theta(m \sum_{i=1}^{\ell} n_i)$-space.

2.4 Support Vector Machines and Kernel Methods

Support vector machines (SVMs) are a group of classification problems under supervised setting and algorithms to solve these problems.[5] We apply SVMs for protein sequence classification in Chapter 4 and protein structure classification in Chapter 5. The purpose of this section is to give the background on SVMs necessary to read these forthcoming chapters. Fortunately, we apply SVMs through techniques called *kernel methods*, which works in modular fashion with SVMs. This enables us to restrict most of our attention to "kernel"s instead of SVMs in the latter chapters. Thus, we make the explanation of SVMs as concise as possible and put an emphasis on its relationship with kernel methods. Each of SVMs and kernel methods has a rich theory that is not covered here. For more detailed information about SVMs and kernel methods, refer to the book by Vapnik [133] and the book by Shawe-Taylor and Cristianini [117] respectively.

This section involves descriptions of optimization problems. For a variable x of an optimization problem, we denote the setting of x that attains the optimal solution of the problem as x^*.

2.4.1 Supervised Classification Problem

We first briefly explain what classification problems are in supervised settings. A supervised classification problem is a problem such as inferring the type of characters from the image of handwritten characters. For these problems, it is difficult for a programmer to hard code a function that maps any (or even most) input image to the correct character type because such a function would be too complex to handle explicitly. Instead, a reasonable approach is to let the machine acquire classification rules by itself through concrete examples with the guidance of a human (or any other exterior entity) acting as a teacher. In fact, it seems more similar to the way we humans learn characters. Researchers have been studying methods to let machines to do such inductive reasoning, not necessarily be limited to classification, for decades. Originally, these studies started in the context of artificial intelligence. However, the scope of the theory and methods developed in this area has been gradually expanded to more practical realm and, at present, the applications include image recognition, audio recognition and anomaly detection to name a few.

In general, a supervised classification problem conform to the following framework:

- At first, the algorithm is given the training set, a set of data already labeled by their groups;
- (Learning phase) From the training set, the algorithm learns the rule of classification by any efficient computation;
- (Prediction phase) The algorithm is given the test set, another set of data that is not labeled. It predicts the group to which data in the test set should belong by applying the classification rule it learned in the learning phase.

[5] SVMs are also applied to regression. We use them only for classification in this thesis.

The important assumption that must be made in any classification problem is that the training set and the test set are derived from the same source. For example, in character recognition, if the training set consists of the images of Japanese characters, then the images for test should also be Japanese characters, not, for example, Roman characters. Otherwise, it is trivially impossible to solve classification problem. In theoretical work, this assumption is usually formulated by letting the training and test set sampled from the same universe with the same probability distribution independently.

Once the classification problem is set up as above, one engineering type goal is to derive better algorithms for the learning phase and prediction phase. Shawe-Taylor and Cristianini listed 3 properties that a learning algorithm should possess [117]:

- (Computational efficiency) Learning algorithm runs efficiently;
- (Robustness) Errors in the input does not have much effect on the result of learning;
- (Statistical stability) The choice of training set does not have much effect on the result of learning.

2.4.2 Support Vector Machines

As is mentioned at the beginning of this section, SVMs are a group of supervised learning problems and methods to solve these problems. In these problems, each datum is a point in the Euclidean space. Each point in the training set is labeled as either positive or negative. In the learning phase, the algorithm finds a hyperplane that separate positively labeled points and negatively labeled points with as large "margin" as possible. In the prediction phase, the algorithm classifies each test datum according to which side of the hyperplane it resides. In general, the positively labeled points and negatively labeled points are not linearly separable. Thus, errors are allowed with penalties. The formal definition of the learning phase of SVM is as follows:[6]

Problem 2.1 (Learning phase of SVM)**.**

$$\begin{aligned} &\textit{Given } \{\mathbf{x}_i, y_i\}_{i=1}^{\ell} \subset \mathbb{R}^d \times \{-1, 1\} \textit{ and } C > 0, \\ &\textit{minimize } \ -\gamma + C\sum_{i=1}^{\ell} \zeta_i, \\ &\textit{subject to } y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq \gamma - \zeta_i, \quad \zeta_i \geq 0 \textit{ for } i = 1, 2, \ldots, \ell \\ &\qquad\qquad \textit{and } \|\mathbf{w}\| = 1. \end{aligned}$$

Note that $y_i(\mathbf{w}^\top \mathbf{x}_i + b)$ is the distance between the hyperplane $\mathbf{w}^\top \mathbf{x} = 0$ and $\mathbf{x}_i$ where a negative number means $\mathbf{x}_i$ is on the wrong side of the hyperplane. The objective function prefers a larger γ and smaller ζ_i's and therefore larger $y_i(\mathbf{w}^\top \mathbf{x}_i + b)$'s. Thus, parameters γ, ζ_i and C can be interpreted as the size of the margin, penalty for misclassifying $\mathbf{x}_i$ and the weight of penalties respectively. This problem is an instance of convex optimization problem and can be

[6] Precisely speaking, this problem is called 1-norm soft margin SVM.

efficiently solved. Also, there is a theoretical bound for the error probability in the prediction phase. (See, for example, the Theorem 7.30 of [117]) Therefore, SVMs satisfy the 3 desirable properties of learning algorithms mentioned in the previous subsection.

In prediction phase, query $\mathbf{x}$ is judged to be positive (resp. negative) if ${\mathbf{w}^*}^\top \mathbf{x} + b^*$ is positive (resp. negative). Thus, the prediction phase takes $\Theta(d)$-time.

However, this formulation has two drawbacks:

1. it assumes the inputs to be points in Euclidean space;
2. it captures the boundary between the classes only by a hyperplane and cannot, for example, separate intertwined input points by a curve.

2.4.3 Kernel Methods for Support Vector Machines

A natural way to avoid the aforementioned drawbacks of SVMs is to introduce *feature map* ϕ, a map from the space of data to Euclidean space, and instead of feeding data themselves to SVMs, feeding the image of feature maps to SVMs. The range of feature map is called *feature space* and an image of feature map is called a *feature vector*. Any map is a valid feature map if the feature space is in Euclidean space. One can apply SVMs to non-vectorial data by first mapping the data to feature vectors by feature map. Also, though SVMs can only handle hyperplanes, the feature map may be non-linear. In fact, feature maps can be such that linearity is not even defined. Of course, to achieve a meaningful classification, the feature map should capture the properties of the data that are relevant to classification.

One can use feature maps for Problem 2.1. In this case, the input data $x_1, x_2, \ldots, x_\ell$ are not necessarily vectors. By some feature map ϕ, we map these data to d-dimensional vectors $\phi(x_1), \phi(x_2), \ldots, \phi(x_\ell)$. The learning problem is the same as Problem 2.1 except that $\mathbf{x}_i$ is replaced by $\phi(x_i)$. In the prediction phase, we map query x to $\phi(x)$ and calculate the sign of ${\mathbf{w}^*}^\top \phi(x) + b$. While this use of feature maps is already useful, in some cases, it is more advantageous to use feature maps for the dual problem of Problem 2.1.

The dual problem of Problem 2.1 is as follows:

Problem 2.2 (The dual problem of SVM learning)**.**

$$\begin{aligned} &\textit{Given } \{\mathbf{x}_i, y_i\}_{i=1}^\ell \subset S \times \{-1, 1\} \textit{ and } C > 0, \\ &\textit{maximize } -\sum_{i,j=1}^{\ell} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^\top \mathbf{x}_j, \\ &\textit{subject to } \sum_{i=1}^{\ell} y_i \alpha_i = 0, \quad \sum_{i=1}^{\ell} \alpha_i = 1 \\ &\qquad \textit{and } 0 \leq \alpha_i \leq C \textit{ for } i = 1, 2, \ldots, \ell. \end{aligned}$$

Let $\alpha := (\alpha_1, \alpha_2, \ldots, \alpha_\ell)$. Let $\mathbf{x}_{\mathsf{neg}}$ and $\mathbf{x}_{\mathsf{pos}}$ be two inputs s.t. $y_{\mathsf{neg}} = -1, y_{\mathsf{pos}} + 1$ and $0 < \alpha^*_{\mathsf{neg}}, \alpha^*_{\mathsf{pos}} < C$. The optimal parameters $\mathbf{w}^*, b^*$ and γ^*

of Problem 2.1 is related to α^* as follows:

$$\begin{aligned}\mathbf{w}^* &= \sum_{i=1}^{\ell} y_i \alpha_i^* \mathbf{x}_i; \\ b^* &= -\frac{1}{2}\left(\mathbf{x}_{\mathsf{neg}}^\top \mathbf{w}^* + \mathbf{x}_{\mathsf{pos}}^\top \mathbf{w}^*\right); \\ \gamma^* &= \mathbf{x}_{\mathsf{pos}}^\top \mathbf{w} + b^*.\end{aligned}$$

In learning phase, query $\mathbf{x}$ is judged to be positive (resp. negative) if ${\mathbf{w}^*}^\top \mathbf{x} + b^*$ is positive (resp. negative). In the learning algorithm for Problem 2.2, $\mathbf{x}_i$'s always appear as inner products such as $\mathbf{x}_i^\top \mathbf{x}_j$ (See page 223 of [117] for the algorithm.). Similarly, in the prediction phase, $\mathbf{x}_i$'s and $\mathbf{x}$ always appear as inner products.

When we use feature map ϕ for Problem 2.2, the input data $x_1, x_2, \ldots, x_\ell$ are not necessarily vectors. The description above remains unchanged except that each $\mathbf{x}_i$ is replaced by $\phi(x_i)$. The real advantage of using feature maps for Problem 2.2 is that, for some feature maps, it is possible to compute the *kernel function* $K(x, y) := \phi(x)^\top \phi(y)$ without constructing the feature vectors. Because, as we have observed above, both learning and prediction can be implemented without using the (feature) vectors $\mathbf{x}_i = \phi(x_i)$'s and $\mathbf{x} = \phi(x)$ as long as the kernel function can be computed, we do not need to compute the feature vectors at all. If we compute $K(x, y)$ by explicitly constructing $\phi(x)$ and $\phi(y)$, it takes at least $\Theta(d)$-time where d is the dimension of $\phi(x)$ and $\phi(y)$. On the other hand, if we can compute $K(x, y)$ directly from x and y, sometimes it is possible to compute it in time independent or less dependent on d. This enables us to consider feature maps with high dimensional feature space without growing computational cost much and it often is advantageous to design accurate classifiers. This implicit use of feature maps through kernel functions is called *kernel methods*.

Another good property of kernel methods is the modularity. As long as it has an access to the oracle for the kernel function, SVM solvers can run. Therefore, the user of SVM solvers can focus on the design of kernel functions appropriate for the problem at hand.

Multiple kernels also work in modular fashion. If K_1 (resp. K_2) is a kernel function with corresponding feature map ϕ_1 (resp. ϕ_2), then for any non-negative real numbers α_1 and α_2, $\alpha_1 K_1 + \alpha_2 K_2 : (x, y) \mapsto \alpha_1 K_1(x, y) + \alpha_2 K_2(x, y)$ is a kernel function. The corresponding feature map maps x to the concatenation of $\sqrt{\alpha_1}\phi_1(x)$ and $\sqrt{\alpha_2}\phi_2(x)$.

Summary of kernel-based SVMs. In this thesis, we use SVMs only in the form of Problem 2.2 and with kernel methods. Thus, we summarize computations needed in this case.

Let $x_1, x_2, \ldots, x_\ell$ be the input for the learning phase, i.e., the training data set. Then, we need to compute the kernel function value $K(x_i, x_j)$ for every pair of data (x_i, x_j). The $\ell \times \ell$ matrix containing $K(X_i, x_j)$ at the i, j entry is called the kernel matrix. Then we input the kernel matrix to SVM solvers such as LIBSVM [19] and obtain α^* and b^*. In the prediction phase, we are given a

query x and need to compute ${\mathbf{w}^*}^\top \phi(x) + b^*$ using the following equation:

$$\begin{aligned}
{\mathbf{w}^*}^\top \phi(x) &= \left(\sum_{i=1}^{\ell} y_i \alpha_i^* \phi(x_i) \right)^\top \phi(x) \\
&= \sum_{i=1}^{\ell} y_i \alpha_i^* \phi(x_i)^\top \phi(x) \\
&= \sum_{i=1}^{\ell} y_i \alpha_i^* K(x_i, x).
\end{aligned}$$

The x_i s.t. $\alpha_i \neq 0$ is called a *support vector*.[7] In the prediction phase, we do not need to compute the term $y_i \alpha_i^* K(x_i, x)$ if x_i is not a support vector.

2.5 Basic Concepts from Molecular Biology

In this section, we introduce prerequisite knowledge of molecular biology necessary to read the following chapters. For more detailed information about molecular biology, refer to the book by Alberts et al. [2].

Cells are the basic building block that is observed in all organisms.[8] There are two types of cells: *prokaryotic cells* and *eukaryotic cells*. Eukaryotic cells contain organella, structures separated from other parts of the cell by membranes while prokaryotic cells are more primitive and do not have such structures.

At molecular level, *Deoxyribonucleic acid (DNA)*, *Ribonucleic acid (RNA)*, and *proteins* are three macromolecules that are essential for most organisms. The *central dogma of molecular biology* [22], roughly speaking, states that, in biological systems, information flow from DNA to RNA, RNA to proteins or DNA to DNA.[9] We elaborate on how these processes work.

DNA and RNA molecules consist of linearly connected nucleotides. A nucleotide consists of a sugar, a phosphate group and nitrogenous base. The type of a nucleotide depends on the base type.[10] The base types in DNA are adenine (`A`), guanine (`G`), cytosine (`C`) and thymine (`T`). RNA contains uracil (`U`) instead of `T`. A DNA molecule consists of two chains of nucleic acids, which form a double helical structure. The nucleotides in one helix are associated to those in the other helix through the base-paring rule of Watson and Crick [141] (`A-T` and `G-C`). Thus, the sequence on one helix determines the other, and a DNA molecule can be thought of as a string on alphabet `A`, `G`, `C`, `T`. In an RNA molecule the chain usually does not form such a pair. Thus, an RNA molecule can be thought of as a string on alphabet `A`, `G`, `C`, `U`. Proteins consist of linearly connected amino acids. There are 20 types of amino acids and thus, an protein can be thought of as a sequence on a size 20 alphabet.

[7] In the original SVM problem where the input data are vectors, the support vectors are vectors. In this thesis, we use the term support vectors even if the input data are not really vectors.

[8] Viruses do not have cells. There is no consensus on whether viruses should be considered as living or not.

[9] There is some exceptions to the central dogma. Most notably, a retrovirus stores its genetic information in RNA and create the copies of it by first transcribing it into DNA and injecting it into the genome of host organisms. The injected DNA is transcribed to RNA by the host.

[10] The difference between DNA and RNA are the type of sugars. The sugar in DNA is deoxyribose while that in RNA is ribose.

The function of DNA is to store information. Some parts of DNA sequence called *genes* encode proteins.[11] A gene can be transformed into an RNA sequence (*transcription*), which, in turn, can be transformed into a protein sequence (*translation*). The protein (resp. RNA) sequence is uniquely determined by the RNA (resp. DNA) sequence. This process of protein production is called *gene expression*. DNA sequences also contain *regulatory sequences*, which are sequences that encode information on how much proteins should be produced in what situations. For example, a promoter is a sequence that, by binding to a particular protein, initiates the transcription of a particular gene.

In a protein molecule, amino acids are connected linearly by *peptide bond*. Figure 2.4 shows how two amino acids are connected. The 20 types of amino acids represent 20 types of side-chains. Proteins are involved in numerous biological phenomena. For example, some proteins can recognize a particular promoter sequence (cf. the previous paragraph) and initiate transcription. Other typical functions of proteins include the catalysis of biochemical processes and DNA replication. They are also structural building block of living things. These functions of proteins are determined by their 3-dimensional structures. In natural environment, a protein molecule is folded into a particular shape called the *native structure* and it is this shape that determines the function of the protein; see also Figure 2.5 to get a grasp on the protein structures. For example, an enzyme's selectivity is realized by the shape of its substructure that fits to the molecules it acts on. The 3-dimensional structure of a protein is often represented by the sequence of the coordinates of C_α atoms on backbone (See the legend of Figure 2.4 for the definition of C_α atoms and backbone). By the nature of peptide bond, the distance between a pair of C_α atoms neighboring on the backbone is restricted to be about 3.8Å.

Individual organism can create other individuals of the same species and this process is called *reproduction*. There are two types of reproductions: asexual reproduction and sexual reproduction. In asexual reproduction, children inherit the copy of the parent's DNA sequence. In sexual reproduction, each child has two parents and is given a DNA sequence generated from the parents' DNA sequence. Sometimes, DNA sequences are accidentally modified by replication errors or external factors such as exposure to radiation. Therefore, the DNA sequences of the descendants of the common ancestor diverge as generations go by. Accordingly, protein sequences, which are encoded by DNA sequences, also acquire variations. Sequences with the common ancestry are said to be *homologous*.[12] This diversification process is affected by *natural selection*. Some variations affect the probability of themselves being passed to the next generation. While those variations that increase the probability are more likely to spread among the population, those variations that decrease the probability are less likely to spread and sometimes go extinct.

[11] Sometimes, the term 'gene' is also defined to be the basic unit of evolution.

[12] The word 'homology' is also used to refer to anatomical features derived from the same ancestry.

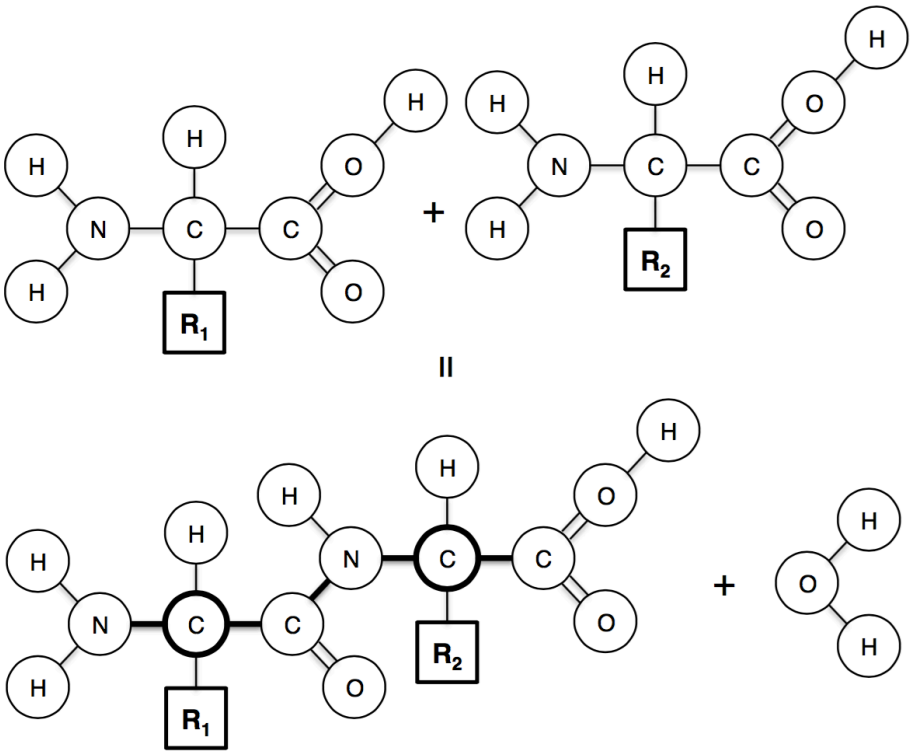


Figure 2.4: Peptide bond. A circle with thick outline represents an α-carbon (C_α). A square represents a side-chain. Proteins are made up of 20 types of side-chains. Thick lines represent the backbone.

Figure 2.5: The 3-dimensional structure of a protein called ribonuclease S (PDB ID 2RNS). The image was produced by JSmol program at `www.rcsb.org`.

2.6 Sequence Alignment

In this section, we briefly explain *sequence alignment*, the foundation of biological sequence comparison. For more detailed explanation about sequence alignment, refer to the book by Gusfield [37]. As is mentioned in the previous section, through evolution, DNA and protein sequences acquire variations over time. The evolutionary relationships, i.e., homologies between sequences provide clues to infer the functions of genes or proteins. Because homologies are not directly observable, usually, sequence similarities are used as a proxy for homologies. Sequence alignment is a standard method to measure biological sequence similarities.

We describe the simplest type of alignment called global alignment. Let S (resp. T) be a string of length n (resp. m). Let $\Gamma := \Sigma \cup \{\texttt{-}\}$, where - is a special symbol that is not in Σ. An alignment A between S and T is a $2 \times \ell$ matrix of elements of Γ that satisfies the following conditions: (See also Figure 1.1)

1. $\max n, m \leq \ell \leq n + m$;
2. The concatenation of the characters in the first (resp. second) row of A that belongs to Σ must be S (resp. T);
3. At least one character of each column must belong to Σ.

For each pair $(c_1, c_2) \in \Gamma \times \Gamma$, cost $\delta(c_1, c_2)$ is defined. The cost of alignment $\delta(A)$ is defined to be $\sum_{i=1}^{\ell} \delta(A[1][i], A[2][i])$.

In global alignment problem, the input is S, T and δ and the goal is to find alignment A of S and T that minimizes $\delta(A)$. Though the number of possible alignments is exponential to the input size, the optimal alignment can be computed in $O(nm)$-time by dynamic programming [137, 97, 114, 115, 138, 139]

The biological interpretation of alignments goes as follows: If $S[i]$ and $T[j]$ are aligned, i.e., they are in the same column, and $S[i] = T[j]$, these characters are thought to be representing the same nucleotide or amino acid that was passed from the common ancestor. If $S[i]$ and $T[j]$ are aligned but $S[i] \neq T[j]$, then these characters are thought to be representing the mutation of single nucleotide or amino acid. A column consisting of a character from Σ and - indicates that insertion or deletion of single nucleotide or amino acid.

There are many variations of alignment problems other than global alignment. For example, in local alignment, the optimal score among all global alignments between all pairs of substrings of S and T is sought [125]. This is useful for detecting similar sequences embedded in unrelated sequences. There are also more complex scoring schemes. For example, in a scheme called affine gap penalty, creating gaps is more expensive than extending them [35]. There are algorithms with better time bounds for the cases when the input sequences or the score matrix satisfies some assumptions [68, 95, 90, 30]. However, the best existing time bound for the general sequence alignment is $O(n^2/\mathrm{polylog}(n))$-time where n is the size of the input sequences [85, 23]. It is known that the existence of $O(n^{2-\epsilon})$-time algorithm for any constant $\epsilon > 0$ would refute a famous conjecture in computational complexity theory called strong exponential time hypothesis [6]. Also, there exist an approximation algorithm for sequence alignment that runs in $O(n^{1+\epsilon})$-time for any constant $\epsilon > 0$ but the approximation

factor is polylog(n). Therefore, developing algorithms with better time bounds for general sequence alignment problem seems to be very difficult.

2.7 A Hash Function for Strings

In this section, we introduce a concrete construction of a hash function for (fixed length) strings we use in Chapter 4 and Chapter 5. The contents of this section are based on the textbook by Mitzenmacher and Upfal [91]. Let V be the set $\{1, 2, \ldots, m\}$ and let U be the universe of the keys of size larger than m.

First we prepare a definition and a proposition.

Definition 2.1. *A family of hash functions $\mathcal{H}$ from U to V is said to be strongly universal if, for any keys $k_1, k_2 \in U$ s.t. $k_1 \neq k_2$ and any values $v_1, v_2 \in V$, $Pr[h(k_1) = v_1$ and $h(k_2) = v_2] = 1/m^2$.*

Proposition 2.1. *If $\mathcal{H}$ is strongly universal and h is a hash function chosen from $\mathcal{H}$ uniformly randomly, then for any $\{k_1, k_2, \ldots, k_n\} \in U$ and any $k \in U$, $E[\#\{i : h(k_i) = h(k)\}] \leq 1 + n/m$ where $E[\cdot]$ denotes the expectation.*

Proof. Let I_i be a random variable s.t. $I_i = 1$ if $h(k_i) = h(k)$; 0 otherwise. Then, $E[\#\{i : h(k_i) = h(k)\}] = E[\sum_i I_i] = \sum_i E[I_i] = \sum_i Pr[h(k_i) = h(k)]$. Also, $Pr[h(k_i) = h(k)] = \sum_{j=1}^{m} Pr[h(k_i) = j$ and $h(k) = j] = \sum_{j=1}^{m} 1/m^2 = 1/m$ if $k_i \neq k$ (The second equality follows from the strong universality) and $Pr[h(k_i) = h(k)] = 1$ if $k_i = k$. Thus, $\sum_{i=1}^{n} Pr[h(k_i) = h(k)] \leq 1+(n-1)/m < 1+n/m$. □

Corollary 2.1. *If we implement a hash table by a strongly universal hash function family, and set the table size linear to the number of items, then, each operation takes constant time in expectation.*

We use the following proposition to construct the hash function for strings. Let key k be a length ℓ string.

Proposition 2.2 ([91] Lemma 13.8)**.** *Let p be a large prime number s.t. $p > \sigma$. For $\mathbf{a} := (a_1, a_2, \ldots, a_k)$ with $0 \leq a_i \leq p-1 (1 \leq i \leq \ell)$, and value b with $0 \leq b \leq p-1$, let $h_{\mathbf{a},b}$ be a map from $\{1, \ldots, p\}^\ell$ to $\{1, \ldots, p\}$ s.t. $h_{\mathbf{a},b}(k) = (\sum_{i=1}^{\ell} a_i k[i] + b) \mod p$. Then, the set of functions $\{h_{\mathbf{a},b} : 0 \leq a_i, b \leq p-1$ for all $1 \leq i \leq \ell\}$ is strongly universal.*

Corollary 2.2. *With the same settings as Proposition 2.2, let $h_{\mathbf{a},b}$ be a map from $\{1, \ldots, p\}^\ell$ to $\{1, \ldots, n\}$ s.t. $h_{\mathbf{a},b}(k) = ((\sum_{i=1}^{\ell} a_i k[i] + b) \mod p) \mod m$ where $m < p$. Then, $\{h_{\mathbf{a},b} : 0 \leq a_i, b \leq p-1$ for all $1 \leq i \leq \ell\}$ is strongly universal.*

To sum up, we showed a concrete construction of strong universal hash function family for fixed length strings. By Corollary 2.1, the hash table implemented by this hash function family, when the table is allocated size linear to the number of items contained, can perform every operations in $O(1)$-time in expectation.

Chapter 3

Text Indexing with Gaps

3.1 Overview

In this chapter, we introduce a data structure and its construction algorithms we use in Chapter 4. The proposed data structure is a variant of the suffix array, which we call the b-suffix array. The parameter b is a binary string and, informally, the b-suffix array is an array containing suffixes sorted considering only the i-th characters s.t. $b[i] = 1$; see Figure 3.1 for an example.

The main motivation to introduce the b-suffix array is the application to the computation of the kernel function for strings we introduce in Chapter 4. By using the b-suffix array, one can compute the kernel function without computing the corresponding feature vectors explicitly. This algorithm for kernel function computation makes it possible to consider high dimensional feature vectors keeping the computational cost independent of the dimension. The detail of the string kernel is described in Chapter 4.

Aside from the application to Chapter 4, the b-suffix array, as a text index, supports the search of patterns with wildcards in certain predetermined positions in time logarithmic to the length of the text. A wildcard `?` is a special character that can match any character. For example, when $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, the pattern `A?C` matches `AAC`, `ACC`, `AGC` and `ATC`. Such patterns arise in spaced seed-based filtering algorithms for sequence homology search. We explain about it in Subsection 3.1.1.

Construction algorithms. The technical contributions of this chapter are the construction algorithms for the b-suffix array. Informally, the main problem and the results are summarized as follows (for formal statements, see Theorem 3.1 and 3.2):

Problem 3.1. *Given a string T of length n and a binary string b of length m, sort the suffixes of T considering only the i-th characters for i s.t. $b[i] = \mathtt{1}$ and construct a size n array b-SA s.t. b-$SA[i] = j$ iff the rank of $T[j : n]$ is i.*

Result 3.1. *Problem 3.1 can be solved either in $\Theta(gn)$-time and $\Theta(n)$-space where g is the number of runs of ones in b or in $\Theta(\frac{mn}{\epsilon \log m})$-time and $\Theta(m^{\epsilon}n)$-space where ϵ is any constant s.t. $0 < \epsilon < 1$.*

Since Problem 3.1 is a newly introduced problem, we assess Result 3.1 by comparing it against a naïve solution. A natural naïve solution for this problem is radix sort, i.e., sorting the suffixes by the i-th character for each $i \in \{1 \leq i \leq$

$m : b[i] = \mathtt{1}\}$ in descending order. It takes $\Theta(wn)$-time and $\Theta(n)$-space where w is the number of ones in b. Thus, the time complexity of the first (resp. second) solution in Result 3.1 is asymptotically smaller than that of the radix sort if $g = o(w)$ (resp. $\frac{m}{\log m} = o(w)$). By using these results, we can obtain bounds for the computation of the string kernel we introduce in Chapter 4 and an existing string kernel. We explain the detail of these bounds in Chapter 4.

We also consider the following problem and give the result below it (See Theorem 3.3 for formal statement.):

Problem 3.2. *Given a string T of length n, a binary string b of length m, sort the suffixes of T considering only i-th characters for i s.t. $b[i \mod m] = \mathtt{1}$ and construct a size n array b^*-SA s.t. b^*-$SA[i] = j$ iff the rank of $T[j : n]$ is i.*

Result 3.2. *Problem 3.2 can be solved in $\Theta(t(T, b)+n)$-time and $\Theta(s(T, b)+n)$-space where $t(T, b)$ (resp. $s(T, b)$) is the time (resp. space) needed to solve Problem 3.1.*

Problem 3.2 asks constructing the b-suffix array for the cases when the positions of wildcards in the pattern are periodic. Such periodic patterns arise in spaced seed-based filtering algorithms such as WABA [58] and PerM [20].

One can also solve Problem 3.2 naïvely by radix sort, i.e., sorting the suffixes by the i-th character for each $i \in \{1 \leq i \leq n : b[i \mod m] = \mathtt{1}]\}$ in descending order. It takes $\Theta(n^2w/m)$-time. Thus, the time complexity of the solution in Result 3.2 with the complexities of the first (resp. second) solution of Result 3.1 plugged in to t is asymptotically smaller than that of radix sort if $g = o(nw/m)$ (resp. $m^2/\log m = o(wn)$).

3.1.1 Related Work

As is mentioned above, the b-suffix array supports the search of patterns with wildcards in predetermined positions. More precisely, if the b-suffix array of a length n string S is given, the occurrences of a pattern P s.t. $P[i] \neq \mathtt{?}$ iff $b[i] = \mathtt{1}$ is enumerated in $O(w \log n + occ)$-time (or $O(w + \log n + occ)$-time if the b-height array (cf. Subsection 3.3.1) is also prepared) where w is the number of non-wildcard characters in P and occ is the number of occurrences. In this subsection, we explain the context in which such pattern matching problem arises. We also review the results of existing work on data structures that support search of patterns with wildcards.

Spaced Seed-based Homology Search Algorithms. In sequence homology search, one is given a database of sequences and a query sequence and the goal is to find all database elements that can be aligned to the query within a given cost.[1] Homology search is helpful to infer the functional or evolutionary relationships between biological sequences and have been studied intensively in bioinformatics community.

Modern algorithms for sequence homology search typified by BLAST [3] are based on filtering. These algorithms first extract short patterns called seeds from the query (See also Figure 3.1) and locate the occurrences of them in the

[1] For simplicity, we assume that the database is represented as a single string even if it is composed of multiple strings.

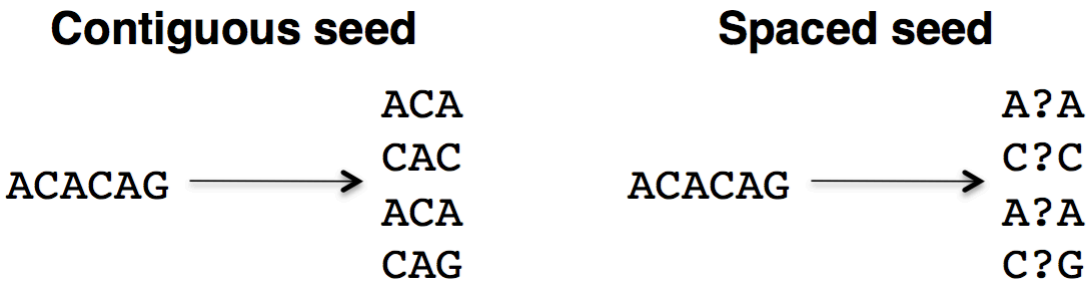


Figure 3.1: The extraction of seeds and spaced seeds. On the left, the set of seeds is {ACA, CAC, CAG}. The right half corresponds to the case when the wildcard position is represented by a binary string 101. The set of spaced seed is {A?A, C?C, C?G}.

database to select elements that are potentially similar to the query. Then, each of these potential matches is verified by sequence alignment. For example, in BLAST for nucleotide sequences, if query length is m, each of $m - 10$ query substrings of length 11 is used as a seed. Because filtering-based algorithms can fail to find some solutions, they are often evaluated by the sensitivity, i.e., the ratio of the number of solutions found to the number of all solutions.

While classical algorithms such as FASTA [106] and BLAST [3] use contiguous seeds, more modern algorithms such as PatternHunter [81], PerM [20] and ZOOM [77] use spaced seeds, which are the same as seeds except that characters of certain fixed positions are wildcards (See also Figure 3.1). It is known that the use of spaced seeds instead of contiguous seeds improves the filtering-based algorithm both in terms speed and sensitivity.

There are many possible combinations of wildcard positions. For example, if we represent the position of a wildcard (resp. non-wildcard) by 0 (resp. 1) and fix the length of the spaced seed to 5 and fix the number of wildcards to 2, there are 8 possible combinations, namely, 11100, 11010, 11001, 10110, 10101, 01110, 01101, 00111.[2] It is known that the performance gain of spaced seeds over contiguous seeds depends heavily on the choice of wildcard positions. Thus, modern algorithms based on spaced seed use some fixed set of wildcard positions that are carefully designed to achieve high sensitivity. For example, Ma et al. [81] recommended using positions $b_0 :=$ 111010010100110111 for nucleotide sequences. If we preprocess the database sequence and prepare the b-suffix array of it where b is the binary string representing the predetermined wildcard positions, the occurrences of a spaced seed can be located in $O(w \log n + occ)$-time (or $O(w + \log n + occ)$-time if we also construct the b-height array).

Text indices for patterns with wildcards. Pattern matching problems allowing errors such as mismatches, insertions and deletions have also been studied in string algorithm community. In particular, we review the results of existing text indices that support the search of patterns containing wildcards.

We first introduce some notations. In the rest of this subsection, we denote the length of the text and the pattern (including wildcards) by n and m respectively. P is the pattern and $occ(P)$ is the number of the occurrences of P in the

[2]Some patterns listed here are essentially the same. For example, 11100, 01110 and 00111 are the same if we ignore zeros not surrounded by ones.

text. We denote the number of wildcards in the pattern by d and the number of contiguous groups of wildcards in the pattern by h. In other words, the pattern P can be decomposed as $P_1 \circ \mathtt{?}^{d_1} \circ P_2 \circ \mathtt{?}^{d_2} \circ P_3 \circ \cdots \circ P_h \circ \mathtt{?}^{d_h} \circ P_{h+1}$ where c^k denotes character c repeated for k times, d_i's are positive integers satisfying $\sum_{i=1}^{h} d_i = d$ and each of P_i's is a substrings of P that does not include any wildcard. In the context of spaced seed search, $d_1, d_2, \ldots, d_h$ and $|P_1|, |P_2|, \ldots, |P_{h+1}|$ are all fixed and known. In such cases, the b-suffix array (in combination with the b-height array) takes $O(n)$-size and supports finding occurrences of patterns in the text in $O(m + \log n + occ(P))$-time.

Cole et al. [21] designed a text index that supports the search of patterns with up to k wildcards (i.e., $d \leq k$) where k is a number that is chosen at the time of index construction. Their index structure takes $O(n \log^k n)$-space and supports the search of patterns with up to k wildcards in $O(m+2^d \log\log n+occ(P))$-time. Bille et al. [12] gave a data structure that takes $O(\sigma^{k^2} n \log^k \log n)$-space and supports the search of patterns with up to k wildcards in $O(m + occ(P))$-time. They also proposed an $O(n)$-space data structure that supports the search of patterns with unbounded number of wildcards in $O(m + \sigma^d \log\log n + occ(P))$-time. Iliopoulos and Rahman [48] proposed an $O(n)$-space index that can be built in $O(n)$-time and supports search of patterns in $O(m + \alpha)$-time where $\alpha := \sum_{1 \leq i \leq h+1} occ(P_i)$. Lam et al. [67] improved this bound for the search time to $O(m + h\beta)$ with the same space and preprocessing time where $\beta := \min_{1 \leq i \leq h+1} occ(P_i)$. Both α and β can be $\Theta(n)$ even if $occ(P) = 0$. These data structures cannot be compared directly with the b-suffix array because they do not assume that the wildcard positions in the pattern are fixed and known at the time of index construction.

There is also another type of text index that is more similar to the b-suffix array. The gapped suffix array [24] is a generalization of the suffix array and a special case of the b-suffix array. More specifically, the (g_0, g_1)-gapped suffix array of a text is the b-suffix array of the case when $b = \mathtt{1}^{g_0} \circ \mathtt{0}^{g_1} \circ \mathtt{1}^{n-g_0-g_1}$. It supports the search of patterns with wildcards in g_0+1-th to g_0+g_1-th positions.

3.2 Notations and Definitions

Here, we introduce notations used in the rest of this chapter.

Let T be a length n string. We denote $T[i : n]$ by T_i for $i \leq n$.

The wildcard `?` is a special character not included in Σ. We denote $\Sigma \cup \{\mathtt{?}\}$ by Γ. We denote the set of length k strings on Γ by Γ^k and the set of finite length strings on Γ by Γ^*. A pattern $P \in \Gamma^m$ is said to occur at position i of T if $P[j] = T[i+j-1]$ for any j from 1 to m such that $P[j] \neq \mathtt{?}$. Let b be a binary string (a string on alphabet $\{\mathtt{0}, \mathtt{1}\}$) of length m. We denote $\{1 \leq i \leq m | b[i] = \mathtt{1}\}$ (resp. $\{1 \leq i \leq m | b[i] = \mathtt{0}\}$) by $\mathbf{1}_b$ (resp. $\mathbf{0}_b$). We call the cardinality of $\mathbf{1}_b$ the weight of b and denote it by $w(b)$. Let $1 \leq i_1 < i_2 < \cdots < i_w \leq m$ be the elements of $\mathbf{1}_b$ sorted in ascending order. $mask_b(T)$ is defined to be $T[i_1] \circ T[i_2] \circ \cdots \circ T[i_{w(b)}]$ if $i_{w(b)} \leq n$ and $T[i_1] \circ T[i_2] \circ \cdots \circ T[i_j]$ if $i_j \leq n < i_{j+1}$ for some j between 1 and $w(b) - 1$. For strings S and T, the expression $S \preceq_b T$ means $mask_b(S)$ is lexicographically less than or equal to $mask_b(T)$. The relation $\preceq_b$ is a preorder and we call it the b-lexicographic order. The expression $S =_b T$ means $S \preceq_b T$ and $T \preceq_b S$.

i	SA[i]	T[SA[i]:n]
1	4	AAC
2	5	AC
3	1	ACCAAC
4	6	C
5	3	CAAC
6	2	CCAAC

i	101-SA[i]	T[101-SA[i]:n]
1	5	AC
2	1	ACCAAC
3	4	AAC
4	6	C
5	2	CCAAC
6	3	CAAC

Figure 3.2: The suffix array and 101-suffix array of "ACCAAC".

When c is a character or a string, we denote c repeated for k times by c^k. For a string T of length n and a character c, we say $T[i:j]$ is a run of c if a) $T[i:j] = c^{j-i+1}$ and b) $i = 1$ or $T[i-1] \neq c$ and c) $j = n$ or $T[j+1] \neq c$, then we say $T[i:j]$.

Let T be a string of length n. For an integer $h > 0$ and $i \in \{1, 2, \ldots, n\}$, the h-rank of i is the rank of i when sorted by $T[i : i+h-1]$ and we denote it by $r_h(i)$. Note that $r_h(SA[i]) \leq r_h(SA[i+1])$ for any $i \in \{1, 2, \ldots, n-1\}$. For a binary string b and $i \in \{1, 2, \ldots, n\}$, the b-rank of i is the rank of i when sorted by $mask_b(T[i:n])$ and we denote it by $r_b(i)$.

3.3 b-Suffix Array

3.3.1 Definition

Definition 3.1. *The b-suffix array of a string T of length n, which we denote by b-SA_T, is an array containing $1, 2, \ldots, n$ sorted by the b-lexicographic order of $T_1, T_2, \ldots, T_n$ where the tie is broken by i.*

An example of the b-suffix array is shown in Figure 3.2.

Definition 3.2. *The b-height array of a string T, which we denote by b-HGT_T, is an array of size n such that*

$$b\text{-}HGT_T[i] = \begin{cases} 0 & \text{if } i = 1 \\ |lcp(mask_b(T_{b\text{-}SA[i-1]}), mask_b(T_{b\text{-}SA[i]}))| & \text{otherwise.} \end{cases}$$

Because we do not consider multiple strings, we omit the subscript T of SA_T, b-SA_T, HGT_T and b-HGT_T in the rest of this chapter.

3.3.2 Search Method

Let T be a string of length n, b be a binary string of weight w and b-SA be the b-suffix array of T. Let P be a pattern of length $p \leq |b|$ s.t. $P[i] = ?$ for $i \in \mathbf{0}_b$ and $P[i] \in \Sigma$ for $i \in \mathbf{1}_b$. Let $left(P)$ be $\min\{i : P \preceq_b T_{b\text{-}SA[i]}\}$ if $\{i : P \preceq_b T_{b\text{-}SA[i]}\}$ is non-empty and $n+1$ otherwise. Also, let $right(P)$ be $\max\{i : T_{b\text{-}SA[i]} \preceq P\}$ if $\{i : T_{b\text{-}SA[i]} \preceq P\}$ is non-empty and 0 otherwise. The

value $left(P)$ can be located by Algorithm 3.1. Each time we compare P and a suffix according to $\preceq_b$, it takes $O(w)$-time. Thus, each of the two cases from line 1 to line 2 and from line 3 to line 4 takes $O(w)$-time and the while loop from line 7 to line 12 takes $O(w \log n)$-time. Thus, in total, this algorithm takes $O(w \log n)$-time. The value $right(P)$ can be located similarly in $O(w \log n)$-time. Because $P =_b T[i : i + p - 1]$ iff $i = b\text{-}SA[j]$ for some $j \in [left(P), right(P)]$, all occurrences of P in T can be found in $O(w \log n + occ)$-time where occ is the number of occurrences.

Algorithm 3.1 $O(w \log n)$-time algorithm to find $left(P)$

Ensure: $left = left(P)$
1: **if** $P \preceq_b T_{b\text{-}SA[1]}$ **then**
2: $\quad left \leftarrow 1$
3: **else if** $P \succ_b T_{b\text{-}SA[n]}$ **then**
4: $\quad left \leftarrow n + 1$
5: **else**
6: $\quad (L, R) \leftarrow (1, n)$
7: $\quad$ **while** $R - L > 1$ **do**
8: $\quad\quad M \leftarrow \lfloor (L + R)/2 \rfloor$
9: $\quad\quad$ **if** $P \preceq_b T_{b\text{-}SA[M]}$ **then**
10: $\quad\quad\quad R \leftarrow M$
11: $\quad\quad$ **else**
12: $\quad\quad\quad L \leftarrow M$
13: $\quad left \leftarrow R$

If we prepare the b-height array of T and the range minimum data structure (cf. Theorem 2.1), for $b\text{-}HGT$, then $left(P)$ can be found in $O(w+\log n)$-time by the same technique as the $O(m+\log n)$-time search algorithm for the suffix array. Let $1 \leq i_1 < i_2 < \cdots < i_w \leq |b|$ be the indices of ones of b. We maintain $\ell := |lcp(mask_b(P), mask_b(T_{b\text{-}SA[L]}))|$ (resp. $r := |lcp(mask_b(P), mask_b(T_{b\text{-}SA[R]}))|$) where L (resp. R) is the index of the left (resp. right) boundary of the interval we successively halve in binary search. If $\ell \geq r$, we compare ℓ and $llcp := |lcp(mask_b(T_{b\text{-}SA[L]}), mask_b(T_{b\text{-}SA[M]}))|$ where $M = \lfloor (L + R)/2 \rfloor$. $llcp$ can be computed in constant time by the range minimum data structure because $llcp = \min\{b\text{-}HGT[i] : L \leq i \leq M\}$. If $llcp > \ell$, then $left(P)$ is in $[M, R]$. If $llcp < \ell$, then $left(P)$ is in $[L, M]$. If $llcp = \ell$, then we need to compare $mask_b(P)$ and $mask_b(T_{b\text{-}SA[M]})$ to determine if $left(P)$ is in $[M, R]$ or $[L, M]$. However, $\ell = |lcp(mask_b(P), mask_b(T_{b\text{-}SA[L]}))| = |lcp(mask_b(T_{b\text{-}SA[L]}), mask_b(T_{b\text{-}SA[M]}))|$, $mask_b(P)$ and $mask_b(T_{b\text{-}SA[M]})$ share the first ℓ characters and thus, it suffices to compare them from $\ell + 1$-th character, $\ell + 2$-th character and so on. If $mask_b(P)[j] = P[i_j]$ is matched to some character of $T_{b\text{-}SA[M]}$ in this comparison, then, $P[i_j]$ is never used in the comparison made in future. Thus, if $\ell \geq r$, either the number of characters in $mask_b(P)$ that can be used for comparison decreases or the interval $[L, R]$ is halved in constant time. The same thing is true for the case when $\ell < r$. Therefore, this algorithm takes $O(w + \log n)$-time. $right(P)$ can also be found in $O(w + \log n)$-time and thus, all occurrences of P can be listed in $O(w + \log n + occ)$-time.

Algorithm 3.2 $O(w + \log n)$-time algorithm to find $left(P)$

Ensure: $left = left(P)$

$\ell \leftarrow lcp(mask_b(T_{b\text{-}SA[1]}), mask_b(P))$
$r \leftarrow lcp(mask_b(T_{b\text{-}SA[n]}), mask_b(P))$
if $\ell = w$ or $P[i_{\ell+1}] \leq T[b\text{-}SA[1] + i_{\ell+1} - 1]$ **then**
    $left \leftarrow 1$
else if $r < w$ and $P[i_{r+1}] > T[b\text{-}SA[n] + i_{r+1} - 1]$ **then**
    $left \leftarrow n + 1$
else
    $(L, R) \leftarrow (1, n)$
    while $R - L > 1$ **do**
        $M \leftarrow \lfloor (L + R)/2 \rfloor$
        if $\ell \geq r$ **then**
            $llcp \leftarrow lcp(mask_b(T_{b\text{-}SA[L]}), mask_b(T_{b\text{-}SA[M]}))$
            if $llcp \geq \ell$ **then**
                $mid \leftarrow \ell + lcp(mask_{b[i_{\ell+1}:m]}(T_{b\text{-}SA[M]+i_{\ell+1}-1}), mask_{b[i_{\ell+1}:m]}(P_{i_{\ell+1}}))$
            else
                $mid \leftarrow llcp$
        else
            $rlcp \leftarrow lcp(mask_b(T_{b\text{-}SA[R]}), mask_b(T_{b\text{-}SA[M]}))$
            if $rlcp \geq r$ **then**
                $mid \leftarrow r + lcp(mask_{b[i_{r+1}:m]}(T_{b\text{-}SA[M]+i_{r+1}-1}), mask_{b[i_{r+1}:m]}(P_{i_{r+1}}))$
            else
                $mid \leftarrow rlcp$
        if $mid = w$ or $P[i_{mid+1}] \leq T[b\text{-}SA[M] + i_{mid+1} - 1]$ **then**
            $(R, r) \leftarrow (M, mid)$
        else
            $(L, \ell) \leftarrow (M, mid)$
    $left \leftarrow R$

3.3.3 Construction of the b-Suffix Array for General b

Basic Idea

Let T be a string of length n and b be a binary string of weight w. As is mentioned in Section 3.1, one can construct the $b\text{-}SA_T$ by radix sort, i.e., repeatedly sorting $i \in \{1, 2, \ldots, n\}$ by $T[i + j - 1]$ while j runs through all $i \in \mathbf{1}_b$ in descending order. It takes $\Theta(wn)$-time and $\Theta(n)$-space. In the following of this subsection, we show non-trivial algorithms to construct the b-suffix arrays. The key idea is to modify the radix sort so that each bucket sort can take into account multiple characters instead of single character.

$O(gn)$-time, $O(n)$-space Algorithm

We first prove the following theorem:

Theorem 3.1. *Given a text T of length n and a wildcard position b of length m, the b-suffix array $b\text{-}SA$ and the b-height array $b\text{-}HGT$ can be constructed in $O(gn)$-time and $O(n)$-space where g is the number of runs of ones in b.*

The binary string b can be divided into runs as $0^{\ell'_1} \circ 1^{\ell_1} \circ 0^{\ell'_1} \circ \cdots \circ 0^{\ell'_g} \circ 1^{\ell_g} \circ 0^{\ell'_{g+1}}$. Let s_j be the index of the first character of the j-th run of one. If we can stably sort $1, 2, \ldots, n$ using $T_i[s_j : s_j + \ell_j - 1]$ as the key of i in $O(n)$-time for any $j \in \{1, 2, \ldots, g\}$, then we can construct the b-suffix array in $O(gn)$-time by repeatedly applying this stable sort for all j from g to 1. Thus, it suffices to show that each of these g stable sort can be done in $O(n)$-time.

Because $T_i[s_j : s_j + \ell_j - 1] = T_{i+s_j-1}[1 : \ell_j]$, sorting by $T_i[s_j : s_j + \ell_j - 1]$ is equivalent to sorting by $T_{i+s_j-1}[1 : \ell_j]$. Also, the rank of i in this sorting is $r_{\ell_j}(i + s_j - 1)$ because $T_{i+s_j-1}[1 : \ell_j]$ is a length ℓ_j prefix of T_{i+s_j-1}. If $HGT[i] < h$, the h-rank of $T_{SA[i]}$ is i. Otherwise, $T_{SA[i]}[1 : h] = T_{SA[i-1]}[1 : h]$ and $r_h(SA[i]) = r_h(SA[i-1])$. Therefore, h-ranks for all $i \in \{1, 2, \ldots, n\}$ can be calculated in $O(n)$-time by Algorithm 3.3.

Algorithm 3.3 $O(n)$-time calculation of h-ranks

Require: the suffix array SA, the height array HGT
Ensure: $R[i]$ is the h-rank of i

```
1: R[SA[1]] ← 1
2: for i = 2 to n do
3:     if HGT[i] < h then
4:         j ← i
5:     R[SA[i]] ← j
```

Once ℓ_j-ranks are obtained, we can plug it into the standard radix sort. Suppose we already have a list derived by iteratively sorting $1, 2, \ldots, n$ using $T_i[s_j : s_j + \ell_j - 1]$ as the key of i for all j from g to $j_0 + 1$. We can further stably sort this list using $T_i[s_{j_0} : s_{j_0} + \ell_{j_0} - 1]$ as the key of i as follows.

We prepare n queues $Q_1, Q_2, \ldots, Q_n$. Scanning the list, we push each element i into $Q_{r_{\ell_j}(i+s_j-1)}$. We prepare a new empty list and for all i from 1 to n, we pop out the elements of Q_i and append it to the new list. Then, the new list contains what is needed. Therefore, we can construct the b-suffix array by applying this procedure for all j from g to 1. The pseudocode of the whole algorithm is shown in Algorithm 3.4. The calculation of R in line 5 is done by calling Algorithm 3.3 as a subroutine.

Algorithm 3.4 The $O(gn)$-time construction of the b-suffix array

Require: text T of length n, binary string b of length m
Ensure: $b\text{-}SA$ is the b-suffix array of T
1: **for** i from 1 to n **do**
2: $\quad b\text{-}SA[i] \leftarrow i$
3: construct the suffix array SA and the height array HGT
4: **for** $j = g$ to 1 **do**
5: $\quad$ construct R s.t. $R[i] = r_{\ell_j}(i)$
6: $\quad$ **for** $i = 1$ to n **do**
7: $\quad\quad$ push $b\text{-}SA[i]$ to queue $Q_{R[b\text{-}SA[i]+s_j-1]}$
8: $\quad i = 1$
9: $\quad$ **for** $k = 1$ to n **do**
10: $\quad\quad$ **while** Q_k is not empty **do**
11: $\quad\quad\quad b\text{-}SA[i] \leftarrow$ pop Q_k
12: $\quad\quad\quad i \leftarrow i + 1$

As shown in Subsection 2.3.2, the suffix array can be constructed in $O(n)$-time [55, 61, 99] and the height array can be constructed from the suffix array in $O(n)$-time [56]. Thus, line 3 of Algorithm 3.4 takes $O(n)$-time. The procedures in the for loop from line 4 to line 12, including the calculation of R in line 6 take $O(n)$-time. Therefore, the total time complexity is $O(gn)$. We use constant number of $O(n)$-size arrays, that is, $SA, HGT, b\text{-}SA, R$. Also we use n queues and the total size of them is also $O(n)$ because the total number of elements in them are at most n. Thus the total space complexity is $O(n)$.

Next, we prove the bound for the b-height array by showing how to construct the b-height array from the b-suffix array in $O(gn)$-time. One can construct the b-height array by computing $lcp(mask_b(T_{b\text{-}SA[i-1]}), mask_b(T_{b\text{-}SA[i]}))$ for all i from 2 to n. If we compute $lcp(mask_b(T_{b\text{-}SA[i-1]}), mask_b(T_{b\text{-}SA[i]}))$ by comparing $T_{b\text{-}SA[i-1]}[j]$ and $T_{b\text{-}SA[i]}[j]$ for each $j \in \mathbf{1}_b$ in ascending order until the first mismatch is found, it takes $\Theta(w)$-time in the worst case and thus, this method of height array construction takes $\Theta(wn)$-time. As is mentioned in subsection 2.3.2, by constructing the data structure for range minimum query [9] over the height array, one can calculate $|lcp(T_i, T_j)|$ for any i and j in constant time. The data structure for range minimum query can be constructed in $O(n)$-time. By using this technique, we calculate $|lcp(T_{b\text{-}SA[i-1]+s_j}, T_{b\text{-}SA[i]+s_j})|$ for each j from 1 to g. If the lcp value is greater than or equal to ℓ_j, then $T_{b\text{-}SA[i-1]}[s_j : s_j + \ell_j - 1] = T_{b\text{-}SA[i]}[s_j : s_j + \ell_j - 1]$ and thus, we proceed to the comparison between $T_{b\text{-}SA[i-1]}[s_{j+1} : s_{j+1}+\ell_{j+1}-1] = T_{b\text{-}SA[i]}[s_{j+1} : s_{j+1}+\ell_{j+1}-1]$. Otherwise, $b\text{-}HGT[i]$ is $|lcp(T_{b\text{-}SA[i-1]+s_j}, T_{b\text{-}SA[i]+s_j})|$ plus the number of characters matched so far. Either of these cases takes constant time and thus, the comparison of each pair of suffixes takes $O(g)$-time. Overall, the b-height array can be constructed in $O(gn)$-time. The pseudocode is presented in Algorithm 3.5. We use constant number of arrays of size $O(n)$, namely $b\text{-}SA$, HGT and $b\text{-}HGT$, and the auxiliary data structure for constant time longest common prefix query, which also takes $O(n)$-space. Therefore, the algorithm uses $O(n)$-space.

Algorithm 3.5 The $O(gn)$-time construction of the b-height array

Require: the b-Suffix array b-SA, the height array HGT
Ensure: b-HGT is the b-height array of T
1: initialize all elements of b-HGT by 0
2: preprocess HGT for constant time lcp computation
3: **for** $i = 2$ to n **do**
4: **for** $j = 1$ to g **do**
5: $h \leftarrow lcp(T_{b\text{-}SA[i]+s_j-1}, T_{b\text{-}SA[i-1]+s_j-1})$
6: $b\text{-}HGT[i] \leftarrow b\text{-}HGT[i] + \min\{h, \ell_j\}$
7: **if** $h < \ell_j$ **then**
8: break

$O(\frac{mn}{\epsilon \log m})$-time and $O(m^\epsilon n)$-space Algorithm

The time complexity of Algorithm 3.4 depends on g, a variable that depends on the problem. Here, we show the construction algorithm whose time and space bounds involve a configurable variable. More precisely we prove the following theorem:

Theorem 3.2. *Given a text T of length n and a binary string b of length m, the b-suffix array b-SA can be constructed in $O(\frac{mn}{\epsilon \log m})$-time and $O(m^\epsilon n)$-space. where ϵ can be any constant s.t. $0 < \epsilon < 1$.*

The strategy of this algorithm is also to modify radix sort in such a way that multiple characters can be taken into account in single bucket sort. While the number of characters taken into account in one bucket sort in Algorithm 3.4 varies among different of instances of bucket sort, in the current algorithm, we fix it to some constant $v \leq m$. For brevity, we assume v is a divisor of m. Let $b_i := b[(i-1)v + 1 : iv]$ for $1 \leq i \leq m/v$. Starting from an array containing $1, 2, \ldots, n$, we iteratively update $0^{jv} \circ b[jv{+}1 : m]$-$SA$ to $0^{(j-1)v} \circ b[(j{-}1)v{+}1 : m]$-$SA$ until we obtain $b[1 : m]$-$SA = b$-SA. In each update, elements of $0^{jv} \circ b[jv{+}1 : m]$-$SA$ are stably sorted using $mask_{0^{(j-1)v} \circ b_j}(T_i)$ as the key of i. Note that $mask_{0^{(j-1)v} \circ b_j}(T_i) = mask_{b_j}(T_{i+(j-1)v})$.

This update procedure can be done almost the same way as Algorithm 3.4. Suppose we already have a list of $1, 2, \ldots, n$ sorted by $mask_{0^{jv} \circ b[jv+1:m]}(T_i)$ as the key of i. We can stably sort this list as follows. We prepare n queues $Q_1, Q_2, \ldots, Q_n$. Scanning the list, we push each element i into $Q_{r_{b_j}(i+(j-1)v)}$. (We show how to calculate b_j-ranks shortly.) We prepare a new empty list and, for all i from 1 to n, we pop out the elements of Q_i and append it to the new list. Then, the new list is sorted by $mask_{0^{(j-1)v} \circ b[(j-1)v+1:m]}(T_i)$.

In the procedure above, we need to calculate b_j-rank $r_{b_j}(i + (j-1)v)$ to determine which queue to push element i into. In order to do this, we prepare the table of b_j-ranks for every possible length v binary string b. $b \circ b'$-rank can be derived from b-rank and b'-rank in $O(n)$-time by two digits radix sort. Therefore the b-ranks for all b of length $k+1$ can be derived from b-ranks for b of length k and length 1. There are $2^{v+1} - 1 = O(2^v)$ binary strings of length at most v, thus the preprocessing takes $O(2^v n)$-time. The construction except the preprocessing takes $O(mn/v)$-time because each update from $0^{jv} \circ b[jv+1 : m]$-SA to $0^{(j-1)v} \circ b[(j{-}1)v{+}1 : m]b$-$SA$ takes $O(n)$-time and there are m/v updates.

Thus, the total time complexity is $O(2^v n + mn/v)$. It takes $O(n)$-space to store the values of b-ranks for each b of length v and thus, the total space complexity is $O(2^v n)$.

v is a parameter that we can choose. In particular, to minimize the total time complexity, it is desirable to balance the preprocessing time and the time for constructing b-SA. One way to do it is to let v to be $\epsilon \log m$ where ϵ is a constant s.t. $0 < \epsilon < 1$.[3] In this case, the time complexity is $O(\frac{mn}{\epsilon \log m})$ and the space complexity is $O(m^\epsilon n)$, which proves Theorem 3.2.

3.3.4 Construction of the b-Suffix Array by Sorting from Forward

The algorithms from the previous subsection construct the b-suffix array by successive applications of bucket sort and each bucket sort takes $\Theta(n)$-time. However, in the applications of radix sort, there are cases when only a few significant digits are enough to uniquely determine the final result. For example, we can conclude that the 3-digit tuples $t_1 = (1, 5, 8), t_2 = (4, 6, 7), t_3 = (2, 7, 6), t_4 = (3, 8, 5)$ are sorted as t_1, t_3, t_4, t_2 even if we do not read the second and third digits of these tuples because the first, i.e., the most significant, digits are enough to determine the sorting. In this subsection, we show how to modify the $O(gn)$-time algorithm in the previous subsections so that more significant digits are treated before less significant digits although, in the worst case, the modified algorithm takes $O(gn \log n)$-time.

First, we explain MSD radix sort. MSD radix sort is a variant of radix sort where more significant digits are treated before less significant digits. MSD means 'most significant digit'.[4] We explain it in the case when sorted elements are strings. In string sorting, each digit corresponds to a character. In MSD radix sort, the elements are sorted at first using the first characters as the keys. This procedure partitions the strings into groups in such a way that the elements of each group share the same first character. Each of these groups are sorted recursively. At the depth j recursion, the j-th character is used as the key. When the group to be sorted consists of only one element, the group is already sorted and thus, recursion stops.

Now we show an MSD radix sort-based construction algorithm for the b-suffix array. We start from an array containing $(1, 2, \ldots, n)$ and the goal is to sort these numbers using the g-tuple $(T_i[s_1 : s_1 + \ell_1 - 1], T_i[s_2 : s_2 + \ell_2 - 1], \ldots, T_i[s_g : s_g + \ell_g - 1])$ as the key for i. We do this by modifying MSD radix sort so that at depth j recursion, $T_i[s_j : s_j + \ell_j - 1]$ is used as the key for i. This is equivalent to sort using $r_{\ell_j}(i + s_j - 1)$ as the key for i because $T_i[s_j : s_j + \ell_j - 1] = T_{i+s_j-1}[1 : \ell_j]$. Suppose we already completed sorting by $(T_i[s_1 : s_1 + \ell_1 - 1], \ldots, (T_i[s_{j_1} : s_{j-1} + \ell_1 - 1]$ and need to sort n' elements from *start*-th by $T_i[s_j : s_j + \ell_1 - 1]$. Similar to Algorithm 3.4, we do this by pushing each element i into queue Q_r where $r = r_{\ell_j}(i + s_j - 1)$ and popping all elements of Q_r for smaller r's to larger r's.

However, if we implement this procedure by enumerating all possibly empty

[3] This choice of v does not balance the time complexity of preprocessing and b-SA construction exactly. We chose this setting because the optimal v cannot be expressed as an elementary function.

[4] The standard radix sort is called LSD radix sort when it is necessary to clarify which type of radix sort it means. LSD means 'least significant digit'.

queues $Q_1, Q_2, \ldots, Q_n$ as Algorithm 3.4, it would take $\Theta(n)$-time independent of the number of elements to be sorted, i.e., n'. Also, if we calculate $r_{\ell_j}(i+s_j-1)$ by Algorithm 3.3, it also takes $\Theta(n)$-time. To avoid this $\Theta(n)$-time cost, we maintain a dynamic set S of non-empty queues. There can be at most n' non-empty queues. As for $r_{\ell_j}(i+s_j-1)$, we calculate it as follows. We construct the range minimum query data structure (cf. Theorem 2.1) for the height array. As is mentioned in Subsection 2.3.2, the range minimum query data structure can be constructed in $O(n)$-time and given i and j $(i \leq j)$, the index k such that $HGT[k]$ is the minimum among $HGT[i], HGT[i+1], \ldots, HGT[j]$ can be found in $O(1)$-time. We also prepare a size n array R that satisfies $R[SA[i]] = i$ for all i from 1 to n at the beginning. Obviously, R can be constructed in $O(n)$-time from SA. Note that $r_h(i) = max\{j \leq R[i] : HGT[j] < h\}$. To calculate $r_h(i)$, we first calculate $\min\{HGT[k] : 1 \leq k \leq R[i]\}$. If it is greater than or equal to h, then $r_h(i)$ is 1. Otherwise, there is at least one k s.t. $HGT[k] < h$ in $[1, R[i]]$. In general, we can find the maximum k in $[left, right]$ s.t. $HGT[k] < h$ (if there is at least one such k) as follows. If $left = right$, then $left$ is the answer. Let $mid := \lfloor left + right/2 \rfloor$ and calculate $\min_k\{HGT[k] : left \leq k \leq mid\}$ and $\min_k\{HGT[k] : mid + 1 \leq k \leq right\}$. At least one of them must be smaller than h. If $\min_k\{HGT[k] : mid + 1 \leq k \leq right\} < h$, recursively find the maximum k in $[mid+1, right]$ s.t. $HGT[k] < h$. Otherwise, recursively find the maximum k in $[left, mid]$ s.t. $HGT[k] < h$. This procedure takes $O(\log n)$-time.

The pseudocode of the construction algorithm is shown in Algorithm 3.6 and Algorithm 3.7. As is mentioned above, the computation of ℓ_j-rank in line 5 takes $O(\log n)$-time. Insertion of element r into the dynamic set S in line 7 takes $O(\log n')$-time if we implement S by the self-balancing binary search tree. Therefore, the for loop from line 4 to line 8 takes $O(n' \log n)$-time. The for loop from line 11 to line 18 takes $O(n')$-time. Overall, each recursive call takes $O(n' \log n)$-time. In the worst case, there can be n elements sorted in depth j recursions and thus, the depth j recursive calls take $O(n \log n)$-time in total. Therefore, it takes $O(gn \log n)$-time in total.

Algorithm 3.6 Construction of the b-suffix array from forward

1: Construct SA, HGT, R and the range minimum data structure for HGT
2: Initialize b-SA by $(1, 2, \ldots, n)$
3: $RecursiveSort(1, n, 1)$

3.3.5 Construction of the b-Suffix Array for Periodic b

In this section, we prove the following theorem.

Theorem 3.3. *Given a text T of length n, a binary string b of length p, the b-suffix array b-SA and an array b-RSA s.t. b-$RSA[i] = r_b(i)$, b^*-SA is obtained in $\Theta(n)$-time and $\Theta(n)$-space where $b^* = b^{\lceil n/p \rceil}$.*

The inequality $T_i \preceq_{b^*} T_j$ holds iff a) $T_i[1:p] \preceq_b T_j[1:p]$ and $T_j[1:p] \not\preceq_b T_i[1:p]$ $(\Leftrightarrow r_b(i) < r_b(j))$ or b) $T_i[1:p] =_b T_j[1:p]$ $(\Leftrightarrow r_b(i) = r_b(j))$ and $T_{i+p} \preceq_{b^*} T_{j+p}$. Thus, $\preceq_{b*}$ on $\{T_i\}_i$ is equivalent to the lexicographic order on strings $\{r_b(i) \circ r_b(i+p) \circ \cdots \circ r_b(i+\lceil (n-i)/p \rceil p)\}_i$. To sort $\{T_i\}_i$ by $\preceq_{b^*}$, it suffices

Algorithm 3.7 *RecursiveSort*$(start, n', j)$

```
1: if n' = 1 or j > g then
2:     return
3: for i ← start to start + n' − 1 do
4:     r ← r_{ℓ_j}(bSA[i] + s_j − 1)
5:     push bSA[i] to Q_r
6:     add r to S if r ∉ S
7: i ← 1, k ← 1
8: Boundary[k] ← 1
9: for r ∈ S in ascending order do
10:    while Q_r is not empty do
11:        bSA[i] ← pop Q_r
12:        i ← i + 1
13:    k ← k + 1
14:    Boundary[k] ← i
15: for i ← 1 to k − 1 do
16:    RecursiveSort(B[i], B[i + 1] − B[i], j + 1)
```

to sort $\{rs(i)\}_i$ by lexicographic order where $rs(i) := r_b(i) \circ r_b(i+p) \circ \cdots \circ r_b(i + \lceil (n-i)/p \rceil p)$. For i s.t. $1 \leq i \leq p$ and $0 \leq k \leq \lceil (n-i)/p \rceil$, $rs(i+kp)$ is a suffix of $rs(i)$. We consider the string $rs := rs(1) \circ \mathtt{0} \circ rs(2) \circ \mathtt{0} \circ \cdots \circ \mathtt{0} \circ rs(p)$. The set $\{rs(i)\}_i$ can be sorted by sorting the suffixes of rs. It is easy to ignore the suffixes starting from `0` because they gather to the head when sorted. Because the comparison of two suffixes ends before or at the time when either of them reaches the end, comparison of corresponding $rs(i)$'s is not affected by inserting `0`s.

The running time is dominated by the suffix sorting of rs, which can be done in $\Theta(n)$-time. Therefore, the total time complexity is $\Theta(n)$. Space complexity is also $\Theta(n)$. Therefore Theorem 3.3 follows. The pseudocode of the algorithm is presented in Algorithm 3.8.

Algorithm 3.8 Construction of the b-suffix array for periodic b

Require: b-suffix array b-SA, b-rank b-RSA, assume p evenly divides n
Ensure: b^*-SA is the b^*-suffix array

```
1: construct rs = rs(1) ∘ 0 ∘ rs(2) ∘ 0 ∘ ··· ∘ 0 ∘ rs(p)
2: construct the suffix array of rs SA1
3: n' ← n/p
4: for i = p to n + p − 1 do
5:     j ← SA1[i]
6:     q ← ⌊j/(n' + 1)⌋
7:     r ← j mod (n' + 1)
8:     b*-SA[i − p + 1] ← q + 1 + pr
```

3.4 Discussion

In this chapter we introduced the b-suffix array, a variant of the suffix array. The main motivation was the application to the computation of the kernel function introduced in Chapter 4 (This application is described in Chapter 4.) Aside from the application to kernel function computation, the b-suffix array, as a text index, supports searching patterns with wildcards in predetermined positions within time logarithmic to the size of the text. Such pattern matching problem arises in the context of spaced seed-based homology search. The technical contributions in this chapter were the construction algorithms of the b-suffix array. The naïve construction algorithm by conventional radix sort takes $O(wn)$-time where w is the number of ones in b. We proposed an $O(gn)$-time, $O(n)$-space algorithm and $O(mn/\epsilon \log m)$-time, $O(m^\epsilon n)$-space algorithm to construct the b-suffix array where g is the number of runs of one in b and ϵ is any number between 0 and 1. The algorithms above are based on radix sort. Although it does not improve the worst-case time bound, we showed how to construct the b-suffix array by applying the techniques of MSD radix sort instead of conventional LSD radix sort. We also gave an $O(gn)$-time algorithm to construct the b-height array from the b-suffix array. Last, we proposed an algorithm to construct the b^*-suffix array from b-suffix array and b-ranks in $O(n)$-time and $O(n)$-space.

The proposed data structure is just a size n array containing $1, 2, \ldots, n$ in a specific order and very easy to implement. We use this data structure in Chapter 4 in kernel function computation.

As a text index for spaced seed search, an interesting direction of research is to design compact text indices. In data structure community, the technology to compress data structures in such a way that operations can be done without decompressing have been vigorously studied since early 2000s and is called succinct data structure. This technology have been successfully applied to text indices for searching contiguous patterns both in terms of theory and practice. For example, when the length of the text is n, a data structure called FM-index takes $\Theta(n)$ bits in the worst case or $o(n)$ bits if the text is compressible while the suffix array takes $\Theta(n \log n)$ bits. The FM-index supports finding all occurrences of a pattern of length m in $O(m+occ \log^\epsilon n)$-time [26]. In practice, the FM-index occupies $3n$ bits while the suffix array takes $32n$ bits [44]. Though the search time of the FM-index is larger than that of the suffix array in 2 or 3 orders of magnitude [44], it was successfully applied to DNA mapping [74, 69] and genome assembly [123]. On the other hand, there have been relatively few work on the compression of text indices for non-contiguous patterns. Recently, Gagie et al. proposed a method to compress multiple b-suffix arrays for different b's by applying the techniques of succinct data structure [29]. This is a promising result because multiple seed, the use of many spaced seeds corresponding to different b's, is known to be a powerful method to increases the sensitivity of homology search [75]. As is mentioned by Gagie et al. themselves, an important open problem is to extend such text index supporting the search of non-contiguous patterns to the collection of similar strings such as DNA sequences from different individuals.

Chapter 4

The Gapped Spectrum Kernel for Support Vector Machines

4.1 Overview

4.1.1 Background

The primary structure, i.e., the amino acid sequence of a protein contains information about the function or structure of the protein. For example, RING-type zinc finger proteins are known to play a key role in ubiquitination pathway [80, 144]. According to protein sequence motif database PROSITE [121], these proteins share conserved sequential pattern `CxHx[LIVMFY]CxxC[LIVMYA]`. Each capital letter represents a particular type of amino acid while `x` represents an arbitrary amino acid. At the fifth and the last position, any type of amino acid in the brackets is valid. Thus, the function in ubiquitination pathway can be associated with the sequence pattern. Because the primary structures of proteins are relatively easier to determine, automatic methods to predict structural or functional features of proteins from sequences have been studied. On the other hand, there is an ongoing paradigm shift in protein study. Although the total number of proteins in nature is difficult to estimate or even define, there is no doubt that currently known proteins represent only a tiny and highly biased fraction of them. The recent development of DNA sequencing technologies enabled us to collect DNA sequences of unprecedented amount by comprehensive manner. Massive amount of protein sequence data are also derived from these DNA sequences.[1] For example, the protein sequence data collected during Global Ocean Survey [112], in which the genomes of marine microbes were sampled from around the world, amounted to about 6.1 million and quadrupled the number of then-known proteins. This type of studies, called metagenomics, are becoming more and more popular these days and producing a large amount of sequence data. In order to analyze these data comprehensively, not only accurate but also computationally efficient sequence analysis methods are needed. String kernel is a promising approach to this goal.

A string kernel is a kernel function defined for strings and can be plugged into SVMs to learn a rule to differentiate between positively labeled data (sequences with some property of interest) and negatively labeled data (sequences without that property); see also Section 2.4. Compared to the approaches based

[1] Protein coding regions of prokaryotic genome can be identified by locating sufficiently long open reading frames, regions that start with `ATG` and end with `TAA`, `TAG` or `TGA`.

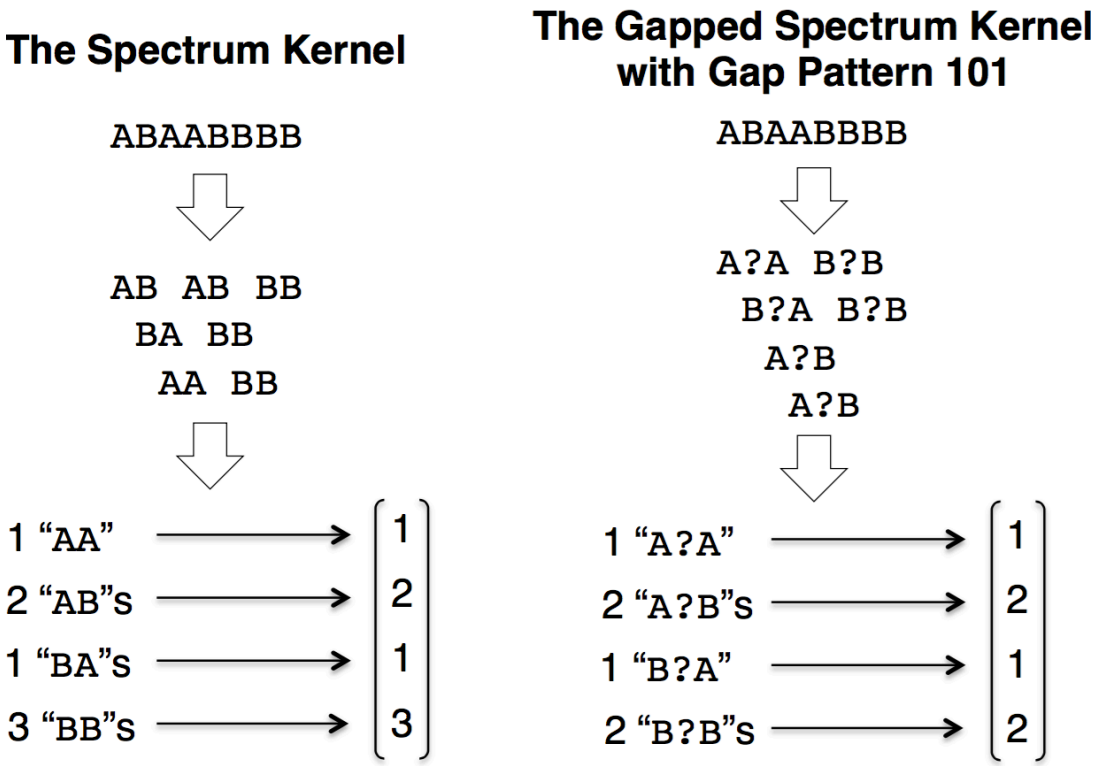


Figure 4.1: The feature vectors of the 2-spectrum kernel and the 101-gapped spectrum kernel for the string ABABBBB.

on explicit signatures of protein groups such as the aforementioned motif from PROSITE, string kernels are able to handle complex differentiation rules that are hard for humans to comprehend. They also conform to the standard formulation of supervised classification problem in computational learning theory (cf. Subsection 2.4.1) and thus, one can apply various techniques developed in computational learning community. In particular, the spectrum kernel [73] is known to be able to achieve classification accuracy and computational efficiency both at high levels. In this kernel, and its extensions, each string is characterized by the frequencies of substructures such as substrings or subsequences contained in the string. For example, in the spectrum kernel, a string is mapped to a σ^k-dimensional vector where k is a parameter. Each dimension corresponds to a k-mer[2] and the value of the dimension is the number of occurrences of that k-mer in the input string (See also Figure 4.1. We give the formal definition of the spectrum kernel in Subsection 4.2.1). The kernel function of two strings is defined to be the inner product of the corresponding vectors. Intuitively, this kernel function measures the closeness of the k-mer compositions of strings.

4.1.2 Our Work

While the spectrum kernel is based on the frequencies of contiguous substrings, there are many motifs of protein groups that are represented as non-contiguous patterns exemplified by the signature of RING-type zinc finger proteins described in Subsection 4.1.1. Thus, it seems reasonable to characterize protein sequences by non-contiguous substructures.

[2]A k-mer means a length k string.

Based on this consideration, we introduce the gapped spectrum kernel, a new variant of the spectrum kernel that is based on the frequencies of gapped patterns. In the gapped spectrum kernel, a string is mapped to a vector in a similar way to the spectrum kernel except that characters of particular positions in each k-mer are not taken into consideration. For example, the right half of Figure 4.1 shows how sequence `ABABBBB` is mapped to a vector while ignoring the second characters of each 3-mer. The gap positions are represented by a binary string. The example in Figure 4.1 corresponds to the case of gap pattern `101`, meaning that the first and the third characters of each 3-mer are respected while the second character is not. We introduce the gapped spectrum kernel formally in Section 4.3.

In order to obtain a rough idea of how the introduction of gaps will affect the string kernel, we calculated the expected values of the (gapped) spectrum kernel between random sequences and those values between patterns containing a motif. To model sequences containing motifs, we selected two functional motifs: `[FILV]Qxxx[RK]Gxxx[RK]xx[FILVWY]` for IQ calmodulin-binding motif from [111] and `[FL]xxxxxx[DN]xx[AGS]x[ST]xG[KRH]GxxGxxxR` for Ribosomal protein L3 signature from PROSITE [121]. Then, we calculated the expected values of (gapped) spectrum kernel for gap patterns of weight (number of ones) 4 and length up to 6. The expectation was taken over all possible choices of random strings where the character for `x` is chosen uniformly randomly from 20 possible amino acids and the character for groups enclosed in brackets are chosen uniformly randomly from valid characters. We padded the left and right of the pattern by 6 uniformly random characters. Because the number of ones in gap patterns we tried are all the same, the expected values for random sequences (without embedded motifs) is all the same. Table 4.1 shows the results. Note that the gap pattern `1111` corresponds to contiguous pattern, i.e., the spectrum kernel. Naturally enough, when patterns are embedded, the expected kernel values increase. However, in almost all cases, the expected value of the gapped spectrum kernel is greater than that of the spectrum kernel. Also, the gap patterns that attain the highest kernel values are different for the two types of motifs.

From the calculations above, we can expect that a) the prediction ability of composition-based kernels can be enhanced by introducing gaps in the substructures counted and b) the effectiveness of gap patterns depends on the application. In practice, even if there exists a simple motif characterizing the proteins with the property of interest, that motif is not known *a priori*. In other words, we do not know which gap pattern to use beforehand. Thus, it is reasonable to combine multiple kernels corresponding to different gap patterns. Indeed, one existing variant of the spectrum kernel called the wildcard kernel [71] corresponds to the case when all possible gap patterns of particular length and weights are combined. The wildcard kernel is known to be able to achieve higher classification accuracy than the spectrum kernel but with higher computational cost. However, it was not known what happens if we incorporate multiple but not all gap patterns. We investigate it in this chapter.

Results. In this paragraph, we describe several time bounds for kernel function computation. In these bounds, n denotes the sum of the lengths of the two input

Table 4.1: The expected values of (gapped) spectrum kernel for random sequences with motifs. The second column is for IQ calmodulin-binding site and the third column is for Ribosomal protein L3 signature. The first row is the expected values for random sequences (without motifs).

Random	0.002756	0.006006
Gap	IQ	Ribosomal
1111	0.424653	0.510544
11101	0.446694	0.718425
111001	1.241633	**3.186738**
11011	0.452865	0.475969
110101	1.168457	0.643206
110011	**2.291657**	0.584056
10111	0.446694	1.747987
101101	0.420833	0.736375
101011	1.464686	1.302725
100111	1.537861	0.482225

strings. We also describe time bounds for the learning and prediction phase of SVM. In these bounds, ℓ denotes the number of sequences in the training set; N is the total length of all input sequences; ℓ' is the number of the support vectors; N' is the total length of all support vectors; and q is the length of the query.

Our first result in this chapter is the introduction of the gapped spectrum kernel and an algorithm to compute it within time independent of the dimension of the feature space. The kernel computation algorithm applies the b-suffix array data structure we introduced in Chapter 3. When the gap pattern is represented by a binary string b, it takes $O(gn)$-time to compute the kernel function where g is the number of runs of ones in b. By this algorithm for kernel function computation, in the learning phase, the kernel matrix can be constructed in $O(g\ell N)$-time. In the prediction phase of SVMs, one can judge if the query sequence belongs to the group observed in the learning phase in time $O(wq)$-time where w is the weight of b.

Then we show that the sum of the gapped spectrum kernels for the parameter b of length k and weights at least $k-m$ is equal to the (k,m)-wildcard kernel. Together with the algorithm for the gapped spectrum kernel computation above, it gives an $O(k^m mn)$-time algorithm to compute the (k,m)-wildcard kernel while the existing algorithm proposed by Leslie et al. [71] takes $O(k^{m+1}n)$-time. By this algorithm, the kernel matrix of the wildcard kernel can be computed in $O(k^{m+1}\ell N)$-time. Also, together with the prediction algorithm for the gapped spectrum kernel, it gives an $O(k^{m+1}q)$-time algorithm for the prediction phase of the wildcard kernel. Different from the existing $O(k^{m+1}(\ell' q + N'))$-time algorithm, our algorithm does not depend on the size of the support vectors.

We experimentally show that the sum of a few gapped spectrum kernels for randomly chosen parameters b is able to predict the protein families comparatively accurately as the wildcard kernel. In order to obtain the wildcard kernel, one needs to take the sum of $\sum_{i=0}^{m}\binom{k}{i} \leq \sum_{i=0}^{m} k^m = O(k^m)$ instances of the

Table 4.2: The time complexities of kernel computation for learning and prediction. ℓ is the number of training sequences. N is the total length of training sequences. ℓ' is the number of the support vectors. N' is the total length of sequences corresponding to the support vectors. q is the length of the query. s is the number of gap patterns considered in the multiple gapped spectrum kernel.

Kernel	Learning	Prediction
k-Spectrum [126]	$O(\ell N)$	$O(q)$
(k,m)-Wildcard [71]	$O(k^{m+1}\ell N)$	$O(k^{m+1}(\ell' q + N'))$
(k,m)-Wildcard (Our method)	$O(k^m m \ell N)$	$O(k^{m+1} q)$
b-Gapped	$O(g\ell N)$	$O(wq)$
(k,m)-Multiple gapped	$O(sm\ell N)$	$O(skq)$

gapped spectrum kernels. Our result indicates that it may not be necessary to take into account all possible gap patterns if we allow a small amount of decrease of the classification accuracy.

Last, we show that the sum above (randomly chosen kernels) can predict protein families as accurately as the kernel derived by taking the sum of the gapped spectrum kernels for different parameters b in a greedy order learned from the training set. This result indicates that, in order to increase classification accuracy by using combining the gapped spectrum kernels for multiple parameters, randomly selecting parameters is already a rather good strategy.

4.1.3 Related Work

Variants of the spectrum kernel. The existing string kernels that are directly relevant to our work are the spectrum kernel and the wildcard kernel. However, there are other variants of the spectrum kernel and they are similar to the wildcard kernel in the sense that they try to achieve higher classification accuracies than the spectrum kernel by taking into account more general substructures than (contiguous) substrings. Thus, here, we give a brief summary of these variants of the spectrum kernel. In this paragraph, we denote the maximum length of input strings for kernel function as n. See Table 4.3 for the summary of the time complexities of the computation of these kernels. The (k,m)-mismatch kernel [72] considers the number of each k-mer that appears in the string with at most $m(<k)$ mismatches as the feature. The computation of the kernel function takes $O(\alpha_{k,m} n)$-time where $\alpha_{k,m} := \sum_{\ell=0}^{\min\{2m,k\}} \binom{k}{\ell}(k-\ell)$ [65]. The k-subsequence kernel [79] considers the number of each k-mer that appears as subsequences[3] in the string as the feature[4]. It takes $O(kn^2)$-time to compute the corresponding kernel function. With an additional condition that the region in the original sequence spanned by each subsequence must be at most h, the (h,k)-restricted gappy string kernel is obtained and the time needed to compute the kernel function drops to $O(\beta_{h,k} n)$-time where $\beta_{h,k} := (h-k)h^{h-k+1}$ [71]. The

[3]A length k subsequence of a string S of length n is $S[i_1]\circ S[i_2]\circ\cdots\circ S[i_k]$ where $i_1, i_2, \ldots, i_k$ are integers satisfying $1 \le i_1 < i_2 < \cdots < i_k \le n$. Note that it is different from a substring.

[4]In the original paper [79], it is called 'string subsequence kernel' with k implicitly assumed.

(k, m)-wildcard kernel [71] is similar to the (k, m)-mismatch kernel but there is a subtlety. See section 4.2 for the detail of the wildcard kernel. The computation takes $O(k^{m+1}n)$-time. Kuksa et al. [65] also gave bounds for the (h, k)-restricted gappy kernel and (k, m)-wildcard kernel that are, except for some minor terms and truncations in analyses, the same as the ones above.

It is difficult to compare the accuracy of string kernels in general because the performance of each kernel depends on the application. Leslie et al. [71] compared the spectrum kernel, the mismatch kernel, the restricted gappy kernel and the wildcard kernel in the context of protein sequence classification, and reported that the mismatch kernel, the restricted gappy kernel and the wildcard kernel are achieve higher prediction accuracies than the spectrum kernel while their accuracies are similar to each other.

Table 4.3: Summary of the time needed to compute related string kernels. n is the sum of input strings for the kernel function. ℓ is the number of training sequences. N is the total length of training sequences. ℓ' is the number of the support vectors. N' is the total length of the support vectors. q is the length of the query. $\alpha_{k,m} := \sum_{s=0}^{\min(2m,k)} \binom{k}{s}(k-s)$. $\beta_{h,k} := (h-k)h^{h-k+1}$.

Kernel	Kernel function	Learning	Prediction
(k, m)-mismatch [72]	$O(\alpha_{k,m}n)$ [65]	$O(\alpha_{k,m}\ell N)$	$O(kq)$
k-subsequence [79]	$O(kn^2)$	$O(kN^2)$	$O(kN'q)$
(h, k)-restricted gappy [71]	$O(\beta_{h,k}n)$	$O(\beta_{h,k}\ell N)$	$O(\beta_{h,k}(\ell' q + N'))$

Alignment-free sequence analysis. Sequence alignment is the standard method to measure biological sequence similarity but it is not appropriate in some cases. For example, although sequence alignment provides informative results when applied to highly similar sequences, aligning sequences with low similarities often lead to meaningless outputs. Also, it is not feasible to apply sequence alignment to exhaustive analyses of large databases because sequence alignment takes time quadratic to the input size. Therefore, though it is not as popular as alignment-based methods, alignment-free methods for sequence analysis have also been studied for a long time. For example, some pioneers in 1960s attempted to classify DNA or protein sequences by the composition of the characters, i.e., nucleotides or amino acids [135]. There exist too many literatures on alignment-free sequence analysis to summarize here. We recommend the recent review articles compiled in a special issue of the journal Briefing of Bioinformatics [136] for more detailed information about this topic. Here, we focus on the relationship of the contents of this chapter to this field.

In general, alignment-free sequence analysis methods first map sequences to Euclidean space and then apply various techniques to analyze vectorial data [135]. Composition-based string kernels such as the spectrum kernel conform to this framework. In fact, one of the main approaches in alignment-free sequence analysis is based on the frequencies of k-mers. (The pioneering work in 1960s mentioned above belongs to the case when $k = 1$.) This is exactly what the spectrum kernel does. Therefore, though the machine learning community and alignment-free sequence analysis community do not seem to be aware of each other, we think that the spectrum kernel and its extensions can also be applied

to problems studied in alignment-free sequence analysis such as phylogenetics and metagenomics. In particular, several recent work [45, 70, 92] reported that, for phylogeny reconstruction problem[5], using the frequencies of gapped patterns leads to higher accuracies than using contiguous patterns.

4.2 Preliminaries

We first introduce notations we use in the rest of this chapter.

The symbol `?` denotes a wildcard and Γ denotes $\Sigma \cup \{\texttt{?}\}$. We denote the set of length k strings on Γ by Γ^k and the set of finite length strings on Γ by Γ^*. The symbol Γ^k_m (resp. $\Gamma^k_{\leq m}$) is defined to be the set of length k strings on Γ with exactly m (resp. at most m) `?`'s. If a k-mer $s \in \Sigma^k$ and a k-mer $t \in \Gamma^k$ satisfy $s[i] = t[i]$ for all i s.t. $t[i] \in \Sigma$, we denote $s \approx t$. An occurrence of $t \in \Gamma^k$ in $S \in \Sigma^*$ is an index i such that $S[i : i + k - 1] \approx t$. We denote the number of occurrences of a string P (on Σ or Γ) in another string $T \in \Sigma^*$ by $occ(P, T)$.

We denote the weight of a binary string b by $w(b)$. Let b be a binary string of length k and weight w and m be $k - w$. The set Γ^b is defined to be $\{S \in \Gamma^k_m : S[i] = \Sigma \text{ iff } b[i] = \texttt{1}\}$. We call each element of Γ^b as a b-mer. Let i_j be the index of the j-th one in b. In other words, $1 \leq i_1 < i_2 < \cdots < i_w \leq k$ and $b[i_1] = b[i_2] = \cdots = b[i_w] = \texttt{1}$. We define $mask_b$ be the map $\Sigma^k \ni x_1 \circ x_2 \circ \cdots \circ x_k \mapsto x_{i_1} \circ x_{i_2} \circ \cdots \circ x_{i_w} \in \Sigma^w$. We also define $mask'_b$ be the map from Σ^k to Γ^k_m such that $mask'_b(S)[i]$ is $S[i]$ if $b[i] = \texttt{1}$; `?` otherwise.

4.2.1 The Spectrum Kernel

Definition. The feature vector $\phi_k(S)$ of the k-spectrum kernel is the vector defined as follows:

$$\begin{aligned}\phi_k(S) :&= (\#\{i : S[i : i + k - 1] = s\})_{s \in \Sigma^k} \\ &= (occ(s, S))_{s \in \Sigma^k}.\end{aligned}$$

Note that the feature vector has σ^k dimensions each corresponding to some k-mer. Thus, the kernel function of two strings S_1 and S_2 is

$$\begin{aligned}K_k(S_1, S_2) :&= \phi_k(S_1)^\top \phi_k(S_2) \\ &= \sum_{s \in \Sigma^k} occ(s, S_1) occ(s, S_2).\end{aligned}$$

Kernel computation. Let $n := |S_1| + |S_2|$. Leslie et al. [73] showed an algorithm based of the suffix tree to compute $K_k(S, T)$ in $O(kn)$-time. Smola and Vishwanathan [126] improved it to $O(n)$-time. We show a version that is based on the suffix array in Algorithm 4.1. We first construct the generalized suffix array SA of S_1 and S_2, i.e., the suffix array of $S := S_1 \circ \$ \circ S_2$ where $\$$ is a character that does not appear in S_1 and S_2. We also construct the height array HGT for SA. Because $S[SA[1] : n + 1], S[SA[2] : n + 1], \ldots, S[SA[n] : n + 1]$ are lexicographically sorted, for each k-mer s, indices i s.t. $S[SA[i] : n + 1]$ is prefixed by s are laid in a consecutive region of the suffix array (if there is any

[5]The phylogeny reconstruction is the problem of reconstructing the history of evolution, e.g., how humans, chimpanzees, gorillas and orangutans emerged from their common ancestor.

such index). Let's call such a consecutive region of the suffix array as a k-region. The suffix array can be decomposed into k-regions. By the definition of the height array, an index i is the first index of a k-region iff $HGT[i] < k$. We scan SA k-region-wise. We maintain the number c_1 of the indices i in the current k-region s.t. $SA[i] \leq |S_1|$. For each such i, $S_1[SA[i] : SA[i] + k - 1] = s$ where s is the k-mer corresponding to the current k-region. In other words, c_1 is the number of occurrences of s in S_1. We also maintain the number of occurrences of s in S_2 similarly. When we reach the end of a k-region, we compute the product c_1c_2. The kernel value is the sum of these products. The construction of SA and HGT takes $O(n)$-time. The main computation is also $O(n)$-time because all operations in the for loop from line 4 takes constant time. Thus, the total time complexity is $O(n)$.

In practice, Algorithm 4.1 is preferable to the suffix tree-based algorithm because the suffix array is more compact than the suffix tree. Also, the main part of Algorithm 4.1 (the for loop from line 4) is an array scan while the corresponding part of the suffix tree-based algorithm is a tree traversal. Thus, the suffix array-based algorithm is more cache friendly.

Algorithm 4.1 The computation of the k-spectrum kernel $K_k(S, T)$

Ensure: $K = K_k(S, T)$
1: Construct the (generalized) suffix array SA of S and T
2: Construct the height array HGT of corresponding to SA
3: $(K, c_1, c_2) \leftarrow (0, 0, 0)$
4: **for** $i = 1$ to n **do**
5: **if** $HGT[i] < k$ **then**
6: $K \leftarrow K + c_1c_2$
7: $(c_1, c_2) \leftarrow (0, 0)$
8: **else**
9: **if** $SA[i] \leq |S|$ **then**
10: $c_1 \leftarrow c_1 + 1$
11: **else**
12: $c_2 \leftarrow c_2 + 1$
13: $K \leftarrow K + c_1c_2$

Learning. If the training data consist of ℓ strings $S_1, S_2, \ldots, S_\ell$ of length n_1, n_2, $\ldots$, n_ℓ respectively, then, the total time to compute the kernel matrix is $\sum_{1 \leq i,j \leq \ell} O(n_i + n_j) = O(\ell N)$ where $N := \sum_{i=1}^{\ell} n_i$.

Prediction. In the prediction phase, the expression

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* K_k(S_i, Q) = \sum_{i=1}^{\ell'} y_i \alpha_i^* \phi_k(S_i)^\top \phi_k(Q) \tag{4.1}$$

is evaluated where $S_1, S_2, \ldots, S_{\ell'}$ are the support vectors, Q is the query sequence, y_i is an input for learning phase and α_i^* is a constant derived in the learning phase (cf. Subsection 2.4.3). The obvious calculation of this expression involves ℓ' kernel function evaluations. The time complexity of this procedure

is $O(\sum_{i=1}^{\ell'}(q+n_i)) = O(q\ell' + N')$ where $q = |Q|$, $n_i = |S_i|$ and $N' = \sum_{i=1}^{\ell'} n_i$. Leslie et al. [73] proposed an $O(kq)$-time algorithm based on table lookup but they did not explain the detail. Smola and Vishwanathan proposed an $O(q)$-time algorithm based on the suffix tree [126]. Though it is not as efficient as Smola et al.'s algorithm, here, we present an average-case $O(kq)$-time algorithm. The algorithm is based on the algorithm of Leslie et al. but we fill a gap in the original description. We choose to present this algorithm because it has a similarity to the prediction algorithm for the gapped spectrum kernel we introduce in Section 4.3. The pseudocode is shown in Algorithm 4.2.

First observe that

$$\begin{aligned}
\sum_{i=1}^{\ell'} y_i \alpha_i^* \phi_k(S_i)^\top \phi_k(Q) &= \sum_{i=1}^{\ell'} y_i \alpha_i^* \sum_{s \in \Sigma^k} occ(s, S_i) occ(s, Q) \\
&= \sum_{s \in \Sigma^k} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ(s, S_i) \right) occ(s, Q) \\
&= \sum_{s=1}^{q-k+1} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ(Q[s : s+k-1], S_i) \right).
\end{aligned}$$

We preprocess the support vectors and prepare the hash table *table* s.t. $table[h(u)] = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(u, S_i)$ for any k-mer u s.t. $occ(u, S_i) > 0$ for some i. When a query Q is given, we lookup $table[h(Q[s : s+k-1])] = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(Q[s : s+k-1], S_i)$ for all s from 1 to $q-k+1$ and output the sum. To implement h, we use the construction described in Section 2.7. Each lookup takes $O(k)$-time to calculate $h(Q[s : s+k-1])$ and $O(1)$-time in expectation to table lookup. Thus, the total time complexity is $O(kq)$ in expectation.

Algorithm 4.2 An $O(kq)$-time prediction algorithm for the spectrum kernel

Require: $table[h(t)] = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(t, S_i)$ for any k-mer u s.t. $occ(u, S_i) > 0$ for some i.
Ensure: $K = \sum_{i=1}^{\ell'} y_i \alpha_i^* K_k(S_i, Q)$
1: $K \leftarrow 0$
2: **for** $i \leftarrow 1$ to $q-k+1$ **do**
3: $\quad K \leftarrow K + table[h(Q[i : i+k-1])]$

4.2.2 The Wildcard Kernel

The feature vector $\phi_{k,m}(S)$ of the (k,m)-wildcard kernel is the vector defined as follows:

$$\begin{aligned}
\phi_{k,m}(S) &= (\#\{i : S[i : i+k-1] \approx t\})_{t \in \Gamma^k_{\le m}} \\
&= (occ(t, S))_{t \in \Gamma^k_{\le m}}.
\end{aligned}$$

Note that each dimension of the feature vector corresponds to some k-mer that contains up to m wildcards. Although both the (k,m)-wildcard kernel and the (k,m)-mismatch kernel are based on the counts of occurrences of the k-mers in the input allowing at most m mismatches, only the former distinguishes where the mismatches occur.

Leslie et al. proposed an $O(k^{m+1}n)$-time algorithm to compute the kernel function $K_{k,m}(S,T) := \phi_{k,m}(S)^\top \phi_{k,m}(T)$ where $n = |S| + |T|$ [71]. By using this algorithm in the learning phase of SVM, the kernel matrix for ℓ strings $S_1, S_2, \ldots, S_\ell$ of length $n_1, n_2, \ldots, n_\ell$ respectively can be computed in $\sum_{1 \le i,j \le \ell} O(k^{m+1}(n_i + n_j)) = O(k^{m+1}\ell N)$-time where $N := \sum_{i=1}^{\ell} n_i$.

In the prediction phase,

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* \phi_{k,m}(S_i)^\top \phi_{k,m}(Q) + b^* = \sum_{i=1}^{\ell'} y_i \alpha_i^* K_{k,m}(S_i, Q) + b^*$$

is evaluated where $S_1, S_2, \ldots, S_{\ell'}$ are the support vectors, Q is the query sequence, y_i is an input for learning phase and α_i^* and b^* are constants derived in the learning phase (cf. Subsection 2.4.3). It takes $\sum_{i=1}^{\ell'} O(k^{m+1}(q + n_i) = O(k^{m+1}(\ell' q + N')$-time where $q := |Q|$, $n_i := |S_i|$ and $N' := \sum_{i=1}^{\ell'} n_i$.

4.3 The Gapped Spectrum Kernel

Definition. Let b be a binary string of length k and weight w. The feature vector $\phi_b(S)$ of the b-gapped spectrum kernel is the vector defined as follows:

$$\begin{aligned}\phi_b(S) &= (\#\{i : mask'_b(S[i : i+k-1]) = t\})_{t \in \Gamma^b} \\ &= (occ(t, S))_{t \in \Gamma^b}.\end{aligned}$$

Note that each dimension of the feature vector corresponds to some b-mer. This feature map is a natural generalization of that of the k-spectrum kernel (It coincides with the k-spectrum kernel when $b = \mathtt{1}^k$.),

Kernel computation. By using the b-suffix array, the kernel function $K_b : (S_1, S_2) \mapsto \phi_b(S_1)^\top \phi_b(S_2)$ can be calculated by an algorithm similar to Algorithm 4.1. The pseudocode is described in Algorithm 4.3. Let $n := |S_1| + |S_2|$. We first construct the generalized b-suffix array b-SA of S_1 and S_2, i.e., the b-suffix array of $S := S_1 \circ \mathtt{\$}^k \circ S_2$ where $\mathtt{\$}$ is a character that does not appear in S_1 and S_2 and $\mathtt{\$}^k$ is $\mathtt{\$}$ repeated for k times. We need k $\mathtt{\$}$'s to delimit S_1 and S_2 independent of b.[6] We also construct the b-height array b-HGT for b-SA. Because $mask_b(S[b\text{-}SA[1] : |S|]), mask_b(S[b\text{-}SA[2] : |S|]), \ldots, mask_b(S[b\text{-}SA[n] : |S|])$ are lexicographically sorted, for each w-mer s, indices i s.t. $mask_b(S[SA[i] : |S|])$ is prefixed by s are laid in a consecutive region of the b-suffix array (if there is any such index). Let's call such a consecutive region of the b-suffix array as a w-region. The b-suffix array can be decomposed into w-regions. By the definition of the b-height array, an index i is the first index of a w-region iff b-$HGT[i] < w$. We scan b-SA w-region-wise. We maintain the number c_1 of the suffixes in the current w-region, i.e., indices i s.t. b-$SA[i] \le |S_1|$. For each such i, $mask_b(S_1[b\text{-}SA[i] : b\text{-}SA[i+k-1]]) = s$ where s is the w-mer corresponding to the current w-region. In other words, c_1 is the number of occurrences of pattern p s.t. $mask_b(p) = s$ in S_1. We also maintain the number of occurrences of such pattern in S_2 similarly. When we reach the end of a w-region, we compute the product $c_1 c_2$. The kernel value is the sum of these products. The construction

[6] For example, if $b = \mathtt{101}$, the gapped 3-mer of $S_1 \circ \mathtt{\$} \circ S_2$ from the $|S_1|$-th position is $mask_b(S_1[|S_1|]\mathtt{\$}S_2[1]) = S_1[|S_1|]S_2[1]$, which is a chimeric string of characters from S_1 and S_2.

of b-SA and b-HGT takes $O(gn)$-time where g is the number of runs of ones in b. The main computation takes $O(n)$-time because all operations in the for loop from line 5 takes constant time. Thus, the total time complexity is $O(gn)$.

Algorithm 4.3 The computation of the b-gapped spectrum kernel $K_b(S_1, S_2)$

Ensure: $K = K_b(S_1, S_2)$
1: Construct the (generalized) b-suffix array b-SA of $S = S_1 \circ \$^k \circ S_2$
2: Construct the b-height array b-HGT of S
3: $(K, c_1, c_2) \leftarrow (0, 0, 0)$
4: $w \leftarrow$ weight of b
5: **for** $i = 1$ to $|S|$ **do**
6: **if** b-$HGT[i] < w$ **then**
7: $K \leftarrow K + c_1c_2$
8: $(c_1, c_2) \leftarrow (0, 0)$
9: **else**
10: **if** b-$SA[i] \leq |S_1|$ **then**
11: $c_1 \leftarrow c_1 + 1$
12: **else if** b-$SA[i] > |S_1| + k$ **then**
13: $c_2 \leftarrow c_2 + 1$
14: $K \leftarrow K + c_1c_2$

Learning. If the training data consist of ℓ strings $S_1, S_2, \ldots, S_\ell$ of length n_1, n_2, $\ldots$, n_ℓ respectively, then the total time to compute the kernel matrix is $\sum_{1 \leq i,j \leq \ell} O(g(n_i + n_j)) = O(g\ell N)$ where $N := \sum_{i=1}^{\ell} n_i$.

Prediction. The expression

$$\sum_{i=1}^{\ell'} y_i \alpha_i K_b(S_i, Q) = \sum_{i=1}^{\ell'} y_i {\alpha_i}^* \phi_b(S_i)^\top \phi_b(Q)$$

can be evaluated in $O(wq)$-time. The pseudocode is shown in Algorithm 4.4.

First observe that

$$\begin{aligned}
\sum_{i=1}^{\ell'} y_i \alpha_i^* \phi_b(S_i)^\top \phi_b(Q) &= \sum_{i=1}^{\ell'} y_i \alpha_i^* \sum_{t \in \Gamma^b} occ(t, S_i) occ(t, Q) \\
&= \sum_{t \in \Gamma^b} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ(t, S_i) \right) occ(t, Q) \\
&= \sum_{s=1}^{q-k+1} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ(mask'_b(Q[s : s+k-1]), S_i) \right).
\end{aligned}$$

We preprocess the support vectors and prepare the hash table $table$ s.t. $table(h[u]) = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(u, S_i)$ for any b-mer u s.t. $occ(u, S_i) > 0$ for some i. When a query Q is given, we lookup $table[h(mask'_b(Q[s : s+k-1]))] = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(mask'_b(Q[s : s+k-1]), S_i)$ for all s from 1 to $q - k + 1$ and output the sum. To implement hash function h, we use the construction described in Section 2.7 Each $mask'_b(Q[i : i+k-1])$ can be represented by a w-mer $mask_b(Q[i : i+k-1])$ and thus, each lookup takes $O(w)$-time in expectation and the total time is $O(wq)$ in expectation.

Algorithm 4.4 An $O(wq)$-time prediction algorithm for the gapped spectrum kernel

Require: $table[h(u)] = \sum_{i=1}^{\ell'} y_i \alpha_i^* occ(u, S_i)$ for any b-mer u s.t. $occ(u, S_i) > 0$ for some i.
Ensure: $K = \sum_{i=1}^{\ell'} y_i \alpha_i^* K_b(S_i, Q)$
1: $K \leftarrow 0$
2: **for** $i \leftarrow 1$ to $q - k + 1$ **do**
3: $\quad K \leftarrow K + table[h(mask'_b(Q[i : i + k - 1]))]$

4.3.1 The Multiple Gapped Spectrum Kernel

The gapped spectrum kernel and the wildcard kernel are linked through the following equation:

$$K_{k,m}(S, T) = \sum_{\substack{b \text{ of length } k \\ \text{weight } \geq k-m}} K_b(S, T). \tag{4.2}$$

As is mentioned in Subsection 2.4.3, taking the sum of kernel functions is equivalent to concatenating the corresponding feature vectors. Let $\phi^{k,m}(S)$ be the concatenation $\phi_b(S)$ for all binary strings of length k and weights less than or equal to m. Here, we concatenate $\phi_b(S)$'s in such a way that $(\phi^{k,m}(S))_{(b,s)} = (\phi_b(S))_s$ for any $(b, s) \in \{\mathtt{0}, \mathtt{1}\}^k \times \Sigma^{k-i}$.

Suppose there is a one-to-one map from Γ_m^k to $\cup_{i=0}^m (\{\mathtt{0}, \mathtt{1}\}^k \times \Sigma^{k-i})$ s.t. if $f(t) = (b, s)$ then $(\phi_{k,m}(S))_t = (\phi_b(S))_s$ for any S. Then,

$$\begin{aligned} K_{k,m}(S, T) &= \phi_{k,m}(S)^\top \phi_{k,m}(T) \\ &= \sum_{t \in \Gamma_m^k} \phi_{k,m}(S)_t \phi_{k,m}(T)_t \\ &= \sum_{(b,s) \in \cup_{i=0}^m \{\mathtt{0},\mathtt{1}\}^k \times \Sigma^{k-i}} \phi_b(S)_s \phi_b(T)_s \\ &= \sum_{\substack{b \text{ of length } k \\ \text{weight } \geq k-m}} K_b(S, T). \end{aligned}$$

Thus, it suffices to show such a map f exists.

Let f be the following map from $\Gamma_{\leq m}^k$ to $\cup_{i=0}^m (\{\mathtt{0}, \mathtt{1}\}_i^k \times \Sigma^{k-i})$: if $f(t) = (b, s)$, $b[i] = \mathtt{1}$ iff $t[i] \in \Sigma$ and s is the concatenation of the non-wildcard characters of t. If $f(t) = (b, s)$, for any string S, $S[i : i + k - 1] \approx t$ iff $mask_b(S[i : i + k - 1]) = s$. Thus, f satisfies the required condition.

From this observation, we define the multiple gapped spectrum kernels as follows. The parameters k, m and some order on $\{\mathtt{0}, \mathtt{1}\}_{\leq m}^k = \cup_{0 \leq i \leq m} \{\mathtt{0}, \mathtt{1}\}_i^k$ are chosen and let b_i be the i-th element of $\{\mathtt{0}, \mathtt{1}\}_{\leq m}^k$. We call $\Sigma_{i=1}^j K_{b_i}$ as the j-th multiple gapped spectrum kernel. From equation (4.2), the last kernel of this sequence matches the wildcard kernel. Let $n = |S| + |T|$. Because each gapped spectrum kernel can be computed in $O(mn)$-time by the algorithm described in Section 4.3, the s-th multiple gapped spectrum kernel can be computed in $O(smn)$-time. This gives $O(k^m mn)$-time bound for the wildcard kernel computation, which is a k/m-factor improvement over the best existing bound proposed by Leslie et al. [71].

In the learning phase, the kernel matrix of the s-th multiple gapped spectrum kernel for strings $S_1, S_2, \ldots, S_\ell$ of length $n_1, n_2, \ldots, n_\ell$ can be computed in $\sum_{1 \leq i,j \leq \ell} O(sm(n_i + n_j)) = O(sm\ell N)$-time. In particular, the kernel matrix of the wildcard kernel can be computed in $O(k^m m\ell N)$-time.

In the prediction phase, we compute contributions from each gap patterns by the prediction algorithm for the gapped spectrum kernel and take the sum. This gives $O(skq)$-time bound for the prediction phase of the s-th multiple gapped spectrum kernel. In particular, it gives an $O(k^{m+1}q)$-time prediction algorithm of the wildcard kernel. Previously, the only existing algorithm for this problem was to compute the ℓ terms of equation 4.1 separately, which takes $O(k^{m+1}(\ell n + N'))$-time.

4.4 Experiments

4.4.1 SCOP Database

To evaluate the efficacy of the proposed methods, we conducted computational experiments of protein family classification. We took the data set from the SCOP database [94, 18, 27]. SCOP had been used in many studies of string kernels [49, 73, 71, 64] and is the *de facto* standard data set in this area. SCOP is a hierarchical database, namely, all proteins in it are first classified into Classes, then members of each Class are further classified into Folds, then into Superfamilies, Families and so on. The levels above Superfamily are classified based on structural features. The levels under Superfamily are based on inferred evolutionary relationships. Superfamily is based both on structural and evolutionary relationships.

There is a caveat for using SCOP data. A SCOP entry represents a domain of a protein instead of the entire protein. A domain is a region of a protein that is considered to be able to function or evolve independently of other parts. A protein can be a single domain or consist of multiple domains. Domain boundaries are not known *a priori* and, in practice, input protein sequences, which is not annotated, may consist of multiple domains. Though it does not exactly reflect the application scenario, we treat each SCOP entry as single datum in order to make our work consistent to the existing work using SCOP data [49, 73, 71, 64]. In practice, if the domain boundaries of the query are unknown, we recommend the user to use all substrings of some fixed length of the original query as the query. It is reasonable to assume that the sizes of the domains in a protein are not drastically different from each other because domains are considered to originate from independent (single domain) proteins. Thus, if the length of substrings is chosen appropriately, most domains contain at least one substring. Also, because the composition-based kernels considered in this chapter do not respect the global position of substructures, they produce similar results for substrings contained in a domain and for the domain itself. Therefore, the relative (dis)advantage of each kernel does not seem to be affected much by such modification.

4.4.2 Experiment Design

We test the ability of multiple gapped spectrum kernel to predict protein Families and compare it to the wildcard kernel. We also test if the ability of multiple gapped spectrum kernel is sensitive to the choice of gap patterns considered. For that purpose, we applied the following procedures for each Family F of the 13 Families including at least 50 protein sequences:

1. split the whole database into two groups, namely, the training set R and the test set E;
2. train the SVM with $R \cap F$ as the positive data set and $R \setminus F$ as the negative data set;
   (a) for each binary string b of length 5 with at most 2 `0`'s, calculate the b-gapped spectrum kernel;
   (b) generate 10 independently random permutations of binary strings of length 5 with at most 2 `0`'s;
   (c) continuously add gapped spectrum kernels according to the permutation giving rise to a multiple gapped spectrum kernels for each permutation;
3. test the SVM with $T \cap F$ as the positive test set and $R \setminus F$ as the negative data set trying every kernel of every sequence.

Note that each set of the multiple gapped spectrum kernels converges to the $(5, 2)$-wildcard kernel because of the equation (4.2). Thus, this experiment yields a kind of interpolations of the gapped spectrum kernel and the wildcard kernel.

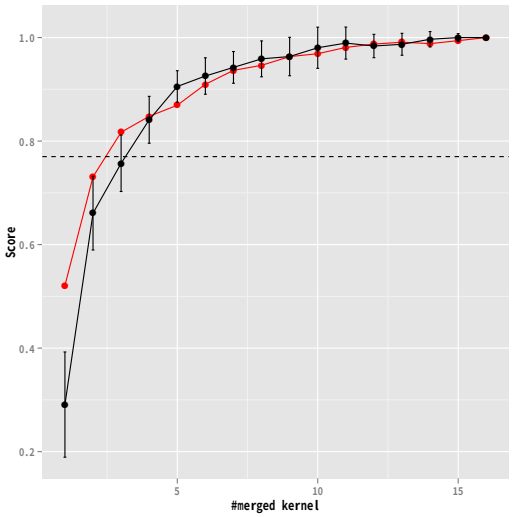
As for the performance evaluation, we used the area under the receiver operation characteristic (ROC) curve [88]. ROC curve is the curve drawn by plotting the false positive rate as the x-coordinate and the true positive rate as the y-coordinate shifting the threshold of classification as a parameter. As we relax the threshold and become more likely to judge one data positive both the true positive rate and the true negative rate increase, producing upper trend ROC curve. Because the increase in terms of the y-coordinate is a good thing while that of the x-coordinate is a bad thing, the more ROC curve extends to upper-left the better. Thus, the performance of a kernel is quantized as area under ROC curve (AUC), which is some value between 0 and 1. Note that the expectation of AUC of uniformly random classification is 0.5, not 0.

In order to assess the effectiveness of the random choice, we also tried a seemingly better yet time expensive method and compared the result with the results of the above experiment. At this time, we conduct 2-fold cross validation in the training set R. Then, we obtained multiple gapped spectrum kernels by greedily adding the gapped spectrum kernels in the order of the score from the highest to the lowest.

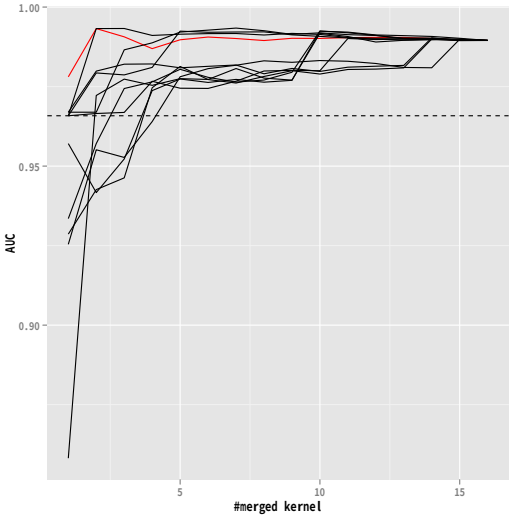
Also, we tried k-spectrum kernels for $3 \leq k \leq 5$ because if the k-spectrum kernel was better than the gapped spectrum kernel or any other kernel, there would be no reason to use that kernel in the first place.

4.4.3 Results

Because it is difficult to argue on the whole results all at once, we first show the summary and, as an example of the result for each Family, the result for Family b.1.1.1. We put the results for other families at the end of this chapter.



(a) Summary of the Results



(b) Result for Family b.1.1.1

Before explanations and discussions, let us mention that the Figure (a) is meant to convey the gist of the results as quickly as possible to the reader and is not suitable for careful analyses. In particular, it went through some post processing, about which we will explain shortly, thus refer to the results for each Family for serious examination.

The original result we obtained for each of the 13 Families looks similar to Figure (b). In each of these figures, the horizontal axis corresponds to the number of the merged kernels and the vertical axis corresponds to the AUC. The black line, the red line and the dashed horizontal line, represent the performances of the random order merges, the greedy order merge and the best k-spectrum kernel respectively. The summary shown in Figure (a) is a kind of 'average' of the results for all Families. Each component is the 'average' version of its counterpart in the results for Families, that is, the black line/dots, the red line/dots and the dashed line represent the 'averages' of the performances of random order merges, the greedy order merges and the best k-spectrum kernels respectively. Besides its construction, Figure (a) clearly demonstrates a) the fast convergence of the sequence of the random order merged multiple gapped spectrum kernels to the wildcard kernel, the rightmost dot; b) good competitiveness of random order strategy against greedy order strategy. Therefore, by using the multiple gapped spectrum kernel derived by merging randomly chosen s gapped spectrum kernels, instead of the wildcard kernel, we can save the computation time by a factor of about $\sum_{i=0}^{m} \binom{k}{i} / s$ and still be able to expect comparatively accurate results. There are several Families for which the k-spectrum kernel performs very well, e.g., c.2.1.2 (Figure (g)), c.37.1.8 (Figure (h)). This is not a problem because the computation of the k-spectrum kernel is cheap and we can just try them first if we want.

Last, we describe how Figure (a) is derived. For each result for a Family, the

horizontal axis was normalized so that the score for the wildcard kernel will be 1 and the average score of the k-spectrum kernels will be 0. Then, black lines are averaged over all the trials. After that, all the components are averaged over all the Families. The vertical bars stabbing black dots represent the average standard deviation.

4.5 Discussion

Predicting protein functions or structures from their sequences is one of the most fundamental problems in bioinformatics. Because the recent development of high-throughput DNA sequencing technologies enabled us to collect virtually unbounded amount of sequence data, not only accurate but also computationally efficient methods for automatic protein function/structure prediction is important. In this chapter, motivated by the existence of motifs including gaps in nature, we first introduced a new variant of the spectrum kernel the gapped spectrum kernel, which is based on the frequencies of gapped patterns in strings. A prototypical calculation indicated that combining multiple gapped spectrum kernels corresponding to different gap patterns is a reasonable approach to design more accurate string kernels. Although a similar idea is used by the wildcard kernel [71], it takes into account all possible gap patterns of a given length and weights and it was not known what happens if we use multiple but not all gap patterns. We experimentally showed that the combination of a few gapped spectrum kernels for randomly chosen gap patterns can predict protein families comparatively as accurately as the wildcard kernel. We also showed that the sum of all gapped spectrum kernels for a given pattern length and weights exactly matches the wildcard kernel. Through this relationship, the algorithms for the gapped spectrum kernel computation and prediction, which are based on the b-suffix array we introduced in Chapter 4, translate to corresponding algorithms for the wildcard kernel. In particular, the time complexity of the derived prediction algorithm does not depend on the size of the support vectors while that of the existing algorithm depends linearly on the size of the support vectors.

Users' Perspective. The training phase of SVM requires annotated sequences. While a large amount of unannotated protein sequences are available and even more are expected to be generated in the future, e.g., by metagenome studies, the number of annotated sequences is relatively small. Thus, in most practical sequence classification problems, prediction phase is more likely to become the computational bottleneck than the learning phase. We raise an example.

A concrete problem to which our methods may be useful is the study of 'protein universe', the set of all possible proteins. Koonin et al. [62] estimated the number of proteins existing in nature to be about 5×10^{10} and stated that determining how these real proteins are distributed in protein universe as a fundamental problem.[7] They also pointed out that "to extract any useful information from this distribution, it needs to be explored in quantitative detail, which can be done only within the framework of a hierarchical taxonomy of proteins".

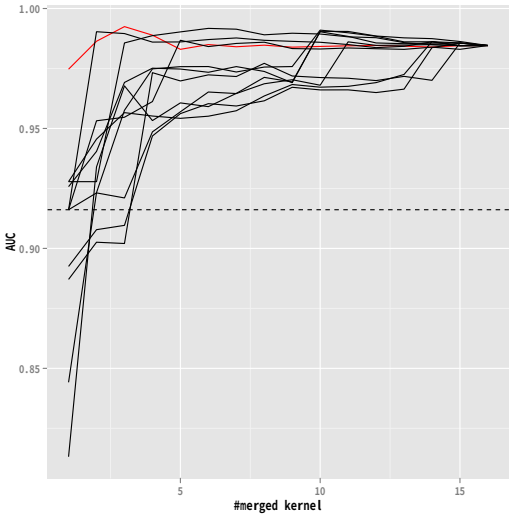
[7]The estimate of the number of real proteins varies among literature. For example, Godzik concludes that the number of eukaryotic proteins can be safely assumed to exceed 10^{12} even without counting intra-species variations [33].

Protein classifications such as SCOP [94, 18, 27] and CATH [104, 122] do provide such hierarchical taxonomies. In this context, because the supply of sequence data from high throughput DNA sequencing technologies is virtually unlimited, it is reasonable to achieve computational efficiency sacrificing accuracy.

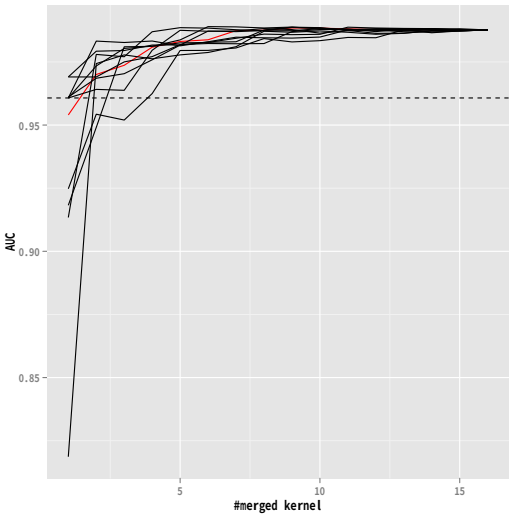
Following existing work [73], we did not tune the soft margin parameter of SVMs in the experiment. However, in general, when one applies machine learning methods to practical purposes, one should tune hyperparameters, i.e., the parameters that are not optimized in the learning phase of the algorithm. Investigating the effect of the soft margin parameter of SVMs on the proposed method is a future work.

As is mentioned in Subsection 4.1.3, it seems possible to apply composition-based string kernels to problems studied in alignment-free sequence analysis community. In particular, Břinda et al. [17] have recently showed that scores based on gapped pattern occurrences can be applied to increase the accuracy of metagenome classification[8]. Although the problem requires to compare short sequences (reads) to long sequences (reference genome), metagenome studies seems to be a promising direction because it often involves a large-scale data and computational efficiency is critically important. Also, in studies on DNA sequences, relatively large k and m are used compared to studies on amino acid sequences. For example, one gap pattern mentioned by Břinda et al. has $k = 36$ and $m = 12$. In such cases, it is very time-consuming to enumerate all gap patterns with the same length and weight and the methods such as the one developed in this chapter may become crucial.

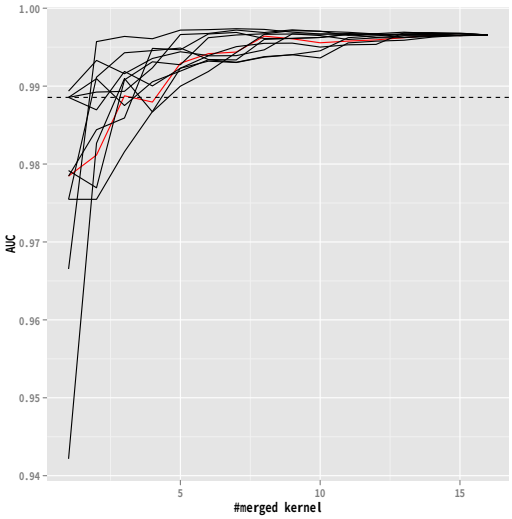
[8]In metagenome classification, one is required to map reads, short fragments of DNA sequences, derived from metagenomic studies to appropriate taxonomical unit, such as mammals or primates.



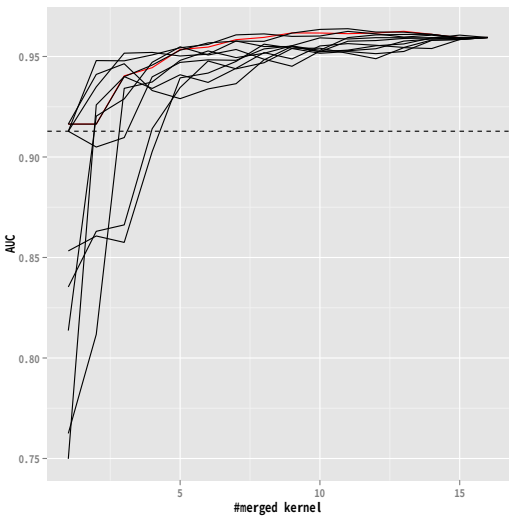
(c) b.1.1.4



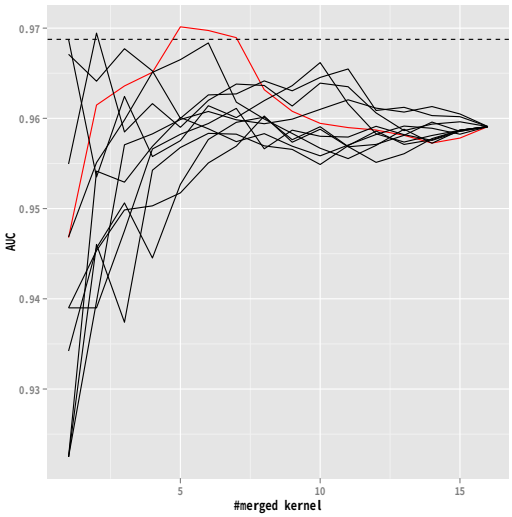
(d) b.1.2.1



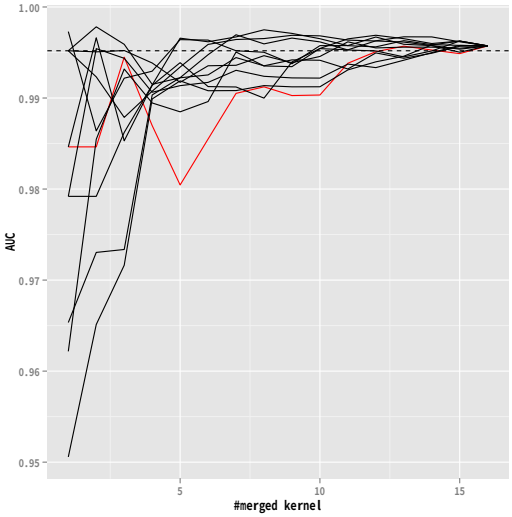
(e) b.36.1.1



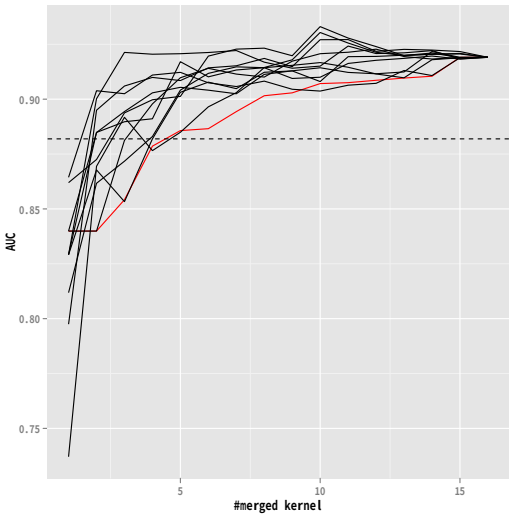
(f) b.40.4.5



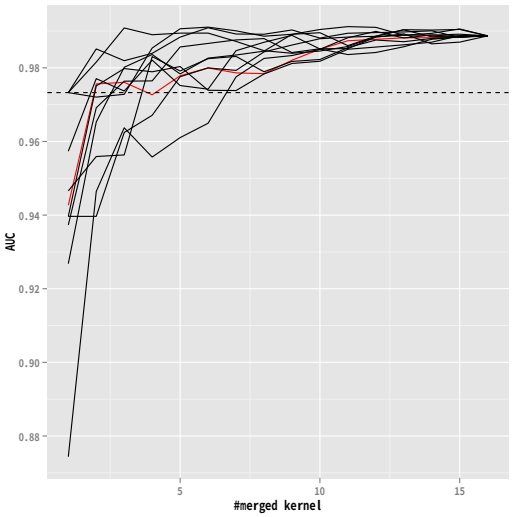
(g) c.2.1.2



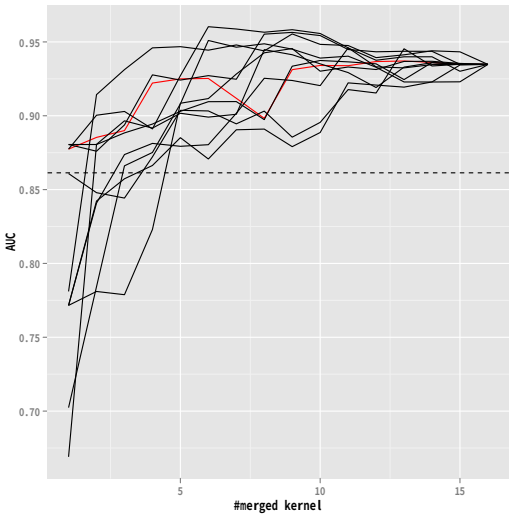
(h) c.37.1.8



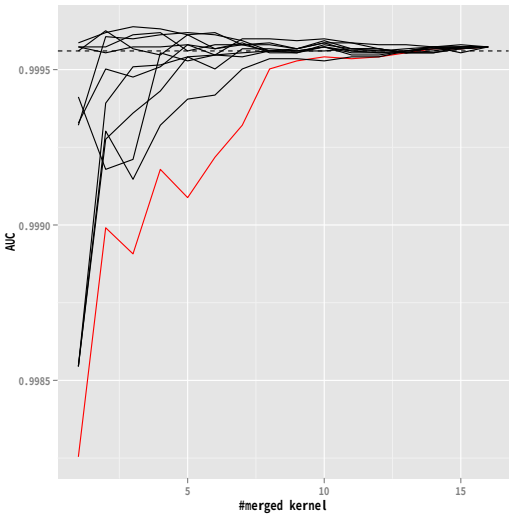
(i) c.94.1.1



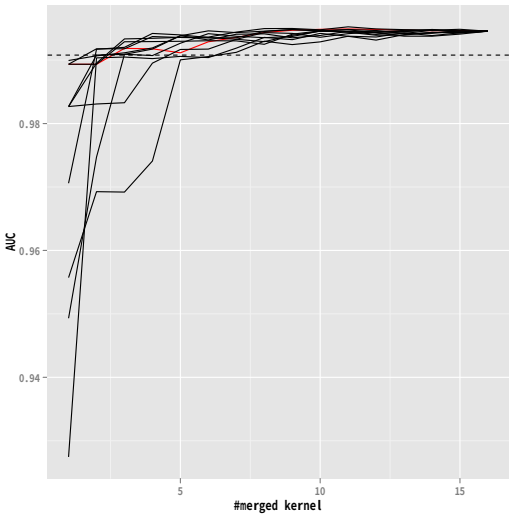
(j) d.58.7.1



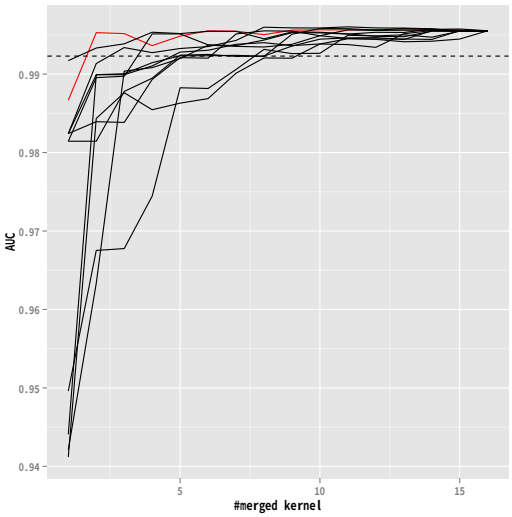
(k) d.108.1.1



(l) d.144.1.7



(m) g.37.1.1



(n) g.39.1.3

Chapter 5

Fast Classification of Protein Structures by an Alignment-free Kernel

5.1 Overview

5.1.1 Background

The functions of proteins are determined mainly by their 3-dimensional structures. For example, an enzyme can recognize its substrate (the substance it acts on) because it has a region called recognition site. The recognition site has a shape that is complementary to the substrate. This complementarity is so strict that the enzyme can recognize substrate specifically. Therefore, if the 3-dimensional structure of a protein is known, we can infer the protein's properties or functions from the structure more directly than from sequences. In some situations, however, computational cost of structure analyses may become a serious concern.

One such situation is the analysis of structure data derived by simulation studies. The methods to obtain protein 3-dimensional structures are categorized into experimental methods and computational predictions. X-ray crystallography and nuclear magnetic resonance are highly successful experimental methods and the throughputs of these technologies are increasing. On the other hand, computational methods, especially molecular dynamics simulation, have several advantages over experimental methods. First, while experimental methods, especially X-ray crystallography, require high quality samples, simulation studies are free from such restrictions. Also, while the structures obtained through experimental methods are basically static, simulation studies can provide high-resolution snapshots of protein folding processes or protein fluctuations. As more computational resources become available, molecular dynamics simulations are starting to produce large amounts of structure data. In a typical molecular dynamics simulation study today, the timestep is on the order of 1 femtosecond $= 10^{-15}$ second and simulation time is on the order of micro to millisecond. If we simulate 1 microsecond of protein folding using 1 femtosecond timesteps and sample every 100 femtoseconds, we ends up with 10^7 structures, only for single simulation. To analyze such large-scale data, computationally efficient methods should be necessary.

Computational efficiency may become important even if the supply of newly identified structure data is limited. One reasonable way to predict the function of a protein from its structure is to categorize proteins with known functions

into groups and, using the structural information, predict the group to which the protein of interest belongs. There, indeed, exist several protein structure databases that maintain classifications such as SCOP [94, 27], CATH [104, 122], and FSSP [41, 42, 43]. However, depending on the applications, one may want to classify proteins differently from these databases. Each time one creates such a new classification, one also needs to rebuild the model used for prediction as well.

Structural alignment. Comparison is the foundation of most analyses. In the past, researchers have developed a group of problems and methods to evaluate the similarities of protein structures called structural alignment. While sequence alignment algorithms align characters, structural alignment algorithms align amino acids. If performed successfully, structural alignment gives results close to the work by human experts and the results are easy to interpret. However, structural alignment is computationally too expensive to apply to classification for the following reasons. First, most formulations of structural alignment involve finding a good correspondence between amino acids in one protein and amino acids in another protein. This gives rise to hard combinatorial optimization problems. Second, most formulations involve complex algebraic scores such as the root mean square deviation. This makes it difficult to classify data by the structural similarity even if the correspondence between amino acids is fixed. There are many instances of structural alignment. For more information about these methods, refer to the review article by Hasegawa and Holm [39].

5.1.2 Our Work

In this chapter, we propose a new similarity measure for protein structures. The similarity measure is a kernel function and we apply it to develop a fast supervised learning method to classify proteins according to their 3-dimensional structures. Our main idea is to apply the techniques of alignment-free method for sequence analysis we saw in Chapter 4 to protein structures to avoid the high computational cost of structural alignment. In fact, the time complexities of most formulations of structural alignment are much higher than quadratic time of sequence alignment. Therefore, the merit of alignment-freeness should be even greater in structure analysis than in sequence analysis. Concretely, we define an alignment-free kernel function for protein structures through a novel use of protein contact maps. The contact map of a protein is a graph with totally ordered vertices representing the amino acids and edges representing the proximity of amino acids in the native state. It was introduced in the context of structural alignment and researchers have used it mainly to formulate contact map overlap problem, where optimal order preserving correspondence between vertices from different contact maps are sought [34]. We, instead, respect the sequential aspect of proteins and characterize each protein by the histogram of square submatrices of the adjacency matrix of the contact map. See also Figure 5.1. We define a feature map for proteins through the histogram of such submatrices.

Physically, the characterization of proteins described above corresponds to focusing on the amino acids contiguous on the backbone chain and counting the patterns of interactions between pairs of such contiguous amino acids. This is a

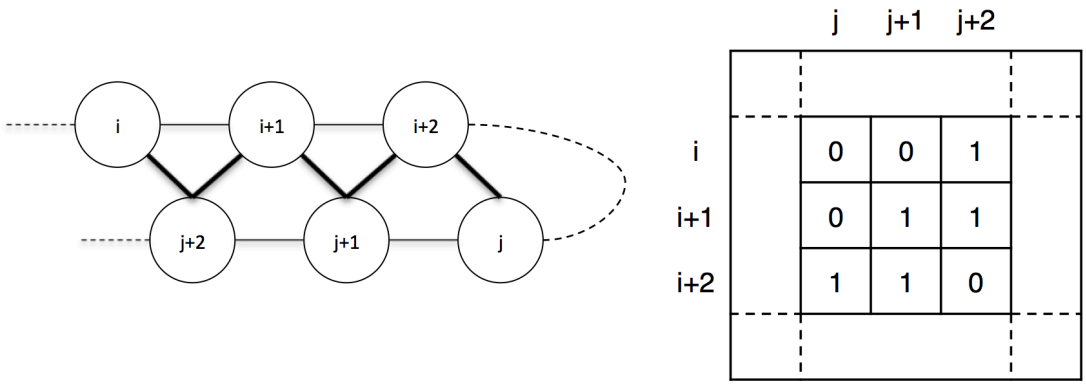


Figure 5.1: The characterization of proteins based on contact map. The left image illustrates a length 3 sub-chain of a protein contacting another length 3 sub-chain. Each circle represents an amino acid. The thin lines are peptide bonds. The doted lines are some possibly long regions of the backbone. The thick lines represent spatial proximities. The right image is the adjacency matrix of the contact map of the protein. The part shown in the left image corresponds to the submatrix shown in the right image.

natural thing to do for two reasons. First, interactions between spatially close amino acids have much more significant impact on the global structure than those between spatially remote amino acids do. In fact, Vassura et al. [134] proposed a method to reliably reconstruct the protein structures from their contact maps[1]. Therefore, by representing proteins by contact map instead of, say, distance matrix (cf. subsection 5.1.3), we lose some information but the loss should be minor. On the other hand, the combinatorial nature of the contact map opens the possibility of efficient solution. Second, amino acids close on the chain are close in space. Thus, we can treat a set of contiguous amino acids as a unit of interaction. In existing work, researchers chose the set of amino acids that are actually spatially close as units of interaction [140, 10]. This is also a reasonable choice indeed but by using the proximity on the chain, and the contact map representation, we can apply the very efficient techniques of combinatorial pattern matching for the computation of the kernel function.

The results of this chapter are summarized as follows:

- We propose a novel alignment-free kernel function for protein structures that is based on protein contact maps;
- We propose an efficient algorithm to calculate the proposed kernel function. The algorithm is based on the two dimensional suffix tree [32, 59] and runs in $\Theta(n^2)$-time where n is the size of input proteins. This bound matches the best existing time bound [140].
- We also propose an algorithm for the prediction phase of the SVMs based the kernel we introduce. The time complexity of the algorithm does not depend on the size of the support vectors;

[1]They reported that contact maps with the threshold value between 10Å and 18Å are usually not realized by multiple drastically different structures.

- We give simple tricks to make the kernel practical that are based on the rigidity and locality of substructures;
- We experimentally show that the combination of the proposed kernel function and the SVMs achieves an accuracy comparative to the most accurate existing method [10] while it runs more than 300 times faster even when implemented by a suboptimal algorithm.

5.1.3 Related Work

Wang et al. [140] and Bhattacharya et al. [10] worked on the same problem addressed in this chapter. For a comparison of the performances of these methods and ours, see Table 5.1, Table 5.2 and Table 5.3. Wang et al. proposed a kernel function for protein structures that incorporates both sequential information, i.e., amino acid types, and structural information. They used the set of amino acids within some distance from an amino acid as a unit of interaction. Their method was also alignment-free, though they did not put an emphasis on this point, and the fastest but the least accurate among the methods we tested. Bhattacharya et al. took the opposite approach to ours and proposed kernel functions based on structural alignment. They used the set of a fixed number of amino acids closest to an amino acid as a unit of interaction. Their method was the most accurate but the slowest among the methods we tested. Qiu et al. [110] also proposed a kernel function based on structural alignment but in a different context of function annotation. This method may be applicable to the problem considered in this chapter but we did not test it because it is similar to Bhattacharya et al.'s method.

DALI [40] is a famous structural alignment algorithm used behind FSSP [41, 42, 43]. It is similar to our method in the sense that it is based on a matrix representation of proteins. However, instead of the contact map, DALI is based on the distance matrix. The distance matrix of a protein of length n is a $n \times n$ matrix whose (i, j) element is the distance between the i-th amino acid and the j-th amino acid. The contact map can be thought of as a kind of binarized version of the distance matrix. In principle, it is possible to use discretized distance matrices in our method but probably contact map is more robust to the error introduced by discretization.

Alignment-free analysis is a well-studied topic for sequential data [136]. The aim of it is to avoid the quadratic cost of pairwise alignment or NP-hardness of multiple alignment. In this chapter, we try the same thing for protein structures, for which even pairwise alignment is hard. In particular, our method can be seen as a two dimensional analogue of the spectrum kernel [73].

5.2 Alignment-free Kernel for Protein Structures

5.2.1 Notation

In this chapter, we model a protein as a sequence of three dimensional coordinates. Each coordinate represents the position of a C_α atom (cf. Figure 2.4). Remember that the distance between neighboring C_α atoms is always about 3.8Å.

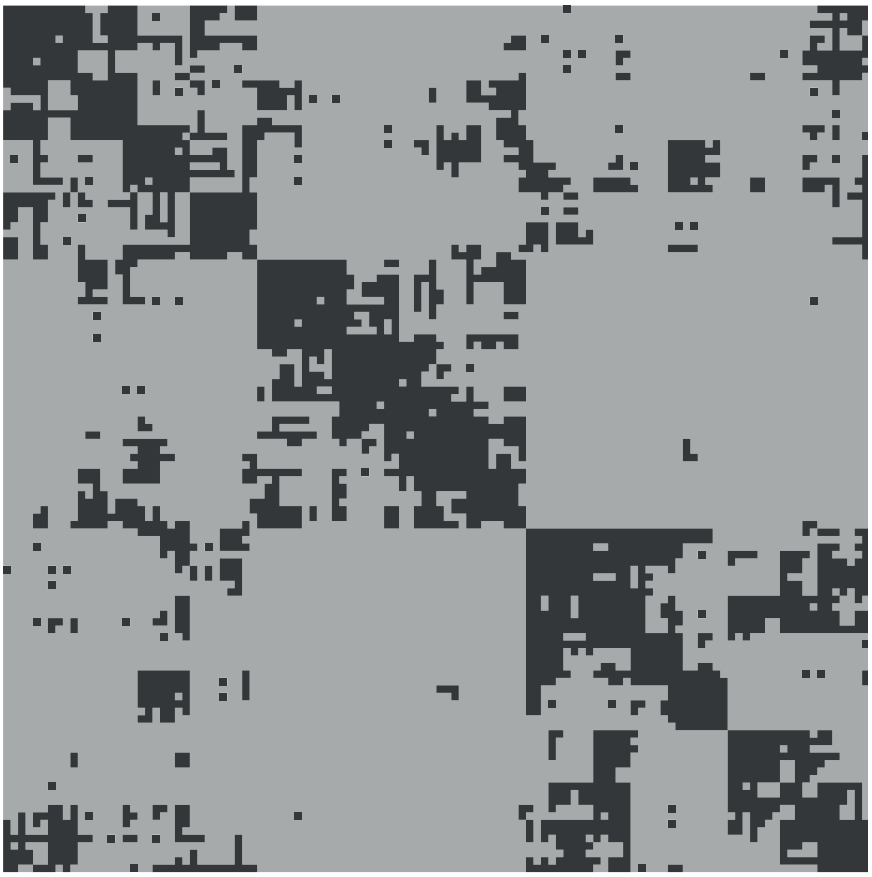


Figure 5.2: The adjacency matrix of the contact map of ribonuclease S (The protein shown in Figure 2.5). t is 12Å. Each black (resp. gray) pixel represents 1 (resp. 0).

Let $P = (p_1, p_2, \ldots, p_n)$ be a protein. The contact map of P is a graph consisting of n totally ordered vertices $v_1 \prec v_2 \prec \cdots \prec v_n$. There exists an edge between v_i and v_j iff the distance between p_i and p_j is less than a parameter $t > 0$. See Figure 5.2 for an example of the adjacency matrix of a contact map.

For a matrices H and M, we denote the number of occurrences of H in M by $occ(H, M)$.

5.2.2 Definition of the Kernel Function

Let P be a protein and A_P be the adjacency matrix of the contact map of P. Let $k > 0$ and $\mathcal{M}_k$ be the space of all $k \times k$ binary matrices. The feature vector $\Phi_k(P)$ is a vector defined as follows:

$$\begin{aligned}\Phi_k(P) :&= (\#\{(i,j) : A_P[i : i+k-1, j : j+k-1] = H\})_{H \in \mathcal{M}_k} \\ &= (occ(H, A_P)_{H \in \mathcal{M}_k}.\end{aligned}$$

See also Figure 5.3. In this figure, $k = 2$. Each dimension of the feature vector corresponds to a 2×2 binary matrix. The dimension corresponding to zero matrix has value 0 because there is no occurrence of 2×2 zero matrix in the adjacency matrix of the contact map. Other values are specified similarly.

The kernel function K_k is the function that takes two proteins P_1 and P_2 and outputs the inner product of $\Phi_k(P_1)$ and $\Phi_k(P_2)$.

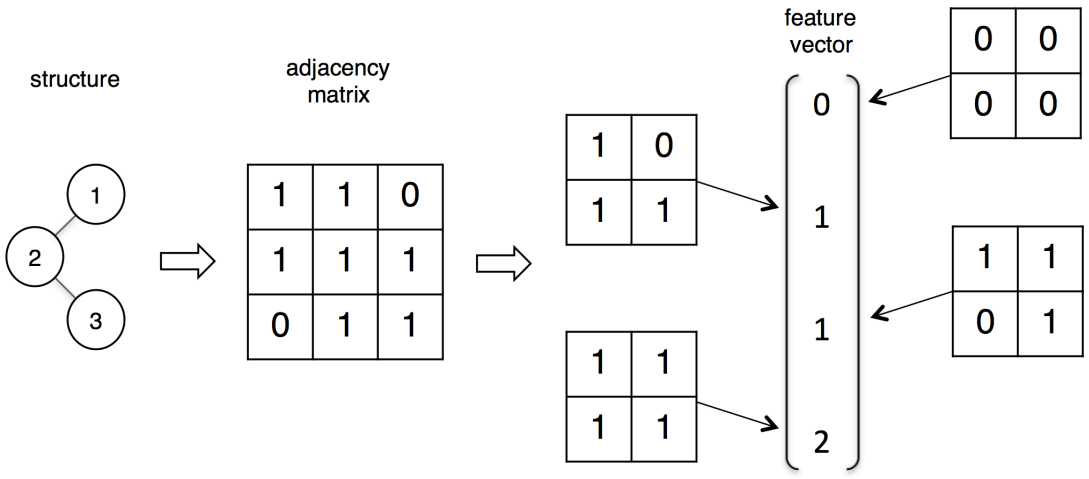


Figure 5.3: The feature map for protein structures.

Table 5.1: Comparison of time complexities. ℓ is the number of training proteins. n_i is the length of the i-th input protein. n is $\max_{i=1}^{\ell} n_i$. ℓ' is the number of the support vectors. q is the length of query protein.

Method	Pairwise	Learning	Prediction
Ours	$O((n_1+n_2)^2)$	$O(\ell^2 n^2)$	$O(q^2k^2)$
[140]	$O((n_1+n_2)^2)$	$O(\ell^2 n^2)$	$O(\ell'(n+q)^2)$
[10]	$O(\mathsf{align}(n_1,n_2))$	$O(\sum_{1\le i,j\le \ell} \mathsf{align}(n_i,n_j))$	$O(\sum_{i=1}^{\ell'} \mathsf{align}(n_i,q))$

5.2.3 Algorithms

In this subsection, we describe the algorithm to calculate the proposed kernel function K_k and algorithms to apply the kernel to the learning and prediction phases of SVM.

Table 5.1 shows the comparison of the time complexities of existing methods and our method. $\mathsf{align}(m,n)$ is the time needed to structurally align proteins of length m and n. $\mathsf{align}(m,n)$ depends on the definition of structural alignment and the algorithms used. Usually, it is not even bounded by polynomial of m and n. In terms of the pairwise kernel function evaluation and learning phase, the time complexity of our algorithm matches the best existing time bound. Also, while the time complexity of the prediction phase of Wang et al.'s method depends linearly on the size of the support vectors $\ell' n$, ours does not depend on it.

Kernel computation. Let P_1 and P_2 be proteins of length n_1 and n_2 respectively. We compute $K_k(P_1,P_2)$ as follows; see also Algorithm 5.1. First, we construct the contact maps of P_1 and P_2. From contact maps, we compute adjacency matrices A_{P_1} and A_{P_2}. Then we construct the generalized Isuffix tree of A_{P_1} and A_{P_2}. Next, we traverse the Isuffix tree in depth first order. During the traversal, the depth of the current node goes up and down. While we are at a node of depth greater than or equal to $2k-1$, we count the number of leaves from A_{P_1} that we have encountered since the last time we were at a node of

depth less than $2k-1$. We also count the same number for A_{P_2}. When we climb up from a node of depth greater or equal to $2k-1$ to a node of depth less than $2k-1$ we add the product of these counts to K and reset counters to 0. After finishing the traversal, the algorithm outputs K.

Let v be a vertex such that the depth of v is greater than or equal to $2k-1$ and the depth of v's parent is less than $2k-1$. Let H be the $k \times k$ matrix s.t. $I(M)$ = the length $2k-1$ prefix of v's path label. The set of labels of the leaves of the subtree rooted at v is equal to the set of occurrences of H. When the depth first traversal goes up from v to v's parent, the algorithm adds (#occurrences of M in A_{P_1}) $\times$ (#occurrences of M in A_{P_2}) to K. This is done for all H that appears in A_{P_1} or A_{P_2}. Thus, $K = K_k(P_1, P_2)$.

Let $n := \max\{n_1, n_2\}$. The sizes of adjacency matrices and the Isuffix tree are all $\Theta(n^2)$. The computation of adjacency matrices and the Isuffix tree and the traversal of the Isuffix tree take $\Theta(n^2)$-time. The contact maps can be constructed in $\Theta(n)$-time probabilistically by hashing [2] but we spend $\Theta(n^2)$-time for other part of the algorithm and thus, we construct the contact map of P_i by calculating the distance of all pairs of coordinates in P_i for $i = 1, 2$, spending $O(n^2)$-time.

Algorithm 5.1 Computation of $K_k(P_1, P_2)$

Ensure: K is $K_k(P_1, P_2)$
1: Compute A_{P_1} and A_{P_2}
2: Construct the generalized Isuffix tree of A_{P_1} and A_{P_2}
3: $(K, c_1, c_2) \leftarrow (0, 0, 0)$
4: **while** traversal of the Isuffix tree in depth first order **do**
5: **if** depth of the current node $< 2k-1$ **then**
6: $K \leftarrow K + c_1 c_2$
7: $(c_1, c_2) \leftarrow (0, 0)$
8: **else**
9: **if** the current node is a leaf from A_{P_1} **then**
10: $c_1 \leftarrow c_1 + 1$
11: **if** the current node is a leaf from A_{P_2} **then**
12: $c_2 \leftarrow c_2 + 1$

Learning. If the training data consist of ℓ proteins $P_1, P_2, \ldots, P_\ell$ of length $n_1, n_2, \ldots, n_\ell$ respectively, then the total time to compute the kernel matrix is $\sum_{1 \le i,j \le \ell} O((n_i + n_j)^2) = O(\ell \sum_{1 \le i \le \ell} n_i^2) = O(\ell^2 n^2)$ where $n := \max_i n_i$.

Prediction. Remember that, in the prediction phase, the expression

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* \Phi(P_i)^\top \Phi(Q) = \sum_{i=1}^{\ell'} y_i \alpha_i^* K_k(P_i, Q)$$

is evaluated where $P_1, P_2, \ldots, P_{\ell'}$ are the support vectors, Q is a query protein, $y_1, y_2, \ldots, y_{\ell'}$ are inputs for learning phase and $\alpha_1^*, \alpha_2^*, \ldots, \alpha_{\ell'}^*$ are constants derived in the learning phase (cf. Subsection 2.4.3). This can be done in $O(k^2 q^2$-time where q is the length of Q. The pseudocode is shown in Algorithm 5.2.

[2] This bound uses the sparsity of protein contact maps. See a discussion in section 5.4.

First, observe that

$$\begin{aligned}\sum_{i=1}^{\ell'} y_i\alpha_i^* K_k(P_i, Q) &= \sum_{i=1}^{\ell'} y_i\alpha_i^* \sum_{H\in\mathcal{M}_k} occ(H, A_{P_i})occ(H, A_Q)\\ &= \sum_{H\in\mathcal{M}_k}\left(\sum_{i=1}^{\ell'} y_i\alpha_i^* occ(H, A_{P_i})\right) occ(H, A_Q)\\ &= \sum_{1\le s,t\le q-k+1}\left(\sum_{i=1}^{\ell'} y_i\alpha_i^* occ(A_Q[s:s+k-1, t:t+k-1], A_{P_i})\right).\end{aligned}$$

We preprocess the support vectors and prepare the hash table *table* s.t. $table[h(u)] = \sum_{i=1}^{\ell'} y_i\alpha_i^* occ(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ(u, A_{P_i}) > 0$ for some i. When a query Q is given, we compute A_Q and lookup $table[h(Q[s : s+k-1, t : t+k-1])] = \sum_{i=1}^{\ell'} y_i\alpha_i^* occ(A_Q[s : s+k-1, t : t+k-1], A_{P_i})$ for all s, t from 1 to $q-k+1$ and output the sum. To implement hash function h, we use the construction described in Section 2.7 treating $k \times k$ matrices as length k^2 strings. Each lookup takes $O(k^2)$-time to calculate $h(Q[s : s+k-1, t : t+k-1])$ and $O(1)$-time in expectation for table lookup. Thus, the total time complexity is $O(k^2q^2)$ in expectation.

Algorithm 5.2 An $O(k^2q^2)$-time prediction algorithm for K_k

Require: $table[h(u)] = \sum_{i=1}^{\ell'} y_i\alpha_i^* occ(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ(u, A_{P_i}) > 0$ for some i.
Ensure: $K = \sum_{i=1}^{\ell'} y_i\alpha_i^* K_k(Q, P_i)$
1: Calculate A_Q
2: $K = 0$
3: **for** $s = 1$ to $q-k+1$ **do**
4: **for** $t = 1$ to $q-k+1$ **do**
5: $K \leftarrow K + table[h(Q[s : s+k-1, t : t+k-1)]$

5.2.4 Practical Considerations

Rigidity. The adjacency matrix of a protein contact map is sparse. See Figure 5.2 for an example. Therefore, K_k, as it is, is dominated by the contributions from sparse submatrices such as the zero matrix and fails to capture the structural similarity correctly. An easy way to fix this problem is throwing away too sparse submatrices. Although this may seem *ad hoc*, it has a conceptual justification. Imagine the space of configurations of an amino acid chain conforming to a given contact map. In one extreme, the contact map does not have an entry whose value is `1` except when it is forced to do so by the neighborhood relationships on the chain. At this point, the chain can take many configurations though it is not completely free. As you add entries of value `1`, the space of allowable configurations will shrink. Therefore, throwing away sparse submatrices corresponds to considering only relatively rigid parts of the protein. This makes sense because we can reliably identify the three dimensional structure from the contact map only when the space of configurations that conform to the contact map is small.

Diagonals and local interactions. Because amino acids close on the chain are close in space, usually, an adjacency matrix has a band of entries of value `1` around the diagonal. See Figure 5.2 for an example. These interactions do not contain much information and thus, it is desirable to discount the contribution of these parts from the kernel function. An easy way to do it is throwing away submatrices too close to the diagonal. It corresponds to counting only interactions between regions remote on the chain. In the experiment, we chose to count only submatrices whose upper left corner (i, j) satisfies $|i - j| \geq k$. Because k is the size of a sub-chain, this is equivalent to counting only mutually exclusive sub-chains. On the other hand, interactions within single group of contiguous amino acids should capture local structures such as helices. To incorporate this information, we define local feature vector $\Psi_k(P)$ as

$$\Psi_k(P) := (\#\{i : A_P[i : i + k - 1, i : i + k - 1] = H\})_{H \in \mathcal{M}_k}$$

and local kernel function $L_k(P_1, P_2)$ as the inner product of $\Psi_k(P_1)$ and $\Psi_k(P_2)$. Without surprise, L_k by itself cannot classify protein structures accurately (results not shown). However, you can increase the accuracy of K_k by combining it with L_k by addition. Note that throwing away local interactions first and then combining local kernel function is not the same as not throwing away local interactions from the beginning.

Formalism. For a protein P, let

$$\Phi'_k(P) := (\#\{(i, j) : A_P[i : i + k - 1, j : j + k - 1] = H, |i - j| \geq k\})_{H \in \mathcal{M}'_k}$$

where $\mathcal{M}'_k$ is the set of all $k \times k$ binary matrices with at least k `1`'s. Let $K'_k(P_1, P_2) = \Phi'_k(P_1)^\top \Phi'_k(P_2)$ and $K''_k(P_1, P_2) := K'_k(P_1, P_2) + L_k(P_1, P_2)$.

Computation. The above modifications can be implemented without increasing the time complexities. The pseudocode of the algorithm to calculate the kernel value $K''_k(P_1, P_2)$ is shown in Algorithm 5.3. The algorithm calculates K'_k, and L_k in parallel and outputs the sum. In line 3, we prepare the prefix sums of A_{P_1} and A_{P_2}. The prefix sum of a binary matrix A is a matrix whose (i, j) entry is the number of entries of value `1` in $A[1 : i, 1 : j]$. Obviously, one can compute the prefix sum in $\Theta(n^2)$-time and once it is derived, one can count the number of entries of value `1` in any submatrix of A by at most 4 lookups to the prefix sum. This method to count entries of value `1` is used in line 12 and line 17 to check if $A[i : i + k - 1, j : j + k - 1]$ contains at least k entries of value `1` or not in constant time. All operations in the while loop takes constant time and thus, the algorithm takes $\Theta(n^2)$-time.

The algorithm for prediction phase is shown in Algorithm 5.4. In prediction phase, we calculate

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* K''_k(P_i, Q) = \sum_{i=1}^{\ell'} y_i \alpha_i^* K'_k(P_i, Q) + \sum_{i=1}^{\ell'} y_i \alpha_i^* L_k(P_i, Q).$$

Each term on the right hand side can be calculated in a similar way as the

Algorithm 5.3 Computation of $K''_k(P_1, P_2)$

Ensure: K'' is $K''_k(P_1, P_2)$

Compute A_{P_1} and A_{P_2}
Construct the generalized Isuffix tree of A_{P_1} and A_{P_2}
Compute prefix sums of A_{p_1} and A_{P_2}
$(K'', K', L, c_1, c_2, c_1^{\mathsf{loc}}, c_2^{\mathsf{loc}}) \leftarrow (0, 0, 0, 0, 0, 0, 0)$
while traversal of the Isuffix tree in depth first order **do**
    if depth of the current node $< 2k - 1$ **then**
        $K' \leftarrow K' + c_1 c_2$
        $L \leftarrow L + c_1^{\mathsf{loc}} c_2^{\mathsf{loc}}$
        $(c_1, c_2, c_1^{\mathsf{loc}}, c_2^{\mathsf{loc}}) \leftarrow (0, 0, 0, 0)$
    else
        if the current node is a leaf representing suffix $(A_{P_1})_{i,j}$ **then**
            if $A_{P_1}[i : i + k - 1, j : j + k - 1]$ contains $\geq k$ 1's and $|i - j| > k$ **then**
                $c_1 \leftarrow c_1 + 1$
            if $i = j$ **then**
                $c_1^{\mathsf{loc}} \leftarrow c_1^{\mathsf{loc}} + 1$
        if the current node is a leaf representing suffix $(A_{P_2})_{i,j}$ **then**
            if $A_{P_2}[i : i + k - 1, j : j + k - 1]$ contains $\geq k$ 1's and $|i - j| > k$ **then**
                $c_2 \leftarrow c_2 + 1$
            if $i = j$ **then**
                $c_2^{\mathsf{loc}} \leftarrow c_2^{\mathsf{loc}} + 1$
$K'' \leftarrow K' + L$

prediction phase of K_k. Observe that

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* K'_k(P_i, Q) = \sum_{i=1}^{\ell'} y_i \alpha_i^* \sum_{H \in \mathcal{M}'_k} occ'(H, A_{P_i}) occ'(H, A_Q)$$

$$= \sum_{H \in \mathcal{M}'_k} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ'(H, A_{P_i}) \right) occ'(H, A_Q)$$

$$= \sum_{\substack{1 \leq s,t \leq q-k+1 \\ |s-t| \geq k \\ A_Q[s:s+k-1, t:t+k-1] \in \mathcal{M}'_k}} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ'(A_Q[s : s + k - 1, t : t + k - 1], A_{P_i}) \right)$$

where, for $k \times k$ matrix H and another matrix M, $occ'(H, M)$ is defined to be

$\#\{(i,j) : M[i:i+k-1, j:j+k-1] = H, |i-j| \geq k\}$. Similarly,

$$\sum_{i=1}^{\ell'} y_i \alpha_i^* L_k(P_i, Q) = \sum_{i=1}^{\ell'} y_i \alpha_i^* \sum_{H \in \mathcal{M}_k} occ^{\mathsf{loc}}(H, A_{P_i}) occ^{\mathsf{loc}}(H, A_Q)$$
$$= \sum_{H \in \mathcal{M}_k} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ^{\mathsf{loc}}(H, A_{P_i}) \right) occ^{\mathsf{loc}}(H, A_Q)$$
$$= \sum_{1 \leq s \leq q-k+1} \left(\sum_{i=1}^{\ell'} y_i \alpha_i^* occ^{\mathsf{loc}}(A_Q[s:s+k-1, s:s+k-1], A_{P_i}) \right)$$

where, for $k \times k$ matrix H and another matrix M, $occ^{\mathsf{loc}}(H, M)$ is defined to be $\#\{i : M[i:i+k-1, i:i+k-1] = H\}$.

To calculate these values, we prepare the support vectors and prepare a hash table $table'$ s.t. $table'[h'(u)] = y_i \alpha_i^* occ'(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ'(u, A_{P_i}) > 0$ for some i and another hash table $table^{\mathsf{loc}}$ s.t. $table^{\mathsf{loc}}[h^{\mathsf{loc}}(u)] = occ^{\mathsf{loc}}(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ^{\mathsf{loc}}(u, A_{P_i}) > 0$ for some i. When a query Q is given, we compute A_Q. To calculate $K' := \sum_{i=1}^{\ell'} y_i \alpha_i^* K'_k(P_i, Q)$, we lookup $table'[h'(A_Q[s:s+k-1, t:t+k-1])]$ for all s, t from 1 to $q-k+1$ s.t. $A_Q[s:s+k-1, t:t+k-1] \in \mathcal{M}'_k$ and take the sum. To calculate $L := \sum_{i=1}^{\ell'} y_i \alpha_i^* L_k(P_i, Q)$, we lookup $table^{\mathsf{loc}}[h^{\mathsf{loc}}(A_Q[s:s+k-1, s:s+k-1])]$ for all s from 1 to $q-k+1$ and take the sum. Then, K'' is the sum of K' and L. To implement h' and h^{loc}, we use the construction described in Section 2.7. We lookup the hash tables for $O(k^2q^2)$ times and each table lookup takes $O(1)$-time in expectation and thus, the total time complexity is $O(k^2q^2)$ in expectation.

Algorithm 5.4 An $O(k^2q^2)$-time prediction algorithm for K''_k

Require: $table'[h'(u)] = y_i \alpha_i^* occ'(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ'(u, A_{P_i}) > 0$ for some i; $table^{\mathsf{loc}}$ s.t. $table^{\mathsf{loc}}[h^{\mathsf{loc}}(u)] = occ^{\mathsf{loc}}(u, A_{P_i})$ for any $k \times k$ matrix u s.t. $occ^{\mathsf{loc}}(u, A_{P_i}) > 0$ for some i.
Ensure: $K'' = \sum_{i=1}^{\ell'} y_i \alpha_i^* K''_k(P_i, Q)$

```
Calculate A_Q
(K'', K', L) = (0, 0, 0)
for s = 1 to q − k + 1 do
    for t = 1 to q − k + 1 do
        if A_Q[s : s + k − 1, t : t + k − 1] ∈ M'_k then
            K' ← K' + table'[h'(A_Q[s : s + k − 1, t : t + k − 1])]
        if s = t then
            L ← L + table^loc[h^loc(A_Q[s : s + k − 1, t : t + k − 1])]
K'' ← K' + L
```

5.3 Experiments

To assess the effectiveness of our algorithm, we tested if it can recover the classification of existing classified databases correctly. We prepared the dataset as follows. First we retrieved the subset of entries of SCOP database [27] with less than 40% sequence identity. Among these, we randomly selected 50 proteins

of length between 100 and 200 from each of the 13 superfamilies containing at least 50 such proteins. Thus, the dataset consists of 13 classes of size 50. We performed 2-fold cross-validation on this dataset using LIBSVM [19]. In 2-fold cross validation, the dataset is randomly divided into two: the training data and the test data. The learning phase of SVM is run on the training data and the labels of the test data are predicted using the results of the learning phase. The same procedure is repeated reversing the role of the training and test data. The average prediction accuracies (the fraction of the data that are assigned correct labels) is output. Each score reported here is the average of the accuracies of 10 independent cross-validations with different training and test sets.

Because the cross-validation involves SVM for multi-class data, we briefly explain how multi-class SVM is constructed from binary SVM in LIBSVM. Let $C_1, C_2, \ldots, C_\ell$ be the classes of data. LIBSVM adopts a one-to-one strategy [60, 86]. In this strategy, the binary classifier for each pair $C_i, C_j (1 \leq i < j \leq \ell)$ is created in the learning phase. There are different methods to use these $\ell(\ell-1)/2$ classifiers in the prediction phase. In LIBSVM, to predict the class of a test datum x, for each $1 \leq i < j \leq \ell$, it is predicted if x belongs to C_i or C_j (by using the binary classifier). If x is predicted to belong to C_i, then C_i receives one vote; otherwise C_j receives one vote. After repeating this procedure for all pairs i, j, the class that received the largest vote is output.

As well as the experiment in Chapter 4, there is a caveat that each entry of SCOP database is a protein domain instead of a whole protein. We used SCOP entries for the experiment in order to make our results easier to compare with existing work [140, 10] that use SCOP data. In practical applications where the query proteins for prediction phase are not annotated, domain boundaries are not known *a priori*, and thus, our experimental setting is not completely the same as practical situations. If the input is a possibly multi-domain protein, we recommend users to use all sub-chains of a particular length of the original query as queries. It does not seem that relative (dis)advantages of different kernels are affected by such a modification.

For the threshold t for contact maps, we used 12 (Å). As for the parameter k, we tried 7 and 8. For each k, we report the performance of both K_k and K''_k. Remember that K''_k is the kernel derived from K_k by throwing away too sparse and too close-to-diagonal matrices and combining L_k. More precisely, we counted neither submatrices that contain less than k entries of value `1` nor submatrices that contain an entry on the diagonal of the enclosing matrix, and we added kernel L_k to the resulting kernel.

We compared our method with the method of Wang et al. [140] and a method of Bhattacharya et al. [10]. The former one is called K_{3Dball} in the original paper. The latter authors proposed many kernels but we only report the result of the kernel called K_1^{Al} in the original paper because we found other kernels were much less accurate than K_1^{Al}. The kernel K_1^{Al} takes the output of a structural alignment algorithm and its performance depends on the structural alignment algorithm. In the original paper, the authors used their own structural alignment algorithm but they did not provide the detail. Thus, we used a famous structural alignment algorithm called combinatorial extension [118] (CE) instead. For the eigendecomposition needed in K_1^{Al}, we used LAPACK [5].

Table 5.2 shows the comparison of classification accuracies. Although K_1^{Al} was the most accurate, K''_8 and K''_9 were comparatively accurate. When we

Table 5.2: Comparison of classification accuracies.

Method	K_{3Dball} [140]	K_1^{Al} [10]	K_7	K_8	K_7''	K_8''
Accuracy(%)	60.31	93.94	83.23	83.78	92.42	92.28

Table 5.3: Comparison of runtimes.

Method	K_{3Dball} [140]	K_1^{Al} [10]	Ours
Time	47 sec	6913 min	21 min

dropped off each trick described in subsection 5.2.4, the accuracy decreased about 2 to 3 % (results not shown). Therefore, each of these tricks has a positive effect on the performance.

Table 5.3 shows the comparison of runtimes. For K_1^{Al} and our methods, we only measured the runtimes of the most time consuming steps. In K_1^{Al}, it was structural alignment and in our kernels, it was suffix sorting. In both cases, other steps took only a few seconds. There is only one entry for the runtimes of our methods because we can use the same suffix tree for the computation of all kernels of ours. For suffix sorting, we did not implement the $\Theta(n^2)$-time algorithm of Kim et al. [59] and instead, we sorted suffixes just by qsort library function of C language. This blows up the time complexity from $\Theta(n^2)$ to $\Omega(n^4 \log n)$. (The precise time complexity depends on the algorithm used inside qsort.) K_{3Dball} was the fastest. Among highly accurate methods, our methods were about 329 times faster than K_1^{Al}.

We also checked the effect of different settings of parameters t and k. Table 5.4 shows the results. We observed a significant gap between $t = 4$ and $t = 6$. This is probably because if t is too small compared to the distance between neighboring C_α-atoms (3.8Å) on the backbone chain, the contact map cannot have any edges. However, the proposed method can achieve high accuracy if t is sufficiently large. It seems that the best value of t will gradually increase according to k increases.

Table 5.4: The effect of parameters on K_k'.

k	t					
	4	6	8	10	12	16
5	7.83	80.15	86.18	85.83	83.57	76.00
6	7.74	82.08	88.89	89.55	89.00	85.95
7	7.75	87.05	89.11	89.03	90.94	87.05
8	7.69	74.57	89.66	89.37	90.94	87.94
9	7.69	70.26	87.71	87.65	89.82	88.35

5.4 Discussion

The 3-dimensional structures of proteins determine their functions. For proteins with known structures, it is more direct and effective to study the structures than the sequences. Though determining protein structures is more difficult than determining sequences, the throughput of experimental methods to identify protein structures are increasing and computational simulations are also producing a large amount of structural data. In this chapter, we proposed an alignment-free kernel function for protein structures that is based on a novel use of the protein contact map and an efficient algorithm to compute the kernel function applying the two dimensional suffix tree. The prediction time of the SVM based on the proposed kernel does not depend on the size of the support vectors. We also experimentally showed that, by using the proposed kernel, one can predict protein superfamilies from structures orders of magnitude faster than the most accurate existing method while achieving an accuracy comparable to it.

Users' perspective. A possible application of the proposed method is the analysis of molecular dynamics simulation trajectories. As we mentioned in Subsection 5.1.1, computer simulation is a method to identify the native structure of proteins. In simulation studies, not only the native structure, but also the information about dynamic processes such as folding, conformation transition driven by environmental factors and thermodynamic fluctuations are produced. Data of dynamic actions of proteins derived through simulation studies can lead to more comprehensive understanding of important biological concepts such as prions or allosteric regulations. The method proposed in this chapter may be applicable to assigning structural classes to dynamically moving structures and finding the point where the properties of structures shift. In applications involving large-scale data such as simulation motion trajectories, it is reasonable to slightly sacrifice annotation accuracies to achieve significant speed-up.

Another possible application is the maintenance of protein structure databases. Several databases of structurally classified proteins exist. The most notable ones are FSSP [41, 42, 43], CATH [104, 122] and SCOP [94, 27]. Though FSSP is maintained completely automatically, CATH and SCOP are maintained by automatic methods based on structural alignment and manual work by experts. Because manual curation by experts does not seem to be able to keep up with the pace of database growth, the method proposed in this chapter may be applicable to support the experts. In this application scenario, the user should be aware of the caveat mentioned in the third paragraph of Section 5.3.

Future work. We close this chapter with some discussions and open problems. Although the number of annotated protein structures are not likely to greatly increase in near future, for large training datasets, the time needed for the learning phase of SVMs may become a serious problem. There are several papers addressing this problem [52, 116]. These results are orthogonal to ours.

In the experiment, we did not fine tune the soft-margin parameter of SVM. As we discussed in Section 4.5, however, it is important to tune hyperparameters to apply machine learning methods to practical problems. Investigating the effect of the soft-margin parameter on the proposed method is a future work.

The proposed kernel involves 2 major parameters: the threshold t for contacts and the size k of submatrices counted. There can also be different ways to throw away too sparse or too close-to-diagonal submatrices. How to optimize these parameters is left as an important future work. However, let us point out that at least learning parameters except t should be computationally cheap because the two dimensional suffix tree is independent of these parameters and thus one can reuse the result of the most time consuming step, the suffix sorting, for different settings of the parameters.

There are several possible ways to improve the proposed method. First, one may be able to speed up our method further by implementing Kim et al's suffix sorting algorithm [59]. This algorithm is based on the same idea as the string suffix sorting algorithm by Kärkkäinen and Sanders [55]. Because the latter is fast in practice, we expect the same is true for the former. Second, one may be able to make our method more accurate by introducing weight rather than throwing away too sparse or too close-to-diagonal submatrices.

When t is independent of n, protein contact maps are sparse because there is a limit on the number of amino acids packed in a certain volume of space. Our algorithm needs quadratic time because it does not take into account the sparsity. Measuring structural similarity of proteins in $o(n^2)$-time is an open problem.

Another idea is to introduce gaps to the kernel function proposed in this chapter in a similar way as the gapped spectrum kernel in Chapter 4. Although it seems to be technically possible, it is not clear if such modification is advantageous or not because we do not have the counterpart of the argument made in Subsection 4.1.2.

One of the main sources of discrepancies between existing classifications of protein structures such as FSSP, CATH and SCOP is that they use different definition of domains of proteins. Domains of proteins are substructures that frequently appear among different proteins and existing databases sometimes choose domains as the database element instead of proteins. In this chapter, we assumed that domains are given but in practice, one needs to determine domains or any other appropriate atomic unit of database first. In principle, domains should be defined by structural comparison according to the definition. However, in existing databases they are chosen heuristically because it is infeasible to perform pairwise structural comparison over the entire database. It is interesting to see if the method developed in this chapter can be used for systematic identification of domains based only on structural information.

Alignment-free analysis of protein structures in general is a vastly open topic. The rapid advancement of DNA sequencing technologies has been increasing the importance of alignment-free methods in sequence analysis. Similarly, for protein structures, alignment-free analysis is a promising approach to achieve efficient analysis, which is going to be more important as the experimental or computational methods to determine protein structures are further developed.

Chapter 6

Detecting Superbubbles in Assembly Graphs

6.1 Overview

In this chapter, we leave database analysis and shift our focus to the *de novo* genome assembly problem. We start by reviewing the problem.

6.1.1 Genome Assembly

DNA sequences are one of the most fundamental information of life and thus, tremendous efforts have been put on developing methods for DNA sequencing, i.e. identifying the sequence of nucleotides in DNA molecules. In particular, since mid 2000s, the so-called next-generation sequencers (NGS) started to appear in the market [132]. By the high throughput and low cost, NGS technologies made whole genome analyses within the reach of smaller laboratories and thus, are revolutionizing the genome research. However, in spite of the extensive efforts put on the development of DNA sequencing technologies, currently there is no method that can read a DNA sequence without splitting it into short fragments called reads. Typical length of an NGS read (in base pairs) is in the order of 10^2 to 10^3 while human genome consists of 3.2×10^9 base pairs and even among bacteria, genome sizes in the order of 10^6 or greater is common.[1] Therefore, genome assembly, the reconstructing the whole genome from reads, became the problem of prime importance.

When the whole genome sequence of an individual of a species, called reference, is available, the genome sequence of another individual of that species or closely related species are usually assembled by associating each read to the similar region of the reference. This procedure is called mapping. On the other hand, when the reference is not available, genome assembly has to be done only from reads. This type of assembly, called *de novo* genome assembly, is, generally speaking, more complicated and needs more computational resources than mapping. Nevertheless, *de novo* genome assembly is necessary to study the or-

[1] A technology called single-molecule real-time sequencing of Pacific Biosciences supports read length of up to 60 kbp. However, this technology is currently not widely used because its cost of reading sequences is relatively higher than the competitor's. As of October 2015, Sequel system from Pacific Biosciences costs around $10,000 to sequence a human genome at 30× coverage while HiSeq X Ten system from Illumina costs less than $1,000. Also, SMRT technologies have a high error rate (more than 10%).

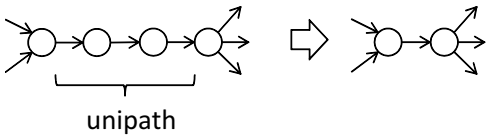
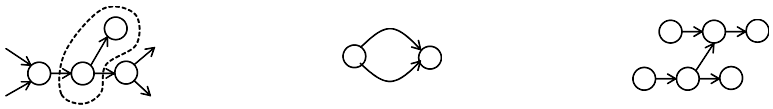


Figure 6.1: Construction of a unipath graph.



(1) A tip (2) A bubble (3) A cross link

Figure 6.2: Assembly graph simple motifs.

ganisms for which references are unavailable. *De novo* genome assembly is also needed for metagenome analysis, the analysis of DNA sequences derived from a collection of organisms in environments such as soils. Because we consider only about *de novo* approach here, in the following, we omit the term '*de novo*' for brevity.

Most assembly algorithms construct some graph in their first stage. They are categorized into two types depending on the types of the graph. Many old-time assemblers utilize a graph called the *overlap graph*, in which a vertex corresponds to a read and an edge corresponds to a pair of reads that have an enough-length overlap [7, 46, 96]. More recent algorithms often utilize a graph called the *de Bruijn graph*, in which an edge corresponds to a k-mer that exists in reads and a vertex corresponds to the shared $(k-1)$-mer between the adjacent k-mers [50, 76, 82, 107, 113, 124, 145]. The de Bruijn graph is said to be more suitable for NGS short reads of large depth.

The next step of most sequencing algorithms after constructing the graph is to simplify the obtained graph by decomposing a maximal unbranched sequence of edges (which is called a *unipath*) into one single edge [50, 82, 113] (Fig. 6.1). The obtained graph is called a *unipath graph*. After obtained the unipath graph, many sequencing algorithms next detect simple typical motif structures caused by sequencing errors. The most common motifs are tips, bubbles, and cross links [50, 76, 113, 145] (Fig. 6.2).

A tip (Fig. 6.2 (1)) is a low-frequency edge whose end (or start) vertex has no outgoing (resp. incoming) edges, which goes out from (resp. comes into) a high-frequency vertex[1]. This motif often appears in case there are some error(s) around the end of a read. A bubble (Fig. 6.2 (2)) consists of multiple edges (with the same direction) between a pair of vertices, which is often caused by error(s) somewhere in the middle of a read. A cross link (Fig. 6.2 (3)) is a low-frequency edge that lies between high-frequency vertices. This appears when a substring of a read accidentally becomes (by error) the same substring that appears in a different region.

[1]We say 'low/high'-frequency vertices/edges for vertices/edges that correspond to few/many reads.

6.1.2 Our work

Problem. All of the aforementioned simple motifs are easy to find in linear time. However, we should consider much more complex structures if input reads are erroneous (as in the case of the third generation sequencers), have many repeats (as in many large-scale genomes/meta-genomes), or have many mutations (as in cancer genomes). Fig. 6.3 shows an example of a subgraph of a unipath graph obtained from actual whole human genome reads (the same set of reads used in the experiments in section 6.4). In this subgraph, paths from the leftmost vertex branch to many paths but they converge into the rightmost single vertex in the end. There are no cycles in this subgraph, i.e., the subgraph forms a directed acyclic graph (DAG). The vertices between the leftmost vertex and the rightmost vertex have no outgoing/incoming edges to/from external vertices (i.e., vertices not in this subgraph). An important point is that all the paths have similar labels with similar lengths.[2] We call this kind of a subgraph a *superbubble*, as it can be considered as an extension of an ordinary simple bubble (more detailed definition of superbubbles will be given in section 6.2). Superbubbles are complicated, but it is apparent that many of them are formed as a result of errors, inexact repeats, diploid/polyploid [3] genomes, or frequent mutations.

Our ideas and results. Given the existence of superbubbles, one important problem, of course, is to find a way to unentangle these structures. However, before finding such methods, we must find the superbubbles themselves. Thinking about the shape of superbubbles, finding them should be helpful for the later analysis too. For example, further time-consuming complicated algorithms (e.g., optimal alignment, paired-end read analyses, etc.) are applicable against the superbubbles, even if they are too computationally expensive to use against the entire graph. However, unlike simple motifs like a bubble, finding superbubbles is not at all trivial. In this chapter, we propose an efficient method to detect superbubbles. In order to do so, we first give a graph theoretic characterization of superbubbles. We, then, prove some properties of superbubbles. In particular, we prove that the number of superbubbles in a graph is bounded by the number of vertices. This motivates the linear time enumeration of superbubbles. Unfortunately, we could not find linear time algorithm but instead, we give a simple algorithm that takes quadratic time in the worst case but runs very efficiently in practice. We also give an explanation on why the proposed algorithm runs fast by showing a linear time bound under a probabilistic model.

6.2 Preliminaries

6.2.1 Superbubble

Here, we formally define superbubbles and show some of their properties which are necessary in the rest of the chapter.

[2]The experiments in section 6.4 will show that the path label lengths of a superbubble are only at most 5% different in more than 85% of the detected superbubbles.

[3]A diploid (resp. polyploid) cell or organism has a pair (resp. a set) of DNA sequences. These sequences are basically the same but not complete copy of each other. Because genome sequencers cannot divide them, reads from different copies are mixed up.

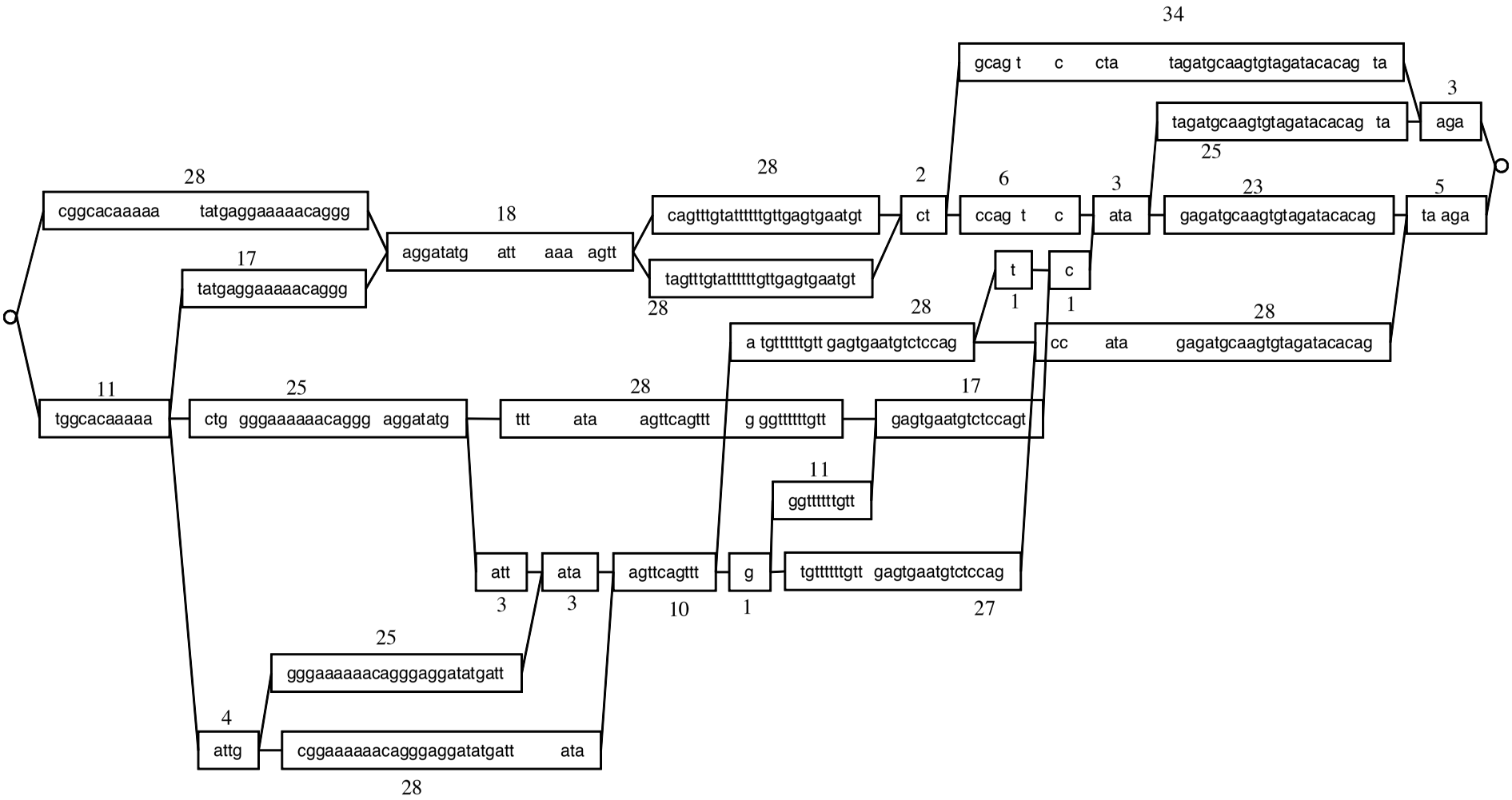


Figure 6.3: A superbubble: A very complicated structure caused by errors or repeats. All the edges are labeled with sequences (vertices are not shown). The gaps in the labels are inserted manually in the figure to show alignment between edge labels that start at different offsets from the entrance of the superbubble.

Definition 6.1. *Let $G = (V, E)$ be a directed graph. If an ordered pair of distinct vertices (s, t) satisfies the following:*

reachability *t is reachable from s;*

matching *the set of vertices reachable from s without passing[4] through t is equal to the set of vertices from which t is reachable without passing through s;*

acyclicity *the subgraph induced by U is acyclic where U is the set of vertices in the above condition;*

minimality *no vertex in U other than t forms a pair with s that satisfies the conditions above,*

then we say that the subgraph in the description of the acyclicity condition is a superbubble *and s, t and $U \setminus \{s, t\}$ are this superbubble's* entrance, exit *and* interior *respectively. For any pair of vertices (s, t) that satisfies the above conditions, we denote the superbubble as $\langle s, t \rangle$.*

To take full advantage of the notation $\langle s, t \rangle$, we first need to confirm that if $(s_1, t_1) \neq (s_2, t_2)$ then $\langle s_1, t_1 \rangle \neq \langle s_2, t_2 \rangle$. The following remark ensures it.

Remark 6.1. *There is a one-to-one correspondence between the vertex pairs satisfying the conditions in Definition 6.1 and superbubbles.*

Proof. The acyclicity condition ensures the vertices of a superbubble can be topologically sorted, i.e., each vertex v can be assigned an integer $ord(v)$ in such a way that $ord(u) < ord(v)$ if v is reachable from u. The vertex s is the vertex of the minimum order because all vertices in $\langle s, t \rangle$ it is reachable from s. The vertex t is the vertex of the maximum order because t is reachable from any vertex in $\langle s, t \rangle$. □

Now we observe a proposition, which clarifies the situation and motivates linear time enumeration of superbubbles.

Proposition 6.1. *Any vertex can be the entrance (resp. exit) of at most one superbubble.*

Note that this proposition does not exclude the possibility that a vertex is the entrance of a superbubble and the exit of another superbubble.

Proof. We prove the proposition by *reductio ad absurdum*. Suppose $\langle s, t_1 \rangle$ and $\langle s, t_2 \rangle$ are distinct superbubbles.

If t_2 is reachable from s without passing through t_1, then t_2 is in the interior of $\langle s, t_1 \rangle$ and (s, t_2) satisfies reachability, matching and acyclicity conditions of superbubble. This contradicts to the minimality of $\langle s, t_1 \rangle$.

If every path from s to t_2 passes through t_1, then t_1 is in the interior of $\langle s, t_2 \rangle$ and (s, t_1) satisfies reachability, matching and acyclicity conditions of superbubble. This contradicts to the minimality of $\langle s, t_2 \rangle$.

□

[4]Passing through a vertex means that visiting and then leaving it, not just visiting or leaving alone.

Corollary 6.1. *There are $O(n)$ superbubbles in a graph with n vertices.*

Before closing this subsection, let us mention yet another property of superbubbles that is not directly necessary for this work but worth mentioning to grasp the picture.

Proposition 6.2. *If two distinct superbubbles share a vertex, either one's exit is the other's entrance or one is included in the other's interior.*

We use the following lemmas to prove Proposition 6.2.

Lemma 6.1. *For any superbubble $\langle s, t\rangle$ and any vertex v, t is reachable from s without passing through v.*

Lemma 6.2. *Let $\langle s_1, t_1\rangle$ and $\langle s_2, t_2\rangle$ be distinct superbubbles such that $s_2 \neq t_1$ and $s_1 \neq t_2$. If s_2 and t_2 are vertices of $\langle s_1, t_1\rangle$, then $\langle s_2, t_2\rangle$ is included in the interior of $\langle s_1, t_1\rangle$.*

Proof of Proposition 6.2. Suppose $\langle s_1, t_1\rangle$ and $\langle s_2, t_2\rangle$ are distinct superbubbles sharing a vertex v and $t_1 \neq s_2, t_2 \neq s_1$.

If $v = t_1$, because $t_1 \neq t_2$ and $t_1 \neq s_2$, t_1 belongs to the interior of $\langle s_2, t_2\rangle$. By Lemma 6.1, there is a path from s_1 to t_1 that does not pass through s_2 and thus $s_1 \in \langle s_2, t_2\rangle$. By Lemma 6.2, $\langle s_1, t_1\rangle$ included in the interior of $\langle s_2, t_2\rangle$.

If $v \neq t_1$ and t_2 is reachable from v without passing through t_1, then $t_2 \in \langle s_1, t_1\rangle$. By Lemma 6.1, t_2 is reachable from s_2 without passing through s_1 and thus, $s_2 \in \langle s_1, t_1\rangle$. By Lemma 6.2, $\langle s_1, t_1\rangle$ is included in the interior of $\langle s_1, t_1\rangle$.

If $v \neq t_1$ and t_2 is reachable from v without passing through t_1, then $t_1 \in \langle s_2, t_2\rangle$. By Lemma 6.1, t_1 is reachable from s_1 without passing through s_2 and thus, $s_1 \in \langle s_2, t_2\rangle$. By Lemma 6.2, $\langle s_2, t_2\rangle$ is included in the interior of $\langle s_2, t_2\rangle$. □

Proof of Lemma 6.2. Because $\langle s_1, t_1\rangle$ and $\langle s_2, t_2\rangle$ are distinct, $s_2 \neq s_1$ and $t_2 \neq t_1$. Thus, s_2 and t_2 are in the interior of $\langle s_1, t_1\rangle$. By Lemma 6.1 t_2 is reachable from s_2 without passing through t_1 and thus, $\langle s_2, t_2\rangle$ is included in the interior of $\langle s_1, t_1\rangle$. □

Proof of Lemma 6.1. The lemma obviously holds if v is not in the interior of $\langle s, t\rangle$. We consider the case when v is in the interior of $\langle s, t\rangle$. We prove the lemma by *reductio ad absurdum*. Suppose every path from s to t passes v. Of course, v is reachable from s. Let V_1 be the set of vertices reachable from s without passing through v. Let V_2 be the set of vertices from which v is reachable without passing through s. Both V_1 and V_2 are subsets of vertices of $\langle s, t\rangle$. For any $u \in V_1$, any path from u to t passes through v because otherwise, t is reachable from s without passing through v. Thus, $V_1 \subset V_2$. For any $u \in V_2$, any path from s to u does not pass through v because otherwise, there is a path from v to v contradicting to the acyclicity of $\langle s, t\rangle$. Thus, $V_2 \subset V_1$. Therefore, $V_1 = V_2$. The subgraph of $\langle s, t\rangle$ induced by V_1 is acyclic because $\langle s, t\rangle$ is acyclic. Therefore, s and v satisfies the first three conditions of superbubble but it contradicts to the minimality of $\langle s, t\rangle$. □

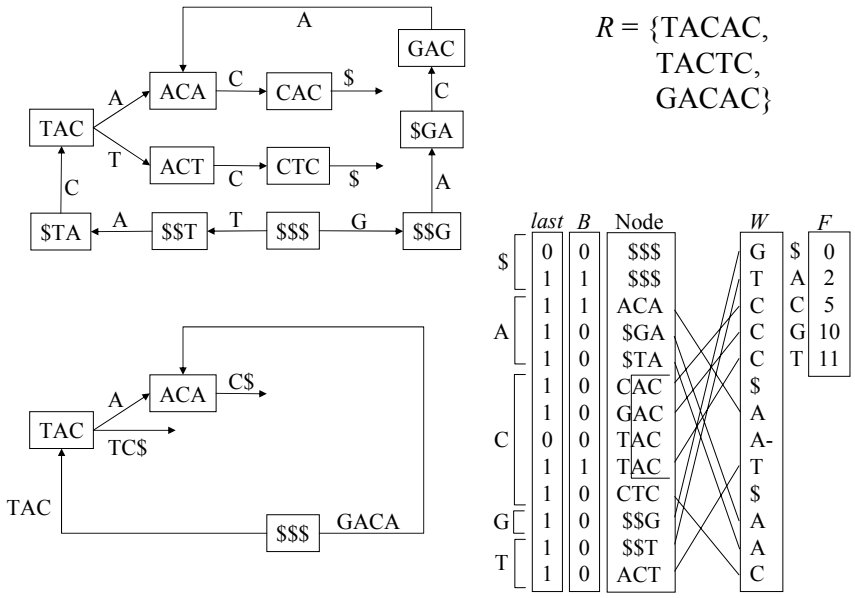


Figure 6.4: Top right: The input set $\mathcal{R}$, top left: The de Bruijn graph of $\mathcal{R}$ with $k = 3, d = 1$, bottom left: the unipath graph, bottom right: the succinct de Bruijn graph and the unipath graph. Non-branching nodes are removed. We store only *last*, B, W and F.

6.2.2 Construction of a Unipath Graph

Given a set $\mathcal{R}$ of reads, we first construct the de Bruijn graph [107]. Let $T = T[1, m]$ be a read of length m in $\mathcal{R}$. The k-mers of T are length-k substrings of T, that is, $T[i, i+k-1]$ for $i = 1, 2, \ldots, m-k+1$. Let K denote the multiset of k-mers of all reads in $\mathcal{R}$, and K_d denote the set of (distinct) k-mers that appear at least d times in K. A k-mer in K_d is called a *solid* k-mer.

The de Bruijn graph $G = (V, E)$ of $\mathcal{R}$ is defined as follows. The vertex set V is the set of $(k-1)$-mers defined as $V = \{T[1, k-1] \mid T[1, k] \in K_d\} \cup \{T[2, k] \mid T[1, k] \in K_d\}$. The edge set E is defined as $\{(u, v) \mid \exists T[1, k] \in K_d, u = T[1, k-1], v = T[2, k]\}$. The edge label of (u, v) is $T[k]$ if $u = T[1, k-1], v = T[2, k]$. Typical values of k and d are $k = 28$, $d = 3$.

We use the succinct de Bruijn graph [13], which is a compressed representation of the de Bruijn graph of $\mathcal{R}$. For a set of m solid k-mers, the succinct de Bruijn graph uses $4m + o(m)$ bits to encode the graph, and supports the following operations.

- *outdeg*(v) (resp. *indeg*(v)) returns the number of outgoing (resp. incoming) edges from (resp. to) vertex v in $O(1)$-time.
- *outgoing*(v, c) returns the vertex w pointed to by the outgoing edge of vertex v with edge label c in $O(1)$-time. If no such vertex exists, it returns -1.
- *incoming*(v, c) returns the vertex $w = T[1, k-1]$ such that there is an edge from w and v and $T[1] = c$ in $O(k)$-time. If no such vertex exists, it returns -1.

From a de Bruijn graph $G = (V, E)$, we construct a unipath graph $G' = (V', E')$ as follows. The vertex set V' is a subset of V such that any vertex in V' has more than one outgoing edges or more than one incoming edges. The edge set E' is the multiset of all pairs (u, v) such that $u, v \in V'$ and there is a path $u, x_1, x_2, \ldots, x_\ell, v$ in G and outdegrees and indegrees of $x_1, x_2, \ldots, x_\ell$ are all one. The edge label of (u, v) is the concatenation of edge labels of $(u, x_1), (x_1, x_2), \ldots, (x_{\ell-1}, x_\ell), (x_\ell, v)$ in G. The length of the edge label is $\ell + 1$.

In addition to the data structure of the succinct de Bruijn graph, we use a bit vector $B[1, m]$ where $m = |E|$ is the number of edges in G to represent the unipath graph G'. We set $B[v] = 1$ if and only if the vertex v of G is also a vertex of G'. The outdegree and the indegree of v in G' is equal to those of v in G. To find the vertex $outgoing(v, c)$ in G', we first compute $w = outgoing(v, c)$ in G. We, then, repeatedly traverse the unique outgoing edge of w until $B[w] = 1$. The resulting vertex is the answer. The unipath graph is constructed in linear time from the succinct de Bruijn graph because each of the *outdeg*, *indeg*, and *outgoing* operations takes constant time. Figure 6.4 shows an example.

6.3 Algorithm

Here, we explain how to enumerate all superbubbles in a given graph. As we have seen in subsection 6.2.1, each vertex can be the entrance of at most one superbubble. Therefore, once we have a way to check if a vertex s has another vertex t s.t. (s, t) is an entrance/exit pair, then we can find all superbubbles by iterating this procedure for all $s \in V$. Below, we focus our attention on this reduced problem.

Description. The algorithm is similar to the topological sorting algorithm by Kahn [54]. The pseudocode is shown in Algorithm 6.1. It takes a directed graph $G = (V, E)$ and $s \in V$ as inputs, and returns $t \in V$ s.t. (s, t) is an entrance/exit pair of a superbubble if such t exists. We call the procedures from line 8 to line 22 of the algorithm as visit to v (v is the vertex dequeued from S in line 8) and denote it by $visit(v)$. Let $indeg(v)$ denote the indegree of vertex v. The algorithm, first, traverses the graph and constructs a table containing the indegrees of all vertices. During the execution of the main loop from line 7 to line 27, the algorithm maintains four dynamically changing values/sets.

- label $label(v) \in \{\mathsf{unseen}, \mathsf{seen}, \mathsf{visitable}, \mathsf{visited}\}$ for each vertex v;
- a queue S containing vertices with label visitable;
- n_{seen} is the number of vertices with label seen;
- for each vertex v, the number of vertex u such that $(u, v) \in E$ and $label(u) = \mathsf{visited}$ is maintained in a variable $ctr(v)$.

Initially, $label(v) = \mathsf{unseen}$ for every $v \neq s$ and $label(s) = \mathsf{visitable}$. Thus, initially, $S = \{s\}$, $n_{\mathsf{seen}} = 0$ and $ctr(v) = 0$ for every vertex v. In $visit(v)$, $label(v)$ is changed to visited. For each vertex u s.t. $(v, u) \in E$, if $label(u) = \mathsf{unseen}$, $label(u)$ is changed to seen and n_{seen} is incremented. If all the vertices v s.t. $(v, u) \in E$ have label visited, $label(u)$ is changed to visitable, u is enqueued into

S and n_{seen} is decremented. The algorithm aborts anytime when it visits a vertex v of outdegree 0, which means a tip, or a vertex v s.t. $(v, s) \in E$, which means a cycle because any vertex visited is reachable from s. After visiting a vertex, the algorithm tests if it is going to visit the exit at the next step as follows. First it checks if S consists of one vertex, for example t, and if $n_{\mathsf{seen}} = 0$, i.e., if no vertex has label seen. If not, the test is negative. Otherwise, the algorithm further checks if $(t, s) \in E$ or not. If $(t, s) \in E$, the algorithm aborts because it just found a path from s to s, a cycle. Otherwise, the algorithm returns t. The algorithm aborts if S becomes empty.

Algorithm 6.1 The algorithm to find the corresponding exit of a potential entrance

Require: directed graph $G = (V, E), s \in V$
Ensure: returns t s.t. (s, t) is an entrance/exit pair of a superbubble if it exists

```
1: traverse the graph and construct the table of indeg(v)
2: label(v) ← unseen for every vertex v
3: label(s) ← visitable
4: queue S ← {s} // S contains vertices that are not visited but can be visited
5: n_seen ← 0
6: ctr(v) ← 0 for every vertex v
7: repeat
8:     dequeue a vertex v ∈ S
9:     label(v) ← visited
10:    if there is no u s.t. (v, u) ∈ E then
11:        abort // tip
12:    for every vertex u s.t. (directed) (v, u) ∈ E do
13:        if u = s then
14:            abort // cycle including s
15:        if label(u) = unseen then
16:            label(u) ← seen
17:            n_seen ← n_seen + 1
18:        ctr(u) ← ctr(u) + 1
19:        if ctr(u) = indeg(u) then // label(w) = visited for ∀w s.t. (w, u) ∈ E
20:            label(u) ← visitable
21:            n_seen ← n_seen − 1
22:            enqueue u into S
23:    if only one vertex t is left in S and n_seen = 0 then
24:        if (t, s) ∉ E then return t
25:        else
26:            abort // cycle including s
27: until |S| = 0 // abort because no vertex is labeled visitable
```

Correctness. Each vertex v can be enqueued into S at most once because it happens when the labels of all the vertices u s.t. $(u, v) \in E$ are changed to visited and once a vertex is labeled visited the label is not changed. Thus, the algorithm can dequeue a vertex from S at most n times and in particular it halts. Below, we prove the correctness of the returned value, which reduces

to the followings: a) If the algorithm returns vertex v, then (s, v) satisfies the conditions of superbubbles except the minimality condition; b) If the input s is the entrance of a superbubble, then the algorithm returns the corresponding exit. To prove a) and b), we first prove two lemmas.

Lemma 6.3. *If a vertex v is labeled* seen *at some point of the algorithm, then v is reachable from the input s. Also, for any vertices u and v, if u has never been enqueued into S when v is labeled* seen*, then, v is reachable from the input vertex s without passing through u.*

Proof. Let v be a vertex that is labeled seen at some point of the algorithm. Because s is labeled visitable at the beginning, $v \neq s$. There is a vertex $p(v)$ s.t. v is labeled seen during $visit(p(v))$. The edge $(p(v), v)$ exists. Because $label(p(v)) =$ visited when v is labeled seen, if $p(v) \neq s$, $p(v)$ is also labeled seen at some point of the algorithm. Consider the following procedure: we initialize a variable x by v, and then, we reassign the value $p(x)$ to x (in other words, $x \leftarrow p(x)$) while $x \neq s$. Because the time x is labeled seen becomes earlier every time we reassign a value to x, no vertex is assigned to the variable x more than once. Thus, this loop halts and the last vertex assigned to x must be s. Suppose k is an integer s.t. $p^k(v) = s$ where p^k is p applied for k times. Then, $(p^k(v), p^{k-1}(v), \ldots, p(v), v)$ is a path from s to v. Thus, the first half of the lemma holds. If a vertex u has never been enqueued into S when a vertex v is labeled seen, $p^\ell(v) \neq u$ for $1 \leq \forall \ell \leq k$ because $p^\ell(v)$ is enqueued into S before v is labeled seen. Thus, the last half of the lemma holds. □

Let $V_{\mathsf{unseen}}, V_{\mathsf{seen}}, V_{\mathsf{visitable}}$ and V_{visited} be the set of vertices labeled as unseen, seen, visitable and visited respectively. Let V_{to} be the set of vertices that are reachable from s without passing through any element of $V_{\mathsf{visitable}} \cup V_{\mathsf{seen}}$ and let V_{from} be the set of vertices from which at least one element of $V_{\mathsf{visited}} \cup V_{\mathsf{visitable}}$ is reachable without following an edge to s.

Lemma 6.4. *After each execution of the main loop from line 7 to 27 of the pseudocode in Algorithm 6.1, $V_{\mathrm{to}} = V_{\mathsf{visited}} \cup V_{\mathsf{visitable}} \cup V_{\mathsf{seen}}$ and $V_{\mathrm{from}} = V_{\mathsf{visited}} \cup V_{\mathsf{visitable}}$.*

Proof. We prove the first half by mathematical induction on the number of iteration steps of the main loop. After the first visit , V_{visited}, $V_{\mathsf{visitable}}$, V_{seen} consist of s, vertices v s.t. $(s, v) \in E$ and $indeg(v) = 1$ and vertices v s.t. $(s, v) \in E$ and $indeg(v) > 1$ respectively. Therefore, the lemma holds. Suppose the lemma holds up to some iteration. During the next visit to a vertex, for example v,

1. v is removed from $V_{\mathsf{visitable}}$ and added to V_{visited};
2. all $u \in V_{\mathsf{unseen}}$ s.t. $(v, u) \in E$ are removed from V_{unseen} and added to V_{seen};
3. all $u \in V_{\mathsf{seen}}$ s.t. $(v, u) \in E$ and $ctr(u) = indeg(u)$, i.e., $label(w) =$ visited for every vertex w s.t. $(w, u) \in E$, are removed from V_{seen} and added to $V_{\mathsf{visitable}}$.

Consequently, because $label(v)$ is changed from visitable to visited, V_{to} acquires those vertices u s.t. $(v, u) \in E$ and $label(u)$ was unseen before $visit(v)$. Because

these are also the vertices that $V_{\mathsf{visited}} \cup V_{\mathsf{visitable}} \cup V_{\mathsf{seen}}$ acquires, $V_{\mathrm{to}} = V_{\mathsf{visited}} \cup V_{\mathsf{visitable}} \cup V_{\mathsf{seen}}$ still holds. After $visit(v)$, $V_{\mathsf{visited}} \cup V_{\mathsf{visitable}}$ acquires those vertices u for which, just before $visit(v)$, v is the only vertex w s.t. $(w, u) \in E$ and $label(w) \neq \mathsf{visited}$. The set V_{from} also acquires these vertices because, by the definition of V_{from}, $V_{\mathsf{visited}} \cup V_{\mathsf{visitable}} \subseteq V_{\mathrm{from}}$ always holds. They are the only vertices that V_{from} acquires because any path to such a vertex u must pass some w s.t. $(w, u) \in E$ and, before $visit(v)$ is executed, such a vertex w is already in $V_{\mathsf{visited}} \cup V_{\mathsf{visitable}} \subseteq V_{\mathrm{from}}$. Thus, $V_{\mathrm{from}} = V_{\mathsf{visited}} \cup V_{\mathsf{visitable}}$ still holds.

□

Proof of a). In this proof, unless otherwise stated, we consider the labels at the time when the algorithm halts. Suppose that the algorithm returns a vertex t. The label of t is visitable. Also, $t \neq s$ because s is labeled visited soon after the algorithm starts running. Thus, t is labeled seen at some point of the algorithm. By Lemma 6.3, t is reachable from s, i.e., the reachability condition holds.

By the condition that the if statement from line 23 is executed, $V_{\mathsf{seen}} = \emptyset$, $V_{\mathsf{visitable}} = \{t\}$. Thus, $V_{\mathrm{to}} = V_{\mathsf{visited}} \cup \{t\} = V_{\mathrm{from}}$. We first show that the subgraph induced by $V_{\mathrm{from}} = V_{\mathrm{to}}$ does not contain a cycle. Then, we show that the matching condition holds and the set of the vertices reachable from s without passing through t, which is equal to the set of the vertices from which t is reachable without passing through s, is equal to $V_{\mathrm{from}} = V_{\mathrm{to}}$. These two claims mean that the acyclicity condition also holds and thus, the proof completes.

We show that the subgraph induced by $V_{\mathrm{from}} = V_{\mathrm{to}}$ does not contain a cycle by showing that it contains neither a cycle including s nor a cycle not including s. If there was an edge (v, s) for $v \in V_{\mathsf{visited}}$, then the algorithm should have aborted in line 14 of $visit(v)$. If there was an edge (t, s), then the algorithm should have aborted in line 26 of the last visit. Therefore, $(v, s) \notin E$ for any $v \in V_{\mathrm{from}} = V_{\mathsf{visited}} \cup \{t\}$. Thus, the subgraph induced by V_{from} does not contain a cycle including s. Suppose the subgraph induced by V_{from} contains a cycle not including s. Every vertex in this subgraph has label visited or visitable. Let v be the first vertex that was labeled visitable in the cycle. When v was labeled visitable, every vertex u s.t. $(u, v) \in E$ must have label visited. Thus, the vertex u in the cycle s.t. $(u, v) \in E$ must have been labeled visitable before v was. However, this contradicts to the way v was chosen. Therefore, V_{from} does not contain a cycle not including s.

Now we show that the matching condition holds and the vertices reachable from s without passing through t is $V_{\mathrm{from}} = V_{\mathrm{to}}$. The last half follows because $V_{\mathsf{visitable}} \cup V_{\mathsf{seen}} = \emptyset \cup \{t\} = \{t\}$. To prove the first half, it suffices to show that V_{from} consists of all vertices from which t is reachable without passing through s. For each $v \in V_{\mathsf{visited}}$, there is some u s.t. $(v, u) \in E$ because otherwise, the algorithm should have aborted in line 11 of $visit(v)$. Also, for such a vertex u, $label(u) =$ seen or visitable or visited because $label(u)$ is changed to seen if it was unseen in line 16 of $visit(v)$. This means that u is also in V_{visited} or $u = t$ because $V_{\mathsf{visitable}} = \emptyset$ and $V_{\mathsf{seen}} = \{t\}$. Thus, from a vertex in $V_{\mathsf{visited}} \subseteq V_{\mathrm{from}}$, we can repeat following an edge to another vertex in V_{from}. We can repeat this edge following procedure as long as we do not reach t. On the other hand, as we already showed, the subgraph induced by $V_{\mathrm{from}} = V_{\mathsf{visited}} \cup \{t\}$ does not contain a cycle. Thus, the edge following procedure above cannot be repeated forever and must end up reaching t. Therefore, V_{from} is the vertices from which t is reachable

without passing an edge to s, which is equal to the set of vertices from which t is reachable without passing through s. Thus, the matching condition holds.

□

Proof of b). Let t be the exit corresponding to s. We first show that the algorithm does not halt before t is enqueued into S. Then we show that, if t is enqueued into S, then the algorithm outputs t.

There are 4 possible reasons for Algorithm 6.1 to halt: 1) it finds a tip (line 11); 2) it finds a cycle including s (line 14 or line 16); 3) S becomes empty (line 27); 4) it finds a superbubble (line 24). We define type-1 (resp. type-2, 3 and 4) halt to be the event that the algorithm halts by reason 1 (resp. 2, 3, and 4) before t is enqueued into S.

Suppose $visit(v)$ is executed before t is enqueued into S. Because t has never been enqueued into S when v is labeled seen, v is reachable from s without passing through t by Lemma 6.3. Thus, v is a vertex of $\langle s, t\rangle$. Because v is a vertex of $\langle s, t\rangle$, t is reachable from v. Thus, v is not a tip. Therefore, type-1 halt does not happen.

Suppose that, type-2 halt happens. If line 14 is executed, let v be the vertex being visited at that time. If line 26 is executed, let v be the only vertex left in S at that time. In any case, $(v, s) \in E$. The vertex t has never been enqueued into S when v is labeled seen. Thus, by Lemma 6.3, v is reachable from s without passing through t, i.e., v is a vertex of $\langle s, t\rangle$. However, $(u, s) \notin E$ for any vertex u of $\langle s, t\rangle$ because otherwise, the concatenation of the path from s to u and the edge (u, s) form a cycle, violating the acyclicity condition of $\langle s, t\rangle$. This contradicts to $(v, s) \in E$. Thus, type-2 halt does not happen.

Next we prove that type-3 halt does not happen. We first show that type-3 halt does not happen just after $visit(s)$. Then, we show that if type-3 halt happens just after $visit(v)$ for some vertex $v \neq s$, then, there is some vertex u s.t. $label(u) =$ seen at that time. Last, we show that the existence of such a vertex u contradicts to the acyclicity condition of $\langle s, t\rangle$. We prove the first part. Because $\langle s, t\rangle$ is a directed acyclic graph, each vertex v of $\langle s, t\rangle$ can be assigned a topological order, i.e., an integer $ord(v)$ satisfying that if v is reachable from u, then $ord(u) < ord(v)$. For any v in $\langle s, t\rangle$ s.t. $v \neq s$, $ord(s) < ord(v)$. Let s' be a vertex of the second smallest topological order. Then, s' is reachable from s but it is not reachable from any other vertex in $\langle s, t\rangle$. Thus, $(s, s') \in E$ and $(u, s') \notin E$ for any vertex u in $\langle s, t\rangle$ s.t. $u \neq s$ or s'. Therefore, just after $visit(s)$, $s' \in S$ and type-3 halt does not happen. Next, we prove the second part. Suppose that type-3 halt happens just after $visit(v)$ for some vertex $v \neq s$. Because $v \neq s$ there was a visit before $visit(v)$. There is a vertex u s.t. $label(u) =$ seen at the beginning of $visit(v)$, because otherwise, the algorithm should have aborted in the if statement from line 23 to line 26 just after the last visit. Because $S = V_{\mathsf{visitable}}$ becomes empty just after $visit(v)$, $label(u)$ is not changed to visitable in $visit(v)$. Therefore, $label(u) =$ seen when the algorithm halts. Now we prove the third part. Because t has never been enqueued into S when u is labeled seen, u is reachable from s without passing through t by Lemma 6.3. In other words, u is in $\langle s, t\rangle$. In the following, unless we state otherwise, we consider labels of the time when the algorithm halts. Let w be a vertex in $\langle s, t\rangle$ s.t. $label(w) =$ seen or unseen. Because s is the first vertex visited, $label(s) =$ visited and thus, $w \neq s$. Because w is a vertex of $\langle s, t\rangle$, w is reachable from s. In particular,

there is at least one vertex $p(w)$ s.t. $(p(w), w) \in E$. Because t is reachable from $p(w)$ via w without passing through s, $p(w)$ is a vertex of $\langle s, t\rangle$. Also, we choose $p(w)$ from $V_{\mathsf{seen}} \cup V_{\mathsf{unseen}}$. It is possible because $w \notin V_{\mathsf{visitable}}$ has at least one vertex w' s.t. $(w', w) \in E$ and $label(w') \neq \mathsf{visited}$ and $V_{\mathsf{visitable}} = \emptyset$. Because $label(s) = \mathsf{visited} \neq label(p(w)) = \mathsf{seen}$ or unseen, $p(w) \neq s$. Consider the following procedure: we initialize a variable x by u, and then, we reassign the value $p(x)$ to x (in other words, $x \leftarrow p(x)$). Because the number of vertices is finite while the procedure above can be repeated infinitely many times, there is a pair of integers k, ℓ s.t. $k > \ell$ and $p^k(u) = p^\ell(u)$ where p^k is p applied for k times. Therefore, $\langle s, t\rangle$ contains a cycle $(p^k(u), p^{k-1}(u), \ldots, p^\ell(u))$, which contradicts to the acyclicity condition.

Suppose type-4 halt happens and $t' \neq t$ is output. Because t has never been enqueued into S when t' is labeled seen, t' is reachable from s without passing through t, i.e., t' is in $\langle s, t\rangle$. Also, (s, t') satisfies all conditions of superbubbles except the minimality condition by a). However, this contradicts to the minimality condition of $\langle s, t\rangle$. Therefore, type-4 halt does not happen.

We proved that the algorithm does not halt before t is enqueued into S. On the other hand, suppose t is indeed enqueued into S. If, at that time, there is a vertex $v \neq t$ in $\langle s, t\rangle$ with $label(v) \neq \mathsf{visited}$, there is a vertex u on the path from v to t s.t. $label(u) = \mathsf{visited}$ or $\mathsf{visitable}$ and there is a vertex w with edge (w, u) and $label(w) \neq \mathsf{visited}$. This contradicts to the condition by which u is labeled $\mathsf{visitable}$. Thus, all vertices in $\langle s, t\rangle$ other than t have label $\mathsf{visited}$. Also, because t is not visited yet, no vertices outside of $\langle s, t\rangle$ has been labeled seen. Thus, there is no vertices labeled seen. Therefore, the algorithm outputs t. □

Analysis. In the worst case, each execution of Algorithm 6.1 takes $\Theta(n+m)$-time and in total the calculation of all superbubbles takes $\Theta(n(n+m))$-time. Below, we show that, under a probabilistic model, the algorithm takes constant time on average and thus all superbubbles can be found in $\Theta(n)$-time in total.

We apply a generative model of network called configuration model [8].

Definition 6.2. *An incoming (resp. outgoing) half-edge to (reps. from) a vertex v is an edge $(*, v)$ (resp. $(v, *)$ where $*$ represents a vertex that is not determined yet. An incoming half-edge $(u, *)$ and an outgoing half-edge $(*, v)$ can be connected to form an edge (u, v). Given a sequence of pairs of integers $((p_1, q_1), (p_2, q_2), \ldots, (p_n, q_n))$ s.t. $0 \leq p_i, q_i \leq 4$ and $\sum_i p_i = \sum_i q_i =: m$, the graph is derived as follows; see also Figure 6.5:*

1. *One prepares vertices $v_1, v_2, \ldots, v_n$ s.t. v_i is attached p_i incoming half-edges and q_i outgoing half-edges;*

2. *One selects a matching between m incoming half-edges and m outgoing half-edges uniformly randomly and connects matched pairs.*

Let a $1, 0$-vertex be a vertex of indegree 1 and outdegree 0. We prove the following:

Proposition 6.3. *For a graph with n vertices that was derived from the model of Definition 6.2 with at least $\lceil cn \rceil$ $1, 0$-vertices where $0 < c < 1$ is a positive constant, Algorithm 6.1 runs in constant time in expectation.*

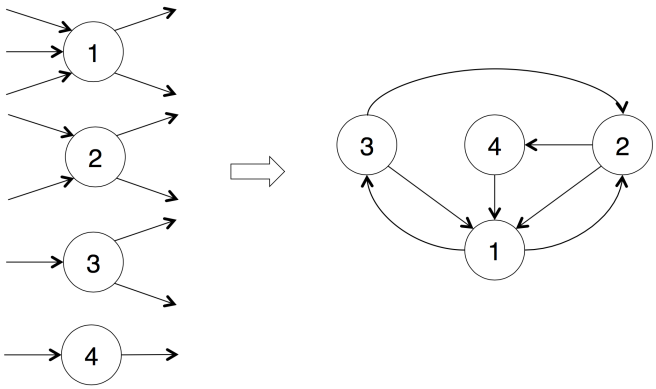


Figure 6.5: Configuration model.

Before proving Proposition 6.3, we explain why this proposition is relevant for the analysis of the asymptotic behavior of Algorithm 6.1, especially for the cases when it is applied to the assembly graphs derived from currently existing DNA sequencers. A 1,0-vertex corresponds to a tip in assembly graphs. Assembly graphs usually contain many tips because reads generated by the current DNA sequencers contain reading errors. It is often possible to detect a tip of this type because compared to the edges corresponding to non-erroneous regions, such an edge usually has a lower coverage, i.e., the number of reads that give rise to the edge is relatively small. However, it is not reasonable that all tips can be removed in this way. It is possible that some tips are introduced by errors in reads but the coverage is not sufficient to reliably judge it to be derived from errors. Also, because the coverage is not uniform across the original sequence, sometimes, reads are not sampled from some regions of the original sequence. In this case, the boundaries between regions from which reads are sampled and regions from which reads are not sampled appear as tips in the assembly graphs. These tips should not be removed because they correctly represent some parts of the original sequence (assuming reading errors did not occur in the corresponding regions). Because the number of tips is related to the error rates, the variance of the coverage or possibly other properties of the DNA sequencing technologies, it is not reasonable to think that the ratio of tips to the non-erroneous reads changes asymptotically as the size of the graph grows. Therefore, here, we analyze the asymptotic behavior of Algorithm 6.1 assuming that the ratio of 1,0-vertices to other type of vertices is some positive constant.

Proof of Proposition 6.3. In line 12, the algorithm checks each edge from v and see which vertex u it is connected to. If u is a $1,0$-vertex, it is enqueued into S and the algorithm halts when it dequeues u from S or sometime before that. On the other hand, if a vertex is enqueued into S at time t_1 (counted from the beginning) and dequeued from S at time t_2, $t_2 = \Theta(t_1)$ because the size of S at time t_1 is $O(t_1)$ and each vertex dequeued from S is processed in constant time. Thus, it suffices to show that, if the algorithm does not stop for other reasons, the expectation of the time a $1,0$-vertex is enqueued into S for the first time is bounded by a constant.

Because incoming half-edges correspond to outgoing half-edges one-to-one and we check each edge at most once, we can think that, instead of a matching

between incoming half-edges and outgoing half-edges is chosen at the beginning, each time we check an edge from v in line 12, one outgoing half-edge from v is connected to a uniformly randomly chosen incoming half-edge that is not yet connected to an outgoing half-edge. Let's focus on a particular edge checking from, for example, a vertex v. For each i from 1 to n, let r_i be the number of half-edges $(*, v_i)$ that was connected to an outgoing half-edge before that point. For each vertex v_j, the probability that an incoming half-edge $(*, v_j)$ is connected to $(v, *)$ at the current edge checking is $(p_j - r_j)/\sum_{i=1}^{n}(p_i - r_i)$. If v_i is a $1,0$-vertex, $p_i = 1$. The probability that the incoming-half edge of a particular $1,0$-vertex is connected to $(v, *)$ at the current edge checking is $1/\sum_{i=1}^{n}(p_i - r_i) \geq 1/\sum_{i=1}^{n} 4 = 1/4n$. If no $1,0$-vertex is enqueued into S so far, no incoming half-edge of $1,0$-vertices was connected to an outgoing half-edge, and thus, the probability that the incoming-half edge of any $1,0$-vertex is connected to $(v, *)$ at the current edge checking is at least $cn \times (1/4n) = c/4$. For $k \geq 1$, the probability that the first incoming half-edge to a $1,0$-vertex is connected to an outgoing half-edge at the k-th edge checking is at most $(1 - c/4)^{k-1}$. Thus, the expectation of such k is bounded by $\sum_{k=1}^{\infty} k(1-c/4)^{k-1} = 16/c^2 = O(1)$. □

In the graph we constructed from human genome reads (The same data we use in Section 6.4), the fraction of the vertices of indegree 1 and outdegree 0 was 28%.

6.4 Experiment

Procedures. We first constructed the succinct de Bruijn graph with parameter $k = 27$ and $d = 3$ for the read set SRX016231, which was derived by sequencing a human individual by an Illumina sequencer. The length of each read is 100bp and the coverage is about 40. Next, we constructed the unipath graph as described in Subsection 6.2.2. The resulting unipath graph consists of 107,154,751 vertices and 210,207,840 edges. Last, we found all superbubbles in the unipath graph by the algorithm in section 6.3.

Results. Table 6.1 is the histogram of the size of superbubbles where the size of a superbubble means the number of vertices in it. The superbubbles of size 2 are omitted because they are ordinary bubbles. The superbubble of Fig. 6.3 is of size 20 and this histogram tells, among other things, that there are hundreds of equally or more complex superbubbles. On the other hand, what matters the most for the application to the genome assembly problem is whether superbubbles really capture erroneous or repeat/mutation abundant regions, which topological complexity alone does not necessarily suggest. One way to assess the relevance of a superbubble in this regard is to compare the length of paths in it where length of an edge is the length of the sequence represented by the edge. Note that topologically close paths can have a variety of lengths because each edge can be originated from a unipath. However, among 23,078 superbubbles of size equal to or greater than 5 we found, 19,926 (86.3%) of them have the longest/shortest path length ratio smaller than 1.05. Therefore, superbubbles such as the one in Fig. 6.3 are indeed typical.

In terms of the computation time, it took 742.1 seconds for a Xeon 3.0GHz CPU to enumerate all superbubbles including ordinary bubbles. The number of

Table 6.1: Histogram of the size of superbubbles

size	3-9	10-19	20-29	30-39	40-49	50-59	60-
#S.B.	71663	4295	347	69	21	8	3

vertices visited was 126,537,254.

6.5 Discussion

We introduced the concept of superbubbles in assembly graphs and showed several properties of superbubbles. We also proposed an efficient algorithm for detecting them. The algorithm is very easy to implement (Our implementation of Algorithm 6.1 needed only 228 lines of C code.).

Future work. Many tasks remain as future work. Several follow up work on the enumeration of superbubbles appeared after the publication of the results in this chapter. Sung et al. [128] gave an $O(m \log m)$-time algorithm where m is the number of edges. Recently, Brankovic et al. published a preprint paper proposing a worst-case linear-time algorithm [15]. These studies did not provide experimental results. To estimate the potential of further speed up in practice, we compared the runtime of our algorithm and graph traversal by breadth first order. For the graph derived from human genome reads, our algorithm takes about 9.05 times longer than graph traversal. Therefore, it seems difficult to design algorithms that run faster than our method by more than that factor. Still, developing a linear-time algorithm that is also practical remains open. There are many future work on more biological issues. Developing methods for categorizing the detected superbubbles (e.g., errors, repeats, mutations, and polyploids), and methods for fixing errors in superbubbles are important future tasks. It is also interesting to extend our algorithm for other bubble-like structures (e.g. the bulge structure [100]).

Users' perspective. In this chapter, we focused on detection of superbubbles, which is necessary no matter what one does by superbubbles. Here, we discuss to what kind of problems superbubbles can possibly be applied.

A natural possible application is the correction of assembly graphs. As we mentioned in Section 6.1, read errors introduce noise to the graph and ordinary bubbles may not be sufficient to fix it. Often, assemblers try to iteratively remove bubbles to fix errors. However, complex structures such as Figure 6.3 are not always decomposed into sequences of bubbles. Therefore, general concepts such as superbubble are potentially useful. As well as bubbles, superbubbles can also emerge for several reasons other than read errors such as repeat sequences, mutations and diploid/polyploid genomes. One needs to identify which superbubble corresponds to sequencing errors. Conventional techniques for the same problem for bubbles such as comparing coverage (the number of reads corresponding to each vertex of the de Bruijn graph) or underlying strings of edges seem to work for superbubbles as well. Coverage information should also be used to remove errors.

Table 6.2: Result of BLAT Search of `tagtttgtatttttgttgagtgaatgt`

chromosome	strand	start	end
6	+	26818088	26818115
6	-	58268167	58268193
6	-	58300525	58300551
6	-	26785643	26785669

Table 6.3: Result of BLAT Search of `cggcacaaaaatatgaggaaaaacagg`

chromosome	strand	start	end
6	+	26818042	26818068
6	-	58268214	58268239

Another potential application of superbubbles is evolutionary studies of repeated sequences. Repeats constitute a large portion of eukaryotic genomes. For example about 50% of human genomes are repeats. One of the main sources of repeated sequences is transposable elements (TEs), "selfish DNAs" that function in a similar way as viruses to host genomes. A TE usually codes proteins that can recognize and cut[5] the TE itself and paste possibly multiple and inexact copies of it onto other locations of genomes. Until 1990s, these interspersed repeats were predominantly considered as "junk"s, sequences without biological relevance [11]. They also complicate tasks such as genome assembly and thus, methods to detect and remove them have been studied [1, 53, 109]. However, more recently, TEs, and thus repeat sequences, have acquired higher status because a wide variety of functional or evolutionary roles of them became known [11]. Repeats are likely to appear as superbubbles in assembly graphs. For example, Table 6.2 (resp. Table 6.3) shows the result of BLAT search of the longest (resp. second longest) edge label in the superbubble in Figure 6.3 on human genome (NCBI36/hg18). These results means the superbubble in Figure 6.3 corresponds to inexact repeats in chromosome 6. Therefore, instead of unentangling superbubbles, it may be more relevant to extract useful information about repeat regions from superbubbles. In particular, cancer researchers Atsushi Niida [98] and Yuichi Shiraishi [119] pointed out that it is promising to use superbubbles as a tool to infer the evolutionary path of TEs in unassembled genomes.

[5]Some TEs are, not cut but copied through transcription to RNA as an intermediate step.

Chapter 7

Conclusion

In order to fully understand how complex and diverse phenomena of life emerge from the fundamental molecules such as DNA, RNA and proteins, computational approach, together with experimental studies, is essential. In particular, the sequences or structures of biopolymers contain much information about their functions and thus, computational methods to assign functional annotations to sequences/structures have been studied extensively.

Currently, computational molecular biology is in the midst of a paradigm shift driven by the emergence of new technologies. The advent of the so-called next generation sequencing (NGS) technologies made it possible to obtain huge amount of sequences at very low costs. The developments in the molecular dynamics (MD) models and the increase of available computational resources enabled millisecond-scale MD simulation of protein structures. The data provided by these technologies are not only large-scale but also less biased. NGS made whole genome analyses a common practice, e.g., in human genome research and lead to drastically new type of studies such as metagenomics. MD simulations not only predict structures of proteins that are not amenable to experimental methods but also reveal the dynamic aspects of proteins, which were invisible in the past.

Although these new technologies open the possibilities to understand the comprehensive picture of the universe of biological molecules, there are several problems to be resolved to fully achieve that goal. First, in order to analyze large-scale data, more computationally efficient methods are needed. In particular, one source of computational cost is alignment, which is used as a *de facto* standard method for object comparison both in sequence analysis and structure analysis. Second, *de novo* genome assembly methods that can cope with complex and large genomes such as metagenomes or eukaryotic genomes are needed. Although NGS has very high throughput, currently existing DNA sequencing methods cannot read DNA without cutting it into short fragments. In this thesis, we studied these problems.

In Chapter 4 and Chapter 5 we investigated the possibilities of automatic sequence/structure annotation methods that are completely alignment-free. In previous work, alignment-free sequence analyses were mainly studied in the context of phylogenetics. A notable exception was composition-based string kernels and its combination with support vector machines (SVMs) [133]. In particular, the spectrum kernel [73] was particularly promising because it can achieve high annotation accuracy and high computational efficiency at the same time: the

kernel function can be computed in $O(n)$-time by applying the suffix tree [142] in a stark contrast to the $O(n^2)$-time needed for sequence alignment. Therefore, we, starting from the spectrum kernel, tried to design better kernels applying more advanced data structures developed in the string algorithms community.

In particular, in Chapter 4, we proposed the gapped spectrum kernel. This kernel is a "gapped" analogue of the spectrum kernel and is based on an experimental observation that, while the spectrum kernel characterizes strings by contiguous substrings, characterization by non-contiguous substrings seems to lead to more accurate results. The idea of introducing gaps (or wildcards in pattern matching) to the spectrum kernel has previously been studied in the wildcard kernel [71]. The wildcard kernel achieves higher accuracy than that of the spectrum kernel by considering all gap patterns of particular length and weights. However, it was not known what happens if multiple but not all gap patterns are considered. In order to compute the gapped spectrum kernel in time independent of the dimension of feature space, we first proposed the b-suffix array data structure in Chapter 3. This data structure is also related to pattern matching problems that appear in the context of spaced seed search-based sequence homology search. Also, we proposed several non-trivial construction algorithms of the data structure in Chapter 3. In addition to the gapped spectrum kernel computation based on the b-suffix array, we also proposed an algorithm for the prediction phase of SVM that takes time independent of the size of the support vectors. Then, we showed that the sum of all gapped spectrum kernels corresponding to all gap patterns of a given length and weights matches the wildcard kernel. This relation gives a new bound for wildcard kernel computation and prediction. In particular, the prediction algorithm does not depend on the size of the support vectors (Existing algorithm depends linearly on the size of the support vectors.). Last, we experimentally showed that the combination of a few randomly chosen gapped spectrum kernels can predict protein families comparatively accuracy as the wildcard kernel.

On the other hand, in Chapter 5, we studied protein structure analysis. Compared to protein sequences, structures are more directly related to functions and thus, if available, structures should be very valuable clues to identify functions. However, structural alignment is much more computationally expensive than the sequence alignment. We showed how to apply the techniques of alignment-free kernel considered in Chapter 4 to protein structures and thus, avoid the cost of structural alignment. The key there was a novel use of protein contact map, a concept previously used in the context of structural alignment [34]. The proposed kernel function can be computed in time independent of the feature space by applying the two dimensional suffix tree [59]. Also, we showed how to perform the computation of the prediction phase of SVMs in time independent of the size of the support vectors. We experimentally showed that the proposed kernel can predict protein superfamilies comparatively accurately as the most accurate existing method [10] while it runs more than 300 times faster.

Both the results of Chapter 4 and Chapter 5 can be used to sacrifice annotation accuracy slightly to obtain a significant speed-up. In the comprehensive studies of large-scale data such as metagenome sequences or protein motion trajectories, the supply of unannotated data is huge and thus, such a trade is reasonable. In almost all practical settings, the size of annotated data is dwarfed by the size of unannotated data. Therefore, in the context of SVM-based classi-

fication, the prediction phase is more likely to be the computational bottleneck than the learning phase. Our methods are relevant to those cases because the prediction time of our algorithm is independent of the size of the support vectors, which can depend linearly on the size of the database.

We studied genome assembly problem in Chapter 6. Existing assembly algorithms first construct some graph encoding the overlap of reads and then, try to recover the original sequence by traversing the graph. The graph contains many spurious edges introduced by sequencing errors, repeats, diploidy/polyploidy and possibly others. In existing methods, these edges were removed by finding some simple motifs. We noticed that very complex bubble-like substructures (typically a directed acyclic graph with an "entrance" and an "exit"), which we named superbubbles, sometimes emerge in assembly graphs. Ideally, it is desirable to understand the true source of superbubbles and correct them if necessary. However, unlike simple motifs considered in existing work, it is not at all trivial how to find superbubbles in the first place. On the other hand, once we find all superbubbles in the assembly graph, we should be able to apply elaborate algorithms such as alignment to each superbubble instead of the entire graph. To detect superbubbles, we first gave a graph theoretic characterization of superbubbles and clarified some properties of it. For example, the number of superbubbles in a graph is bounded by the number of vertices. Then, we proposed an efficient algorithm to enumerate all superbubbles in a graph. This algorithm takes quadratic time in the worst case but runs very efficiently in practice. We then showed that, under a probabilistic model, the algorithm runs in linear time in expectation.

The most likely application scenario of superbubbles is the noise removal of assembly graphs. Existing genome assemblers often try to resolve complex structures by iteratively resolving simple bubbles. However, there may be structures that cannot be resolved without taking into account related regions all simultaneously and the topological structure of superbubbles seems to be suitable to capture the minimal regions that must be considered together.

Open problems. Several important open problems remain to be resolved to further advance the comprehensive studies of computational molecular biology.

We considered protein sequence analysis in this thesis but DNA sequence analysis is another important line of research. In particular, metagenome analysis is one of the most important applications of alignment-free methods today. A typical problem in metagenome analysis is species estimation. In this problem, sequences (usually NGS reads) from a collection of (possibly unknown) species are given. The goal is to predict which species in the database each sequence was derived from. An input sequence may not belong to any species in the database. In such a case, coarser taxonomical group, e.g., genus, family and so on, should be assigned. Both alignment-based methods [47, 14] and alignment-free methods [4, 143, 57, 105, 17] have been studied. It seems that alignment-free methods are more promising especially for large-scale and highly diverse data because alignment-based methods tend to suffer from higher computational cost and low similarities of input data. Sequence comparison based on k-mer counts/frequencies is one of the main approaches used in alignment-free methods. As we mentioned in Section 4.5, Břinda et al. [17] showed that

use of gapped k-mers instead of contiguous k-mers increase the annotation accuracy. Although existing work indicates that alignment-free methods are very promising approaches, currently there is no *de facto* standard method sequence similarity measures for this problem. Important open problems include establishing good similarity measures for metagenomic sequences and developing efficient algorithms to annotate metagenomic sequences.

In this thesis, we considered annotation of protein MD simulation data. Another important problem related to MD simulation studies is data compression. The outputs of MD simulations, called motion trajectories, are the snapshots of atoms in the system. In typical simulation studies, the simulation time is in the order of microseconds. Marais et al. [84] reported the size of several trajectory files. For example, the trajectory file for 250,005 frames of protein ubiquitin (consisting of 7053 atoms) takes 19.73 GB. Time step used in this simulation was 2 femtoseconds ($= 2 \times 10^{-15}$ seconds). This means that trajectory file produced by 1 microsecond simulation of ubiquitin will take 39,459 GB. Currently, only few compression methods specialized to motion trajectory files exist [89, 84]. More compact representation of motion trajectories would greatly facilitate MD simulation studies.

Alignment-free analysis is potentially relevant to problems outside of computational molecular biology. The essence of an alignment is a correspondence between linear objects. In sequence alignment, each object is a character while in structural alignment, each object is a 3-dimensional coordinate. There are many other linear objects in the world. For example, most time series data can be represented as a sequence of numbers (1-dimensional coordinate). The contour of an image is sometimes represented as a set of points sampled from the contour. Because a contour is a curve, there are natural linear relationships between these points. It seems that alignment can be generalized to any such objects. However, such generalization should inherit the high computational cost of sequence alignment as well. This may become problematic for large-scale applications such as database search (There are huge supplies of both time series data and image data.) and for those cases, alignment-free approach is promising.

References

[1] http://www.repeatmasker.org.

[2] B. Alberts, A. Johnson, J. Lewis, D. Morgan, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Taylor & Francis, 2014.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen. Scalable metagenomic taxonomy classification using a reference genome database. *Bioinformatics*, 29(18):2253–2260, 2013.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, 1999.

[6] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the 47th Symposium on Theory of Computing*, pages 51–58, 2015.

[7] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Research*, 12:177–189, 2002.

[8] E. A. Bender and E. Canfield. The asymptotic number of labeled graphs with given degree sequences. *Journal of Combinatorial Theory, Series A*, 24(3):296–307, 1978.

[9] M. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. 2000.

[10] S. Bhattacharya, C. Bhattacharyya, and N. Chandra. Structural alignment based kernels for protein structure classification. In *Proceedings of the 24th International Conference on Machine Learning*, pages 73–80, 2007.

[11] C. Biémont. A brief history of the status of transposable elements: From junk DNA to major players in evolution. *Genetics*, 186(4):1085–1093, 2010.

[12] P. Bille, I. Gørtz, H. Vildhøj, and S. Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 55(1):41–60, 2014.

[13] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. 2012.

[14] A. Brady and S. Salzberg. PhymmBL expanded: confidence scores, custom databases, parallelization and more. *Nature methods*, 8(5):367–367, 2011.

[15] L. Brankovic, C. S. Iliopoulos, R. Kundu, M. Mohamed, S. P. Pissis, and F. Vayani. Linear-time superbubble identification algorithm. *CoRR*, abs/1505.04019, 2015.

[16] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[17] K. Břinda, M. Sykulski, and G. Kucherov. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, 2015.

[18] J. Chandonia, G. Hon, N. S. Walker, L. Lo Conte, P. Koehl, M. Levitt, and S. E. Brenner. The ASTRAL compendium in 2004. *Nucleic Acids Research*, 32(suppl 1):D189–D192, 2004.

[19] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:1–27, 2011.

[20] Y. Chen, T. Souaiaia, and T. Chen. PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25(19):2514–2521, 2009.

[21] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004.

[22] F. Crick. Central dogma of molecular biology. *Nature*, 227(5258):561–563, 1970.

[23] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

[24] M. Crochemore and G. Tischler. The gapped suffix array: A new index structure for fast approximate matching. In *String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 359–364. 2010.

[25] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[26] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[27] N. K. Fox, S. E. Brenner, and J.-M. Chandonia. SCOPe: Structural classification of proteins-extended, integrating SCOP and ASTRAL data and classification of new structures. *Nucleic Acids Research*, 42(D1):D304–D309, 2014.

[28] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[29] T. Gagie, G. Manzini, and D. Valenzuela. Compressed spaced suffix arrays. In *Proceedings of the 2nd International Conference on Algorithms for Big Data (ICABD)*, volume 1146, pages 37–45, 2014.

[30] Z. Galil and R. Giancarlo. Speeding up dynamic programming with application to molecular biology. *Theor. Comput. Sci.*, 64(1):107–118, 1989.

[31] G. Gaston. Efficient searching of text and pictures. Technical Report OED-88-02, University of Waterloo, 1988.

[32] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995.

[33] A. Godzik. Metagenomics and the protein universe. *Current Opinion in Structural Biology*, 21(3):398 – 403, 2011.

[34] D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, pages 512–521, 1999.

[35] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[36] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[37] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[38] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[39] H. Hasegawa and L. Holm. Advances and pitfalls of protein structural alignment. *Current Opinion in Structural Biology*, 19(3):341–348, 2009.

[40] L. Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *Journal of Molecular Biology*, 233(1):123 – 138, 1993.

[41] L. Holm and C. Sander. Mapping the protein universe. *Science*, 273(5275):595–602, 1996.

[42] L. Holm and C. Sander. Dali/FSSP classification of three-dimensional protein folds. *Nucleic Acids Research*, 25(1):231–234, 1997.

[43] L. Holm and C. Sander. Touring protein fold space with dali/FSSP. *Nucleic Acids Research*, 26(1):316–319, 1998.

[44] W.-K. Hon, T.-W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments*, pages 31–38, 2004.

[45] S. Horwege, S. Lindner, M. Boden, K. Hatje, M. Kollmar, C.-A. Leimeister, and B. Morgenstern. Spaced words and kmacs: fast alignment-free sequence comparison based on inexact word matches. *Nucleic Acids Research*, 42(W1):W7–W11, 2014.

[46] X. Huang and S. P. Yang. Generating a genome assembly with PCAP. *Current Protocols in Bioinformatics*, Unit 11.3, 2005.

[47] D. H. Huson, S. Mitra, H.-J. Ruscheweyh, N. Weber, and S. C. Schuster. Integrative analysis of environmental sequences using MEGAN4. *Genome Research*, 21(9):1552–1560, 2011.

[48] C. S. Iliopoulos and M. S. Rahman. Pattern matching algorithms with don't cares. In *Proc. 33rd SOFSEM*, pages 116–126, 2007.

[49] T. Jaakkola, M. Diekhans, and D. Haussler. Using the Fisher kernel method to detect remote protein homologies. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology*, pages 149–158, 1999.

[50] B. Jackson, M. Regennitter, X. Yang, P. Schnable, and S. Aluru. Parallel *de novo* assembly of large genomes from high-throughput short reads. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, pages 1–10, 2010.

[51] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[52] T. Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 217–226, 2006.

[53] J. Jurka. Repeats in genomic DNA: mining and meaning. *Current Opinion in Structural Biology*, 8(3):333 – 337, 1998.

[54] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, Nov. 1962.

[55] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, pages 943–955, 2003.

[56] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. 2001.

[57] J. Kawulok and S. Deorowicz. CoMeta: Classification of metagenomes using *k*-mers. *PLoS ONE*, 10(4):e0121453, 04 2015.

[58] W. J. Kent and A. M. Zahler. Conservation, regulation, synteny, and introns in a large-scale *C. briggsae-C. elegans* genomic alignment. *Genome Research*, 10(8):1115–1125, 2000.

[59] D. K. Kim, J. C. Na, J. S. Sim, and K. Park. Linear-time construction of two-dimensional suffix trees. *Algorithmica*, 59(2):269–297, 2011.

[60] S. Knerr, L. Personnaz, and G. Dreyfus. Single-layer learning revisited: a stepwise procedure for building and training a neural network. In *Neurocomputing*, volume 68, pages 41–50. Springer Berlin Heidelberg, 1990.

[61] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. 2003.

[62] E. V. Koonin, Y. I. Wolf, and G. P. Karev. The structure of the protein universe and genome evolution. *Nature*, 420(6912):218–223, 2002.

[63] A. Krogh, M. Brown, I. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235(5):1501 – 1531, 1994.

[64] R. Kuang, E. Ie, K. Wang, K. Wang, M. Siddiqi, Y. Freund, and C. Leslie. Profile-based string kernels for remote homology detection and motif extraction. In *Computational Systems Bioinformatics Conference*, 2004.

[65] P. P. Kuksa, P.-H. Huang, and V. Pavlovic. Scalable algorithms for string kernels with inexact matching. In *Advances in Neural Information Processing Systems 21*, pages 881–888. 2009.

[66] S. Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999.

[67] T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu. Space efficient indexes for string matching with don't cares. In *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 846–857. 2007.

[68] G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 220–230, 1986.

[69] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[70] C.-A. Leimeister, M. Boden, S. Horwege, S. Lindner, and B. Morgenstern. Fast alignment-free sequence comparison using spaced-word frequencies. *Bioinformatics*, 30(14):1991–1999, 2014.

[71] C. Leslie and R. Kuang. Fast kernels for inexact string matching. In *Learning Theory and Kernel Machines*, volume 2777 of *Lecture Notes in Computer Science*, pages 114–128. 2003.

[72] C. S. Leslie, E. Eskin, A. Cohen, J. Weston, and W. S. Noble. Mismatch string kernels for discriminative protein classification. *Bioinformatics*, 20(4):467–476, 2004.

[73] C. S. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proceedings of Pacific Symposium on Biocomputing*, pages 566–575, 2002.

[74] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[75] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter II: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 02(03):417–439, 2004.

[76] R. Li, H. Zhu, J. Ruan, W. Qjan, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, H. Yang, and J. Wang. *De novo* assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20:265–272, 2010.

[77] H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li. ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

[78] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[79] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *J. Mach. Learn. Res.*, 2:419–444, 2002.

[80] K. L. Lorick, J. P. Jensen, S. Fang, A. M. Ong, S. Hatakeyama, and A. M. Weissman. Ring fingers mediate ubiquitin-conjugating enzyme (E2)-dependent ubiquitination. *Proceedings of the National Academy of Sciences*, 96(20):11364–11369, 1999.

[81] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[82] I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea, L. Williams, S. Young, C. Nusbaum, and D. B. Jaffe. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology*, 10(R103), 2009.

[83] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[84] P. Marais, J. Kenwood, K. C. Smith, M. M. Kuttel, and J. Gain. Efficient compression of molecular dynamics trajectory files. *Journal of Computational Chemistry*, 33(27):2131–2141, 2012.

[85] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

[86] E. Mayoraz and E. Alpaydin. Support vector machines for multi-class classification. In *Engineering Applications of Bio-Inspired Artificial Neural Networks*, volume 1607 of *Lecture Notes in Computer Science*, pages 833–842. 1999.

[87] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[88] C. E. Metz. Basic principles of ROC analysis. *Seminars in Nuclear Medicine*, 8(4):283 – 298, 1978.

[89] T. Meyer, C. Ferrer-Costa, A. Pérez, M. Rueda, A. Bidon-Chanal, F. J. Luque, C. A. Laughton, , and M. Orozco. Essential dynamics: A tool for efficient trajectory compression and management. *Journal of Chemical Theory and Computation*, 2(2):251–258, 2006.

[90] W. Miller and E. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50(2):97–120, 1988.

[91] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[92] B. Morgenstern, B. Zhu, S. Horwege, and C.-A. Leimeister. Estimating evolutionary distances from spaced-word matches. In *Algorithms in Bioinformatics*, volume 8701 of *Lecture Notes in Computer Science*, pages 161–173. 2014.

[93] D. R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[94] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. SCOP: A structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247(4):536 – 540, 1995.

[95] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[96] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. J. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M. Rubin, M. D. Adams, and J. C. Venter. A whole-genome assembly of *Drosophila*. *Science*, 287:2196–2204, 2000.

[97] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[98] A. Niida. personal communication, August 2015.

[99] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference*, pages 193–202, 2009.

[100] S. Nurk, A. Bankevich, D. Antipov, A. Gurevich, A. Korobeynikov, A. Lapidus, A. Prjibelsky, A. Pyshkin, A. Sirotkin, Y. Sirotkin, R. Stepanauskas, J. McLean, R. Laskin, R. Stepanasukas, J. McLean, R. Laskin, S. R. Clingenpeel, T. Woyke, G. Tesler, M. A. Alekseyev, and P. A. Pevzner. Assembling genomes and mini-metagenomes from highly chimeric reads. *Proc. 17th International Conference on Research in Computational Molecular Biology*, pages 158–170, 2013.

[101] T. Onodera, K. Sadakane, and T. Shibuya. Detecting superbubbles in assembly graphs. In *Algorithms in Bioinformatics*, volume 8126 of *Lecture Notes in Computer Science*, pages 338–348. 2013.

[102] T. Onodera and T. Shibuya. An index structure for spaced seed search. In *Algorithms and Computation*, volume 7074 of *Lecture Notes in Computer Science*, pages 764–772. 2011.

[103] T. Onodera and T. Shibuya. The gapped spectrum kernel for support vector machines. In *Machine Learning and Data Mining in Pattern Recognition*, volume 7988 of *Lecture Notes in Computer Science*, pages 1–15. 2013.

[104] C. Orengo, T. Flores, W. Taylor, and J. Thornton. Identification and classification of protein fold families. *Protein Engineering*, 6(5):485–500, 1993.

[105] R. Ounit, S. Wanamaker, T. Close, and S. Lonardi. CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC Genomics*, 16(1):236, 2015.

[106] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[107] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98:9748–9753, 2001.

[108] A. Poleksic. Algorithms for optimal protein structure alignment. *Bioinformatics*, 25(21):2751–2756, 2009.

[109] A. L. Price, N. C. Jones, and P. A. Pevzner. *De novo* identification of repeat families in large genomes. *Bioinformatics*, 21(suppl 1):i351–i358, 2005.

[110] J. Qiu, M. Hue, A. Ben-Hur, J.-P. Vert, and W. S. Noble. A structural alignment kernel for protein structures. *Bioinformatics*, 23(9):1090–1098, 2007.

[111] A. R. Rhoads and F. Friedberg. Sequence motifs for calmodulin recognition. *The FASEB Journal*, 11(5):331–40, 1997.

[112] D. B. Rusch, A. L. Halpern, G. Sutton, K. B. Heidelberg, S. Williamson, S. Yooseph, D. Wu, J. A. Eisen, J. M. Hoffman, K. Remington, K. Beeson, B. Tran, H. Smith, H. Baden-Tillson, C. Stewart, J. Thorpe, J. Freeman, C. Andrews-Pfannkoch, J. E. Venter, K. Li, S. Kravitz, J. F. Heidelberg, T. Utterback, Y.-H. Rogers, L. I. Falcón, V. Souza, G. Bonilla-Rosso, L. E. Eguiarte, D. M. Karl, S. Sathyendranath, T. Platt, E. Bermingham, V. Gallardo, G. Tamayo-Castillo, M. R. Ferrari, R. L. Strausberg, K. Nealson, R. Friedman, M. Frazier, and J. C. Venter. The *Sorcerer II* global ocean sampling expedition: Northwest atlantic through eastern tropical pacific. *PLoS Biol*, 5(3):e77, 03 2007.

[113] M. Sahli and T. Shibuya. Arapan-S: a fast and highly accurate whole-genome assembly software for viruses and small genomes. *BMC Research Notes*, 5(243), 2012.

[114] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

[115] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):pp. 787–793, 1974.

[116] A. Severyn and A. Moschitti. Large-scale support vector learning with structural kernels. In *Machine Learning and Knowledge Discovery in Databases*, volume 6323 of *Lecture Notes in Computer Science*, pages 229–244. 2010.

[117] J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[118] I. N. Shindyalov and P. E. Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998.

[119] Y. Shiraishi. personal communication, August 2015.

[120] C. J. A. Sigrist, L. Cerutti, N. Hulo, A. Gattiker, L. Falquet, M. Pagni, A. Bairoch, and P. Bucher. Prosite: A documented database using patterns and profiles as motif descriptors. *Briefings in Bioinformatics*, 3(3):265–274, 2002.

[121] C. J. A. Sigrist, E. de Castro, L. Cerutti, B. A. Cuche, N. Hulo, A. Bridge, L. Bougueleret, and I. Xenarios. New and continuing developments at prosite. *Nucleic Acids Research*, 41(D1):D344–D347, 2013.

[122] I. Sillitoe, T. E. Lewis, A. Cuff, S. Das, P. Ashford, N. L. Dawson, N. Furnham, R. A. Laskowski, D. Lee, J. G. Lees, S. Lehtinen, R. A. Studer, J. Thornton, and C. A. Orengo. CATH: comprehensive structural and functional annotations for genome sequences. *Nucleic Acids Research*, 43(D1):D376–D381, 2015.

[123] J. T. Simpson and R. Durbin. Efficient *de novo* assembly of large genomes using compressed data structures. *Genome Research*, 2011.

[124] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, and S. J. Jones. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19:1117–1123, 2009.

[125] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[126] A. J. Smola and S. V. N. Vishwanathan. Fast kernels for string and tree matching. In *Advances in Neural Information Processing Systems 15*, pages 585–592. 2003.

[127] G. D. Stormo, T. D. Schneider, L. Gold, and A. Ehrenfeucht. Use of the ‘ perceptron ’ algorithm to distinguish translational initiation sites in e. coli. *Nucleic Acids Research*, 10(9):2997–3011, 1982.

[128] W.-K. Sung, K. Sadakane, T. Shibuya, A. Belorkar, and I. Pyrogova. An $O(m \log m)$-time algorithm for detecting superbubbles. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, PP(99):1–1, 2014.

[129] W. R. Taylor and C. A. Orengo. Protein structure alignment. *Journal of Molecular Biology*, 208(1):1 – 22, 1989.

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

[131] N. Välimäki and E. Rivals. Scalable and versatile k-mer indexing for high-throughput sequencing data. In Z. Cai, O. Eulenstein, D. Janies, and D. Schwartz, editors, *Bioinformatics Research and Applications*, volume 7875 of *Lecture Notes in Computer Science*, pages 237–248. Springer Berlin Heidelberg, 2013.

[132] E. L. van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes. Ten years of next-generation sequencing technology. *Trends in Genetics*, 30(9):418 – 426, 2014.

[133] V. N. Vapnik and V. Vapnik. *Statistical learning theory*, volume 1. Wiley New York, 1998.

[134] M. Vassura, L. Margara, P. Di Lena, F. Medri, P. Fariselli, and R. Casadio. Reconstruction of 3D structures from protein contact maps. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 5(3):357–367, July 2008.

[135] S. Vinga. Biological sequence analysis by vector-valued functions: revisiting alignment-free methodologies for DNA and protein classification. *Advanced Computational Methods for Biocomputing and Bioimaging*, pages 71–107, 2007.

[136] S. Vinga. Editorial: Alignment-free methods in computational biology. *Briefings in Bioinformatics*, 15(3):341–342, 2014.

[137] T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.

[138] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[139] R. A. Wagner and R. Lowrance. An extension of the string-to-string correction problem. *J. ACM*, 22(2):177–183, Apr. 1975.

[140] C. Wang and S. D. Scott. New kernels for protein structural motif discovery and function classification. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 940–947, 2005.

[141] J. D. Watson and F. H. Crick. Molecular structure of nucleic acids. *Nature*, 171(4356):737–738, 1953.

[142] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[143] D. Wood and S. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, 2014.

[144] M. Yokouchi, T. Kondo, A. Houghton, M. Bartkiewicz, W. C. Horne, H. Zhang, A. Yoshimura, and R. Baron. Ligand-induced ubiquitination of the epidermal growth factor receptor involves the interaction of the c-Cbl RING finger and UbcH7. *Journal of Biological Chemistry*, 274(44):31707–31712, 1999.

[145] D. R. Zerbino and E. Birney. Velvet: algorithms for *de novo* short read assembly using de Bruijn graphs. *Genome Research*, 18:821–829, 2008.

[146] Y. Zhang and J. Skolnick. Scoring function for automated assessment of protein structure template quality. *Proteins: Structure, Function, and Bioinformatics*, 57(4):702–710, 2004.