Research on Fast Algorithms for Comparison and Indexing of Biological Sequence Information
生物配列情報の比較と検索のための高速なアルゴリズムの研究

by
Tetsuo Shibuya
渋谷 哲朗

A Dissertation

Submitted to

The Graduate School of Information Science
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree for Doctor of Science

Thesis Supervisor: Satoru Miyano　　宮野 悟
Title: Professor of Information Science

ABSTRACT

As biological sequence data such as DNA, RNA and protein sequences are increasing with an accelerated pace by the recent epoch-making innovation of sequencing technology, many problems in molecular biology require fast algorithms that enable efficient processing of such large data sets. Sequence comparison and indexing techniques, such as sequence alignment algorithms, similarity search algorithms for sequence databases and motif discovery algorithms are very important and fundamental techniques for solving the problems in the field. These techniques are also called pattern matching techniques. In this thesis, we propose various fast algorithms for sequence comparison and indexing so that we can solve very important problems in molecular biology. Moreover, we also examine the property and efficiency of these algorithms through experiments using actual large sequence data sets for most of the problems.

First, we deal with the multiple alignment problem, which is one of the most frequently-used and important sequence comparison techniques in molecular biology. According to biologists, the optimal alignment based on a computational model is not always the biologically most significant alignment. To solve this problem, we propose two flexible and efficient approaches. One possible approach is to provide many suboptimal alignments as alternatives for the optimal one, but there are so many similar suboptimal solutions and it is difficult to find a desired solution among them. Hence we discuss what kind of suboptimal alignment is unnecessary to enumerate, and propose a new fast algorithm to reduce them. Biologists also often say that the obtained optimal solution with fixed parameters is not always the biologically best alignment. We also discuss a technique called parametric multiple alignment as the second approach, which can enumerate efficiently all the solutions as some parameter varies in a parametric space.

We next deal with a clustering problem in which we cluster sequences of a full-length cDNA library into groups of alternative splice form candidates, using a variant of alignment technique called a spliced alignment. This is a very important problem because its results are very useful for the biological study of splice sites or alternative splicing, which is one of the hottest topics in molecular biology of today. We can also use the obtained clusters as the training set for gene finding which is the next subject of this thesis. We propose a fast and accurate algorithm for this problem, and examine its efficiency and property through experiments using FANTOM, a large mouse cDNA library.

We then deal with the gene finding problem which is one of the most important problems in molecular biology. There have been many methods proposed for solving this problem, which can be categorized into two groups, statistical methods and similarity search-based methods. We propose a new gene identification scheme that combines the best characteristics of these two groups. Our method determines gene candidates by using the statistical behavior of matching patterns of a large pattern database called the Bio-Dictionary. For this problem, we propose an efficient pattern indexing scheme for matching patterns of the database, with which we succeeded in achieving the competitive speed against the statistical methods. We also demonstrate the performance of our algorithm through experiments using genomes of many prokaryotic organisms.

RNA structures are known to play a very important role in the determination of their properties. We then discuss efficient techniques for finding frequent secondary structures from a set of RNA sequences or a set of RNA structures by generalizing a data structure called a suffix tree. The suffix tree is a very useful and important indexing data structure for searching for substrings or finding frequent substring patterns of

texts. We first discuss what kind of RNA's substrings can have the same 3-D structure, and then generalize suffix trees to discover such patterns. We also propose an on-line algorithm for constructing the generalized data structure. It is known that RNA secondary structures can be represented by tree structures. Suffix trees can be generalized also for discovering patterns from such tree structures. We also propose a new algorithm for constructing the generalized suffix trees for trees, whose computation time bound is better than the best-known algorithm.

Through all these topics, we show that pattern matching techniques of sequence comparison and indexing play an important role in the problems of computational molecular biology and efficiently solve them.

論 文 要 旨

近年、分子生物学において、配列解析技術などの急速な進展にともない、DNA、RNA、蛋白質といった生物配列情報が爆発的に増大しており、そのためそのような大量のデータを効率的に処理するための様々な高速なアルゴリズムが必要となってきている。これらの問題において最も重要な基盤技術の一つは、配列のアラインメント技術やモチーフ発見技術、DNA や蛋白質などの生物データベースの検索技術などの配列比較及び検索の技術である。これらの技術はパタンマッチング技術とも呼ばれる。本論文では、そのような配列比較や検索などのパタンマッチングの問題に対し高速なアルゴリズムを提供することで、分子生物学上非常に重要ないくつかの問題を効率的に解決する。さらに、それらのアルゴリズムの多くについて、実際に大規模な生物学的データを用いた実験を通じて性質及び性能を検証する。

本論文では、まず、分子生物学全般で極めて多用されている最も基本的かつ重要な配列比較技術の一つであるマルチプル・アラインメント問題をとりあげる。この問題には、計算論的に得られる最適解が生物学的には最適ではない、という問題がある。これに対し本論文では２つの手法を提案する。まず一つ目の手法として、多くの準最適解を出力することで代替の解を提供するという解決法を考えられるが、そのような準最適解はわずかに異なるだけの解が極めて多くあり、欲しい解を探すのが困難である。そこで、本論文ではいかなるアラインメントが列挙する必要がないかを論じ、必要なアラインメントだけを高速に列挙するアルゴリズムを提案する。また、このアラインメント問題においては、スコア行列などのパラメータを固定して求めた従来の最適解は生物学的に最適であるとは限らないともいわれている。そこで本論文ではさらに二つ目の手法として、パラメータを変化させた時の解をすべて得るための効率的なパラメータ空間の探索技法について議論する。

次に本論文では、先に述べたアラインメントの一変形であるスプライスト・アラインメントという手法を用いて、cDNA ライブラリに含まれる配列集合を選択的スプライス集合と呼ぶ集合にクラスタリングするという問題を扱う。最近ヒトゲノムの遺伝子数と実際の蛋白質数の違いがわかってきたことで非常に脚光を浴びている選択的スプライシングの研究において、このクラスタリング結果は極めて有用である。また、本論文でも次に扱う遺伝子発見のツール等の学習セットとしても有用である。本論文では、この問題を解くための高速かつ正確なアルゴリズムを提案する。さらに、マウスの大規模な cDNA ライブラリである FANTOM を用いた実験を通して、このアルゴリズムの性能を検証する。

さらに、分子生物学研究上最も重要な問題の一つである遺伝子発見あるいは遺伝子同定と呼ばれる問題を扱う。この問題に対しては従来から様々な手法が提案されてきているが、それらは主に 2 種類に分類される。一つは隠れマルコフモデルなどの統計学的な手法を用いる方法であり、もう一つは異なる種の間の配列比較や

データベースからの類似配列検索に基づくパタンマッチング的手法である。前者は統計的な傾向が種によって異なるため、自身の種かそれに近い種の十分な学習セットがない場合うまく推定できないという問題がある。一方、後者は類似しているものがないものについては全く見つけることができないなどの問題がある。それに対し、本論文では、両者の長所をあわせ持つような従来にはない全く新しい手法を提案する。この手法は、きわめて大規模なパタンデータベースのパタンを検索し、それらのパタンの統計的振舞いに基づいて遺伝子発見を行なう。この方法は、全く新しい学習セットのないような種のゲノムに対しても遺伝子を従来手法以上に正確に推定することが可能である。また、一般的にはパタンマッチング手法による遺伝子発見は統計的な手法より計算時間が大きいことが多いという問題点があるが、この手法では極めて大きなパタンデータベースからの高速検索を行なうための新しいパタン検索アルゴリズムを提案することで、一般的な統計的手法と比べても十分高速な遺伝子同定を実現している。本論文では、さらにこの手法を実際に様々な原核生物ゲノムに適用し検証を行なう。

RNA などの生物配列は、それが形作る立体構造がその性質に極めて影響を及ぼすことが知られている。本論文では、最後に、RNA などの生物配列や 2 次構造データベースにおいて、接尾辞木というデータ構造を用いて頻出する構造を高速に捜し出す手法を提案する。接尾辞木は、テキストなどから頻出文字列を発見したり、キーワードを検索する際に非常に重要かつ有用な検索のためのデータ構造である。本論文では、まず、同様の構造を持ち得るような RNA の部分配列とは何かを考察し、それを考慮するように接尾辞木を一般化したデータ構造とそれを構築する高速なオンライン・アルゴリズムを提案する。また、RNA の 2 次構造は木構造で表現できることが知られている。さらに、そのような木構造から様々なパタンを検索あるいは発見することができる木に対する接尾辞木というものも知られているが、本論文ではそれに対する既存の最良の計算量より良いアルゴリズムを提案する。

本論文では、これらのアルゴリズムを通じて、配列比較や検索といった効率的なパタンマッチング的技法を用いることで、いかに分子生物学における様々な重要な問題を解決することができるかを示す。

Acknowledgments

It is a great pleasure for me to acknowledge a lot of helpful and useful advice and suggestions of Prof. Satoru Miyano, the thesis committee chairperson, for the development of this thesis. I am very grateful also to Prof. Hiroshi Imai, who was the supervisor when I was a master course student, and the members of his laboratory for many useful discussions and comments on various subjects of this thesis.

I would like to express my sincere appreciation to Dr. Isidore Rigoutsos, who was the advisor to me when I stayed at IBM Watson Research Center in 2000, for discussions and a great deal of useful advice on the gene finding research and other research areas. With him, we succeeded in developing a tool called BDGF for prokaryotic gene finding. I also thank Dr. Stephen Chin-bow and Dr. Tien Huynh for building a Web user interface for the BDGF. I would also like to thank Prof. Akihiko Konagaya and Dr. Christian Schönbach of RIKEN, who are co-authors of the paper on cDNA clustering, for fruitful discussion on cDNA-related projects. I would also appreciate the helpful and useful advice and encouragement by people at and around IBM Tokyo Research Laboratory, especially Mr. Hisashi Kashima, Dr. Kazuyoshi Hidaka, and Prof. Michael McDonald.

I am grateful to Prof. Takeshi Tokuyama of Tohoku University for giving me inspiration on various algorithms through discussions. I would like to thank Dr. Kunihiko Sadakane of Tohoku University for very various discussions on various subjects such as pattern matching and data compression. I would like to thank Dr. Hiroki Sasaki of National Cancer Center and Dr. Shinichi Oka of International Medical Center of Japan for discussions on various biological issues.

Finally, I would like to thank the members of the committee for my thesis, Prof. Satoru Miyano (to thank again), Prof. Masaki Hagiya, Prof. Sinichi Morishita, Prof. Tetsushi Yada and Prof. Tatsuya Akutsu, for various useful comments and suggestions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 Pattern Matching Algorithms in Molecular Biology

The innovation of experimental techniques of molecular biology in these days have lead to the breathtaking increase in the sizes of various biological databases, and how we can manage to deal with such large data sets efficiently is one of the greatest issues that both computer scientists and molecular biologists are faced with. Matching and searching on discrete structures including simple text strings have been very important and pervasive issues in computer science. As biological sequences like DNA and protein, that are the most important elements in molecular biology, are known to be represented by simple strings of nucleic acids or amino acids, it is very reasonable that such pattern matching researches are highlighted in this field. Hence this thesis focuses on the development of pattern matching-based fast techniques for solving several very important biological problems.

There are two very important main categories of matching problems in computational molecular biology: sequence comparison and indexing. If we have two or more relevant biological sequences, we need to compare them to see what kind of relevance there is among them, which is the former problem. There are tremendously many methods for such purpose, among which the most important and fundamental technique is the sequence alignment [73, 110, 160]. For the sequence alignment problem, a dynamic programming (DP) algorithm that searches for the shortest path in a grid-like graph is the most fundamental algorithm. There are also other important comparison problems, such as discovery of frequent patterns from a set of sequences for which the suffix tree [50, 102, 155, 163] and the suffix array [100] are very important and fundamental data structures.

The latter category includes various database search problems. If we have newly sequenced data of DNA or protein, the first thing we do is searching for similar sequences or substrings in existing databases, or searching for known motifs (sequence patterns). The problem is how to preprocess the database so as to search such matching database entries fast. The most famous tools for similarity search on sequence databases are the FASTA [114] and the BLAST [8] algorithms. The suffix tree and the suffix array are very important and fundamental data structures for indexing too.

This thesis deeply deals with pattern matching algorithms in both of the two categories. Figure 1.1 shows the algorithmic categories and relations between the chapters of this thesis. In Chapter 3, we work on how

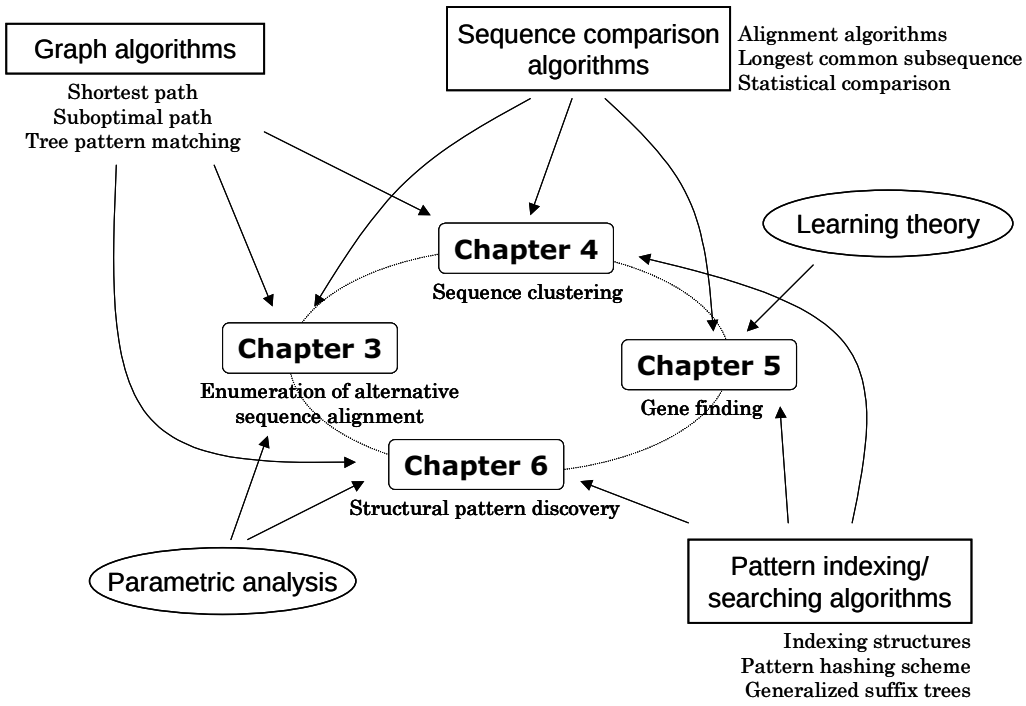


Figure 1.1: Algorithmic topics of this thesis.

to find effectively a good solution from a large set of possible solutions on the multiple sequence alignment problem. In Chapter 4, we deal with a clustering problem of biological sequences based on both accurate sequence comparison and sequence indexing. In Chapter 5, we present a new method for finding genes from a given large DNA genomic sequence using a large-scale pattern matching technique. In Chapter 6, we present new techniques to mine hidden frequent RNA structures from a set of sequences or RNA structures based on suffix trees.

We deal with alignment algorithms and several other sequence comparison techniques in Chapters 3, 4 and 5. As for indexing algorithms, we study a variant of similarity search problem in Chapter 4, propose a new pattern indexing algorithm in Chapter 5, and generalize suffix trees for the use of structural analysis of biological sequences in Chapter 6. Other than these matching algorithms, graph algorithms like shortest path algorithms and tree pattern matching algorithms take an important role in this thesis as shown in the figure.

1.1.2 Molecular Biology Topics in this Thesis

The most important backbone concept in molecular biology is the *central dogma*. It states that DNA carries the genetic information which is transcribed to RNA and subsequently translated to protein. How this mechanism works is the most important research issue in this field.

The regions of a DNA sequence that are transcribed to RNAs are called genes. It is very difficult to find such regions correctly without any wet experiments, which is one of the most challenging data mining

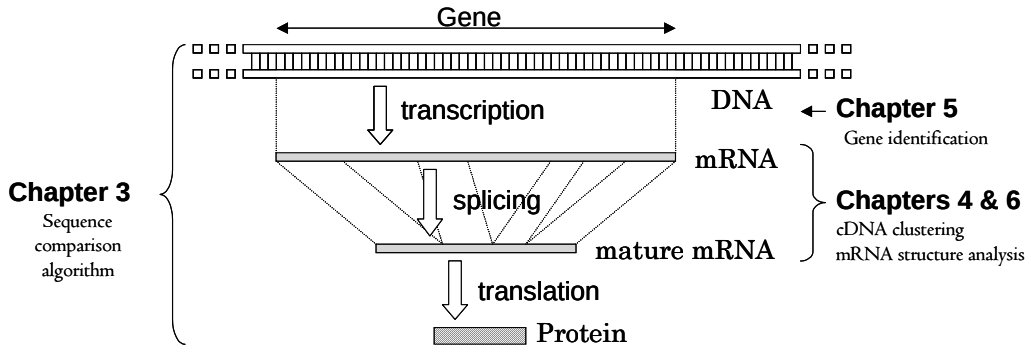


Figure 1.2: Biological topics of this thesis.

subjects in the computational molecular biology. The RNAs are known to be spliced before translation to proteins in eukaryotic genomes. It is also a very important problem to analyze how they are spliced. Due to this splicing mechanism, the variety of proteins is much larger than the number of genes in human genome. The 3-D (or lower-level) structures of these biological sequences (especially proteins and RNAs) are also very important research issues, because such structures are said to determine their functions.

Figure 1.2 is a simple picture that describes the central dogma and relations of the topics in this thesis to the dogma. In Chapter 5, we deal with the gene identification problem stated above. In Chapter 4, we present work on clustering cDNA sequences (that are DNA sequences artificially translated from mRNA sequences), which can contribute to the research on splicing mechanisms. We also present methods to analyze or mine RNA structures in Chapter 6. The alignment algorithms presented in Chapter 3 can be used for analyzing any of these sequences.

1.2 Our Contribution

In this section, we briefly describe the four main topics of this thesis. All the four topics utilize pattern matching techniques and apply them to very important problems in molecular biology. All in all, we contribute not only to the research of algorithms in computer science but also to the genomic research in molecular biology.

1.2.1 Efficient Enumeration of Alternative Multiple Sequence Alignments

The multiple sequence alignment problem is applicable and important in various fields in molecular biology such as the prediction of three dimensional structures of proteins and the inference of phylogenetic tree. However, the optimal alignment based on the scoring criterion is not always the biologically most significant alignment. In Chapter 3, we propose two flexible and efficient approaches to solve this problem. One approach is to provide many suboptimal alignments as alternatives for the optimal one. Although this problem is well-studied for the alignment of two sequences, it has been considered impossible to investigate such suboptimal alignments of more than two sequences because of the enormous difficulty of the problem.

We propose algorithms for enumeration of suboptimal alignments based on Eppstein's algorithm. We also discuss what kind of suboptimal alignment is unnecessary to enumerate and propose an efficient enumeration algorithm to enumerate only necessary alignments. The other approach is parametric analysis. The obtained optimal solution with fixed parameters such as gap penalties is not always the biologically best alignment. Thus, it is required to vary parameters and check how the optimal alignments change. The way to vary parameters has been studied well on the problem of two sequences, but not in the multiple alignment problem because of the difficulty of computing the optimal solution. We present techniques for the parametric multiple alignment problem and examine the features of obtained alignments by various parametric analyses. For both approaches, this thesis performs experiments on various groups of actual protein sequences and examines the efficiency of these algorithms and property of sequence groups.

These results appeared primarily in [141]. Note that there are several related conference papers [137, 138, 139, 140]. As for related work, we also applied variants of these algorithms to geographic databases, whose results appeared primarily in [136, 142, 143]. In these papers, we utilize the A* algorithm, to which our work [131, 134] is related. Note also that we published another paper [133] on a variant of sequence alignment problem.

1.2.2 Accurate cDNA Clustering based on Spliced Alignment

cDNA library is a database of cDNAs (*i.e.* DNA sequences translated from mRNAs) expressed in some organism. An alternative splice form is a group of mRNAs (or cDNAs) that are transcribed from the same gene region. A problem of finding alternative splice forms from a cDNA library is very important not only for the analyses of the splicing mechanism but also for reducing the costs of cDNA-related experiments including the construction of cDNA libraries. Current clustering algorithms for cDNAs tend to produce too many incorrect clusters containing splice form candidates. In Chapter 4, we develop a new efficient and accurate algorithm to cluster sequences of a full-length cDNA library such as FANTOM into alternative splice form candidates. Our algorithm is based on a spliced sequence alignment algorithm, which is a variant of an ordinary dynamic programming algorithm for aligning a sequence with its spliced more mature sequence. It requires $O(nm)$ time for checking a pair of sequences where n and m are the lengths of the two sequences. Since the time bound is too large to perform all-pair comparison for a large set of sequences, we developed a new technique that reduces the computation time without decreasing the accuracy of the output clusters. Our algorithm was applied to 21,076 mouse cDNA sequences of the FANTOM1.10 database to examine its performance and accuracy. In these experiments, we achieved about 4,000 to 20,000-fold speedup against a naive all-pairs comparison algorithm. Moreover, without using any information of the mouse genome sequence data or any gene data in public databases, we succeeded in listing 87-89% of all the clusters that biologists have annotated manually.

This result appeared primarily in [145].

1.2.3 Dictionary-driven Prokaryotic Gene Finding

Gene identification, also known as gene finding or gene recognition, is among the important problems of molecular biology that has been receiving increasing attention with the advent of large scale sequencing

projects. Previous strategies for solving this problem can be categorized into essentially two schools of thought: one school employs sequence composition statistics, whereas the other relies on database similarity searches. In Chapter 5, we propose a new gene identification scheme that combines the best characteristics from each of these two schools based on a very fast new pattern indexing algorithm. In particular, our method determines gene candidates among the ORFs that can be identified in a given DNA strand through the use of the Bio-Dictionary, a database of patterns that covers essentially all of the currently available sample of the natural protein sequence space. Our approach relies entirely on the use of redundant patterns as the agents on which the presence or absence of genes is predicated and does not employ any additional evidence, e.g. ribosome-binding site signals. BDGF (for Bio-Dictionary Gene Finder), the algorithm's implementation, is a single computational engine able to handle the gene identification task across distinct archaeal and bacterial genomes. The engine exhibits performance that is characterized by simultaneous very high values of sensitivity and specificity, and a high percentage of correctly predicted start sites. Using a collection of patterns derived from an old (June 2000) release of the SwissProt/TrEMBL database that contained 451,602 proteins and fragments, we demonstrate our method's generality and capabilities through an extensive analysis of 17 complete archaeal and bacterial genomes.

This result appeared primarily in [144]. Note that we also published another related paper on pattern indexing algorithms based on the compressed suffix array data structure [124].

1.2.4 Suffix Tree Data Structures for RNA Structure Analyses

In molecular biology, it is said that two biological sequences tend to have similar properties if they have similar 3-D structures. Hence, it is very important to find not only similar sequences in the string sense, but also structurally similar sequences from databases. In Chapter 6, we discuss several data structures based on suffix trees for the analyses of RNA structural patterns.

We first propose a new data structure that is a generalization of a parameterized suffix tree (p-suffix tree for short) introduced by Baker. The new data structure can be used for finding structurally related patterns of RNA or single-stranded DNA. Furthermore, we propose an $O(n(\log|\Sigma| + \log|\Pi|))$ on-line algorithm for constructing it, where n is the sequence length, $|\Sigma|$ is the size of the normal alphabet, and $|\Pi|$ is that of the alphabet called "parameter," which is related to the structure of the sequence. Our algorithm achieves optimal linear time when it is used to analyze RNA and DNA sequences. Furthermore, as an algorithm for constructing the p-suffix tree, it is the first on-line algorithm, though the computing bound of our algorithm is same as that of Kosaraju's best-known algorithm. The results of computational experiments using actual RNA and DNA sequences are also given to demonstrate our algorithm's practicality.

RNA structures can be represented by tree data structures. The suffix tree of a tree is a generalization of the suffix tree of a string for the analysis of tree structures. The best-known algorithm for constructing it is an $O(n\log|\Sigma|)$ time algorithm where n is the size of the target tree and $|\Sigma|$ is the alphabet size, which requires $O(n\log n)$ time if $|\Sigma|$ is large. We improve this bound by giving an optimal linear-time algorithm for integer alphabets. We also propose another new data structure called the Bsuffix tree, which can also be used for some tree pattern matching problems, and we propose an optimal $O(n)$ algorithm for constructing it.

The first part of this work appeared primarily in [135]. The second part of this work appeared primarily in [132]. Note that the implemented tools in this work were used in our work [154].

1.3 Organization of This Thesis

The rest of this thesis is organized as follows. In Chapter 2, we describe several basic notations, definitions and problems that relates to this thesis overall. In Chapter 3, we discuss efficient enumeration algorithms for alternative solutions of multiple sequence alignment problems. In Chapter 4, we propose an accurate clustering algorithm for cDNA library based on a variant of alignment algorithm called the spliced alignment. In Chapter 5, we demonstrate the power of pattern matching algorithms for a gene identification problem which is one of the most important problem in molecular biology today. In Chapter 6, we show three pattern matching data structures and algorithms for them based on suffix trees, that can be used for RNA structure analyses. In Chapter 7, we summarize the results of this thesis.

Chapter 2

Preliminaries

In this chapter, we first describe the sequence alignment problem in section 2.1. We survey the exact algorithms for it. We then describe the suffix tree data structure in section 2.2. We also describe two famous algorithms for constructing the data structure. Both techniques are very fundamental in the research areas of string pattern matching and important in the rest of this thesis.

2.1 Sequence Alignment Problem

In this section, we introduce the alignment problem and algorithms for computing the optimal solution.

2.1.1 Problem Definition

In this section, we describe what the multiple sequence alignment problem is. Before explaining the problem definition, let us describe some of the notations we use. Let Σ be a fixed set of alphabets that represent residues. The size of Σ is 20 in the case of protein sequences, and it is 4 in the case of DNA sequences. Let $\Sigma' = \Sigma \cup$'-', where - denotes a gap, and consider a score function between its members $s : \Sigma' \times \Sigma' \to R$. The score function is usually given by a score table. Table 2.1 shows a famous and widely-used score table called PAM-250 [4, 44].

Each member in Σ' except for a gap is called a character, and a finite string of characters is called a sequence. On the other hand, a finite string of members in Σ' is called a padded sequence. Then the set of the ith elements of two or more padded sequences is called the ith column of the set of the sequences. Furthermore, a set of consecutive columns is called a region. A set of padded sequences $(S'_1, S'_2, \ldots, S'_d)$ is called a d-alignment or simply an alignment of $(S_1, S_2, \ldots, S_d)$ if and only if all of the padded sequences have same length l, the ith column contains at least one character for any $i (1 \leq i \leq l)$, and $(S'_1, S'_2, \ldots, S'_d)$ becomes $(S_1, S_2, \ldots, S_d)$ if all gaps are deleted. Furthermore, the (pairwise) projection A_{ij} of an alignment $A = (S'_1, S'_2, \ldots, S'_d)$ is defined as the alignment obtained from (S'_i, S'_j) by removing columns without any characters.

The score of a 2-alignment is defined as the summation of scores of all columns obtained directly with the score function. The score of a d-alignment is defined as the summation of the scores of all the pairwise projections of the d-alignment. Finally, we can define the problem: the alignment problem is a problem of

Table 2.1: PAM-250 score matrix.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	12																			
S	0	2																		
T	−2	1	3																	
P	−3	1	0	6																
A	−2	1	1	1	2															
G	−3	1	0	−1	1	5														
N	−4	1	0	−1	0	0	2													
D	−5	0	0	−1	0	1	2	4												
E	−5	0	0	−1	0	0	1	3	4											
Q	−5	−1	−1	0	0	−1	1	2	2	4										
H	−3	−1	−1	0	−1	−2	2	1	1	3	6									
R	−4	0	−1	0	−2	−3	0	−1	−1	1	2	6								
K	−5	0	0	−1	−1	−2	1	0	0	1	0	3	5							
M	−5	−2	−1	−2	−1	−3	−2	−3	−2	−1	−2	0	0	6						
I	−2	−1	0	−2	−1	−3	−2	−2	−2	−2	−2	−2	−2	2	5					
L	−6	−3	−2	−3	−2	−4	−3	−4	−3	−2	−2	−3	−3	4	2	6				
V	−2	−1	0	−1	0	−1	−2	−2	−2	−2	−2	−2	−2	2	4	2	4			
F	−4	−3	−3	−5	−4	−5	−4	−6	−5	−5	−2	−4	−5	0	1	2	−1	9		
Y	0	−3	−3	−5	−3	−5	−2	−4	−4	−4	0	−4	−4	−2	−1	−1	−2	7	10	
W	−8	−2	−5	−6	−6	−7	−4	−7	−7	−5	−3	2	−3	−4	−5	−2	−6	0	0	17

finding the alignment of given sequences with the largest score.

In this definition of the problem, the gap penalty is not influenced by whether two gaps are consecutive or not. This kind of gap penalty is called the linear gap penalty. If there is starting gap penalty which is costed only to the beginning of consecutive gaps, it is called the affine gap penalty. In this thesis, we mainly deal with the linear gap penalty.

The weighted sum-of-pairs multiple alignment problem [5, 69] is a generalization of the simple sum-of-pairs multiple alignment problem described above. This version of the problem is often used when the phylogenetic tree is given. In this problem, we optimize the sum of weighted scores of each pairwise sequences alignment: we multiply the score of the alignment of the ith and the jth sequence by w_{ij}. We call (w_{ij}) a weight matrix. Note that w_{ii} is always 0 for any i.

The multiple alignment problem can be easily transformed to the shortest path problem on a grid-like directed acyclic graph with no negative edges as follows. Let S_k be the kth sequence of d sequences to be aligned, and $n_k = O(n)$ be the length of S_k. Then suppose a directed acyclic graph $G = (V, E)$ such that $V = \{(x_1, \ldots, x_d) | x_i = 0, 1, \ldots, n_i\}$ and $E = \{(v, v + e) | v \in V, e \in [0, 1]^d, e \neq \mathbf{0}\}$. In this graph, a path from $s = (0, \ldots, 0)$ to $t = (n_1, \ldots, n_d)$ corresponds to an alignment of the sequences.

In the alignment problem of two sequences, the length of an edge is defined from the score table between

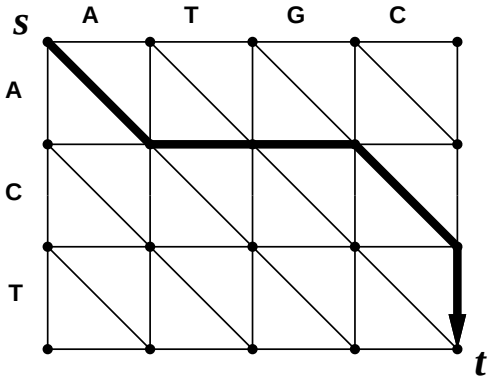


Figure 2.1: The graph for the alignment of two sequences ATGC and ACT. The s-t path in the bald line represents the alignment of ATGC- and A--CT.

characters, and the length of a path from s to t equals the score of the corresponding alignment. Figure 2.1 shows an example of it. In the graph, each diagonal edge corresponds to the match of characters, and each horizontal or vertical edges corresponds to the inserted gaps. In the multiple alignment problem, the sum of all the scores for alignments of pairwise sequences is generally used as the score. This score of the alignment equals the length of the corresponding path, if we define the length of each edge as the sum of the lengths of the corresponding edges in the pairwise projections of the alignment. In this way, the longest path problem from s to t in this graph is equivalent to the original alignment problem. This longest path problem can be easily transformed to the shortest path problem by reversing the signs of the lengths [72, 85, 87]. Thus we can use shortest path algorithms for graphs to compute the optimal alignment of given sequences.

2.1.2 Exact Algorithms for Sequence Alignment

Dynamic Programming

The alignment graph presented in the section 2.1.1 is a layered graph and we can easily use the dynamic programming (DP) technique to obtain the shortest path [44, 67, 73, 129, 150, 160] in time linear to the size of the graph size.

Algorithm 1 (Dynamic Programming) *Let $l(u, v)$ be the length of edge (u, v) in the alignment graph, s be the source $(0, 0, \ldots, 0)$ and t be the destination $(n_1, n_2, \ldots, n_d)$.*

1. *Let $p(s)$ be 0.*
2. *For $i = 1$ to $i = \sum_{1 \le j \le d} n_j$ do the following:*
   For all $v = (x_0, x_1, \ldots, x_d)$ such that $\sum_{1 \le k \le d} x_k = i$, compute the following value $p(v)$, and let previous(v) be $v - e$ which satisfies this equation (2.1). Note that this $p(v)$ equals the shortest path

length from s to v.

$$p(v) = \min_{e \in [0,1]^d, e \neq \mathbf{0}} (p(v-e) + l(v-e, v)) \tag{2.1}$$

3. *We can obtain the shortest path by tracing back previous(v) from the destination t.*

But the DP requires $O(n^d)$ memory space where n is the length of the longest sequence and d is the number of sequences, because it needs to store all vertices of the graph in memory. Thus the DP requires too much memory in the case of large d, and the DP can only deal with alignment of 2 or 3 sequences in ordinary case of aligning actual protein sequences. Note that there is an algorithm based on a divide and conquer technique that reduces the space to $O(n^{d-1})$, but it requires about twice computation time [129].

Dijkstra Algorithm

The Dijkstra method [48] is the most famous algorithm for the shortest path problem on graphs without negative edges. The outline of this algorithm is as follows:

Algorithm 2 (Dijkstra Method) *Let the graph in assumption be $G = (V, E)$, $l(v, w)$ be the length of edge (v, w) which is always non-negative, s be the source and t be the destination.*

1. *Let $p(v)$ be the potential of a vertex v, which is initialized with $+\infty$ except for the source s whose potential is zero, and S be an empty set.*

2. *Add to S the vertex v_0 which has the minimum potential in $V - S$. Then, stop if v_0 is t.*

3. *For all edges (v_0, v) in E, if $p(v_0) + l(v_0, v)$ is smaller than $p(v)$, replace $p(v)$ with $p(v_0) + l(v_0, v)$ and replace the path kept in v with the shortest path from s to v_0 added with the edge (v_0, v).*

4. *Go to step 2.*

This algorithm requires $O(m + n \log n)$ time where m is the number of the edges and n is the number of the vertices in the graph, which is worse than DP, and this algorithm itself is not so useful for the alignment problem.

A* Algorithm

The Dijkstra method will be remarkably more efficient if it is extended to the A* algorithm [21, 45, 64, 78, 112, 113, 134, 146] as follows. The A* algorithm will not search the whole graph in finding the shortest path if a good estimate for the shortest path length from each vertex to the destination t can be used.

The basic algorithm for the A* algorithm is like following:

Algorithm 3 (A* Algorithm) *Let the graph in assumption be $G = (V, E)$, $l(v, w)$ be the length of the edge (v, w) which is always non-negative, s and t be the source and the destination, and $h(v)$ be heuristic estimate for the length of the shortest path to t which is not longer than the actual one.*

1. *Let $p(v)$ be the potential of a vertex v, which is initialized with $+\infty$ except for the source s whose potential is zero, and S be an empty set.*

2. *Add to S the vertex v_0, whose value of $p(v) + h(v)$ is smallest in $V - S$. Then, stop if v_0 is t.*

3. *For all edges (v_0, v) in E, if $p(v_0) + l(v_0, v)$ is smaller than $p(v)$, replace $p(v)$ with $p(v_0) + l(v_0, v)$ and replace the path kept in v with the shortest path from s to v_0 added with the edge (v_0, v). If v is in S, remove it from S.*

4. *Go to step 2.*

In this algorithm, $p(v)$ for vertex v in S is also the length of the shortest path from s to v. $p(v) + h(v)$ is the estimate for the shortest path from s to t via v. The searched vertices by the A* algorithm is always within searched vertices by the Dijkstra method. In this way, the A* algorithm can get the shortest path more effectively. If the estimate $h(v)$ equals the actual shortest path length from v to t, this algorithm searches only on the shortest path.

The estimate $h(v)$ must not be longer than the shortest path length from v to t, because the final obtained path must not be longer than the other paths. In case some of the estimate $h(v)$ is longer than the actual shortest path, this algorithm is called the A algorithm, and is often used as a heuristic algorithm for the shortest path problem.

In the A* algorithm, the shortest path from s may not appear first, and a shorter path may be found in the future search, which is the reason of the removal of vertices from S in step 3. It makes this algorithm rather inefficient. This can be avoided if the estimator is dual feasible. The definition of 'dual feasible' is as follows:

Definition 1 *The estimator h for the shortest path to t is called dual feasible if and only if h satisfies the following constraint, which is called monotone restriction:*

$$\forall (u, v) \in E \quad l(u, v) + h(v) \geq h(u) \tag{2.2}$$

If the estimator is dual feasible, the A* algorithm can be easily translated to the Dijkstra method by modifying the length of the edges [81, 85, 87, 131]:

Theorem 1 (Ikeda Hsu et al.) *Let h be a dual feasible estimator for s. The Dijkstra method on a graph in which the length of edge (u, v) or $l(u, v)$ is replaced by $l'(u, v)$ as follows is equivalent to the A* algorithm on the original graph.*

$$l'(u, v) = l(u, v) + h(v) - h(u) \tag{2.3}$$

Proof: $l'(u, v)$ is non-negative because of dual feasibility of h. Thus, the shortest path can be searched with the Dijkstra method in the modified graph. Let p be the shortest path from s to v. Then the potential of v used in searching with the Dijkstra method on the modified graph is described as follows:

$$\begin{aligned} p(v) &= \sum_{(u,v) \in p} l'(u, v) \\ &= \sum_{(u,v) \in p} l(u, v) + h(v) - h(s) \end{aligned}$$

This means the Dijkstra method on the new graph is equivalent to the A* algorithm on the original graph, because $h(s)$ is constant. □

Ikeda and Imai [87] show the following estimator is very useful for the alignment problem in case $d > 2$. Let G_{ij} be the corresponding graph to the alignment of S_i and S_j, v_{ij} be the corresponding vertex in G_{ij} to v in G, and $L^*(u, v)$ be the shortest path length from u to v. Then $h(v) = \sum_{1 \leq i < j \leq d} L^*(u_{ij}, v_{ij})$ can be used as a powerful estimator for the multiple alignment problem. This estimator is easily be shown to be dual feasible, *i.e.* $l(u, v) + h(v) \geq h(u)$. Hence the A* algorithm can be applied as following.

Algorithm 4 (A* algorithm for the Alignment Problem)

1. *For each of i and j $(1 \leq i < j \leq d)$, apply DP to the graph G_{ij} from t_{ij} to calculate $L^*(v_{ij}, t_{ij})$ for each v_{ij} in V_{ij}. Then let $h(v)$ be $\sum_{1 \leq i < j \leq d} L^*(u_{ij}, v_{ij})$.*

2. *Modify the length of edge (u, v) in G using $h(v)$ as follows, and compute the shortest path with the Dijkstra method.*

$$l'(u, v) = l(u, v) + h(v) - h(u) \tag{2.4}$$

Note that the time and space used for the DP in the step 1 is negligible, if d is large. This A* algorithm can deal with the alignment problem of 5 to 6 normal sequences in reasonable time.

A vertex in the graph for the multiple alignment has $2^d - 1$ edges going out from it, and the A* algorithm examines all the descendant vertices and keeps in a heap the information about all of them. If an upper bound $L^+(s, t)$ for the s-t shortest path, which corresponds to the lower bound of the score of the alignment, is given, the necessary space for the heap can be reduced [87]: we can ignore w such that $L^*(s, v) + l(v, w) > L^+(s, t)$, when we examine the descendant vertices of v. If the necessary space for the heap is reduced, the computing time of the A* algorithm will also be reduced. This is called the enhanced A* algorithm.

Note that the branch-and-bound techniques implemented in `MSA` program [72] is equivalent to this enhanced A* algorithm, and Araki et al. [10] showed that the estimated score computed directly by the score matrix is useful for the 2-alignment problem.

2.1.3 Spliced Sequence Alignment

There are many variations of alignments [73], and we here introduce one of them, the spliced sequence alignment, or the spliced alignment for short, which is designed for aligning a pair of sequences called *a splicing pair*. As we described in chapter 1, an mRNA sequence will be spliced after the transcription from the DNA sequence. A splicing pair is a pair of sequences one of which is derived from the other by the splicing mechanism. Figure 2.2 shows a picture of the mechanism. The standard alignment tools for the above problem are not suitable for aligning a splicing pair, because they do not take the splicing mechanism into account.

There are long gaps in the spliced sequence when we align a splicing pair. Regions of the template sequence that correspond to long gaps are called *splice sites*. The aligned regions of both sequences have very high similarity, say 95% to 99.9%, and therefore no long gaps will be inserted into the template sequence. Note that the errors (normally less than 5%) between the aligned regions are due to sequencing errors. The error rates due to transcription are known to be very low. Thus the minimum similarities between the aligned regions can be estimated while doing sequencing experiments for cDNA library construction. The 5'

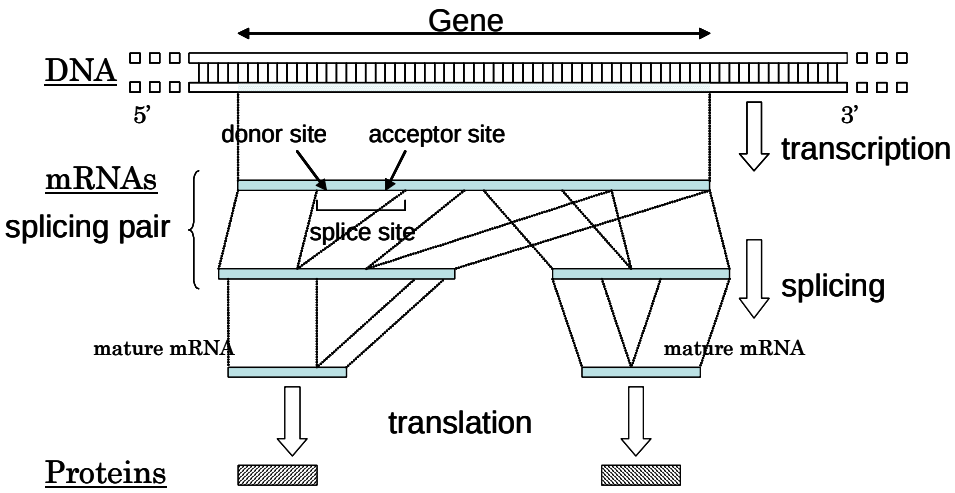


Figure 2.2: The Splicing Mechanism.

end and the 3' end of a splice site are called the donor site and the acceptor site, respectively. It is known that most of the splice sites start with `GT` and end with `AG`. We must consider all of these facts when we align two sequences to determine whether they form a splicing pair or not.

There are several algorithms [59, 63, 82, 84, 105, 107, 151, 156] for the spliced alignment problem. Among them, Mott's algorithm [107] is a very reasonable modification of the ordinary dynamic programming alignment algorithm to align a splicing pair. If there is a set of consecutive gaps in the alignment that could be considered as a splice site candidate, Mott's algorithm gives an appropriate splice site penalty (decided by the signals found in the splice site candidate) to the set of consecutive gaps regardless of its length of the region, instead of the ordinary gap score given by the score function given above. His algorithm runs in $O(nm)$ time where n and m are the lengths of the two sequences, which is the same as the Smith-Waterman algorithm. See [107] for more detail. A modified version of his algorithm is used in chapter 4.

2.2 Suffix Trees

In this section, we introduce the suffix tree data structure and describe the algorithms for constructing it. We also introduce several variants of suffix trees.

2.2.1 Suffix Tree Data Structure

The suffix tree [50, 73, 102, 155, 163] of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S^+ = S\$$ where \$ is a character such that $\$ \notin \Sigma$. Figure 2.3 shows an example of this data structure. It shows the suffix tree data structure of a string 'mississippi'. This data structure is very useful for various problems in sequence pattern matching. Using it, we can query a substring of length m in $O(m \log |\Sigma|)$ time, we can find frequently appearing substrings in a given sequence in linear time, we can find a common substring of many sequences, also in linear time, and so on [73].

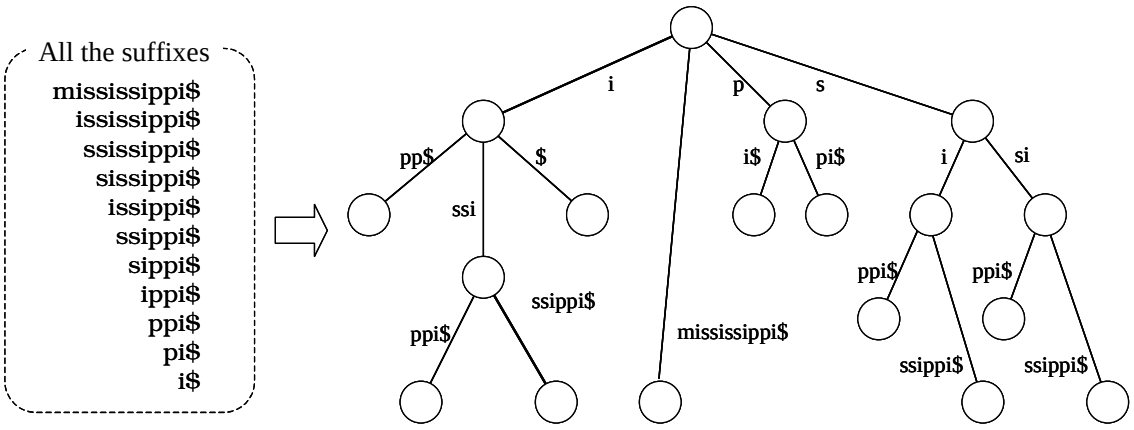


Figure 2.3: The suffix tree of a string 'mississippi.'

The tree has $n+1$ leaves, and each internal node has more than one child. Each edge is labeled with a non-empty substring of S^+, and no two edges out of a node can have labels that start with the same character. Each node is labeled with the concatenated string of edge labels on the path from the root to the node, and each leaf has a label that is a different suffix of S^+. Because each edge label is represented by the first and the last indices of the corresponding substring in S^+, the data structure can be stored in $O(n)$ space.

This data structure was first proposed by Weiner [163], who gave an $O(n|\Sigma|)$ algorithm for constructing it, where n is the string length and $|\Sigma|$ is the size of the alphabet. McCreight [102] improved it by giving an $O(n \log |\Sigma|)$ algorithm. After that, Ukkonen [155] proposed an on-line $O(n \log |\Sigma|)$ algorithm, which processes a string character by character from left to right. Then Farach [50] proposed an $O(n)$ algorithm for an integer alphabet $\{1, \ldots, n\}$.

In this thesis, we use the following definitions. In a suffix tree, let $parent(u)$ be the parent node of node u, let σ_u be the string label of node u, and let $node(\alpha)$ be node u in the tree such that $\sigma_u = \alpha$ if it exists. The suffix link of u is a link to a node with label α if u has a label of $c\alpha$, where c is any single character. It is known that a suffix link always exists for any u except for the root in a suffix tree [73, 102, 155]. If u is the root we let its suffix link be u itself. Let $sl(u)$ be the suffix link of u.

Ukkonen's Algorithm

In this subsection, we introduce the Ukkonen's algorithm that enables us to construct the suffix tree on-line in $O(n \log |\Sigma|)$ time:

Theorem 2 (Ukkonen [155]) *The suffix tree of a sequence ($\in \Sigma^n$) can be constructed on-line in $O(n \log |\Sigma|)$ time.*

From now on, we describe the algorithm briefly. The implicit suffix tree of S is the compacted trie of all the suffixes of S, and a label for an edge that ends at a leaf is represented by only the first index of the

label. Let T_i $(1 \leq i \leq n+1)$ denote the implicit suffix tree of $S[1..i]$, where $n = |S|$. Ukkonen's algorithm consists of $n+1$ phases, and we construct the implicit suffix tree T_i from T_{i-1} in the ith phase.

In the ith phase, we construct a new node $u = node(S^+[j..i])$ for all $1 \leq j \leq i$ in this order if there is no locus for $S^+[j..i]$ in the tree. When we must construct such new node u, if there is no node with a label of $S^+[j..i-1]$, we must also construct a new internal node at the locus of $S^+[j..i-1]$, and let it be the parent of u. We call this procedure for single j the jth extension of the ith phase.

Notice that we do not have to construct node $u = node(S^+[j..i])$ if $v = node(S^+[j..i-1])$ was a leaf in the previous phase, because of the definition of the implicit suffix tree: σ_v is $S^+[j..i]$ in this phase. Thus, if there is a leaf for each of $node(S^+[j..i-1])$ for all $j < k$ in phase $i-1$, we can begin by constructing $node(S^+[j+1..i])$ in this phase. Furthermore, if there is a locus for $S^+[j..i]$ for some j, it is easy to see that there already exist loci for $S^+[k..i]$ $(k > j)$ too, and that there is no need to construct nodes for them in this phase.

Ukkonen's algorithm, like McCreight's algorithm, maintains at each node u of the suffix tree a suffix link $sl(u)$. In any phase, we construct nodes $u_j = node(S^+[j..i])$ for several consecutive j's and u'_j =node($S^+[j..i-1]$) if necessary, in the manner described above. Notice that $u_{j+1} = sl(u_j)$ and $u'_{j+1} = sl(u'_j)$ if they exists. For the last u_j to be constructed in this phase, we will check the locus for $S^+[j+1..i]$, which is $sl(u_j)$ in the next extension according to the algorithm. Thus we will know within the phase the suffix links of all the constructed nodes in the same phase. In this way, we can maintain the suffix links.

Using the suffix links, we can construct node $u_j = node(S^+[j..i])$ faster: It is easy to see that $sl(parent(u_{j-1}))$ must be an ancestor of u_j, and we can find the locus of $S^+[j..i-1]$ by tracing edges from $sl(parent(u_{j-1}))$. We call tracing from the suffix link to the target locus "scanning."

In this way, the algorithm achieves an $O(n \log |\Sigma|)$ time complexity. For more details of the algorithm and the analysis of the computing time bound, see [73] or [155].

Farach's Algorithm

Alphabet $\{1, \ldots, n\}$ is called an integer alphabet. Ukkonen's algorithm requires $O(n \log n)$ time for computing the suffix trees with the integer alphabet. There is another algorithm for constructing the suffix trees with integer alphabets by Farach [50] that is theoretically faster than Ukkonen's algorithm in the case of integer alphabet:

Theorem 3 (Farach [50]) *The suffix tree of a string $S \in \{1, \ldots, n\}^n$ can be constructed in $O(n)$ time.*

For the details of this algorithm, see [50]. What we must note is that Farach's suffix tree construction algorithm and our algorithms to be presented in chapter 6 use the following theorem:

Theorem 4 (Harel and Tarjan [76]) *For any tree with n nodes, we can find the lowest common ancestor (LCA) of any two nodes in a constant time after $O(n)$ preprocessing if the following values can be obtained in a constant time: bitwise AND, OR, and XOR of two binary numbers, and the positions of the leftmost and rightmost 1-bit in a binary number.*

This algorithm is described in detail in [73, 76]. Note that Bender and Farach [25] proposed recently a simpler algorithm with the same time bound for the LCA query problem. This theorem indicates that the

longest common prefix (LCP) of any two suffixes can be obtained from the suffix tree in a constant time after linear-time preprocessing.

2.2.2 p-Suffix Trees

A parameterized string [16, 18, 19], or a p-string for short, is a string over two alphabets Σ and Π, where Σ is an ordinary alphabet and Π is a set of parameters. Two p-strings are said to match if they are same except for a one-to-one correspondence between the characters in Π occurring in them. For example, two p-strings $\mathtt{AC}x\mathtt{BC}y z y\mathtt{A}z x\mathtt{C}$ and $\mathtt{AC}y\mathtt{BC}z x z\mathtt{A}x y\mathtt{C}$ match ($\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $\Pi = \{x, y, z\}$).

As in [18], we define prev(S) for any p-string S as follows:

Definition 2 *Let N be the set of nonnegative integers. Consider a string $S[1..n] \in (\Sigma \cup \Pi)^*$. If $S[i] \in \Pi$, let c_i be the index of the nearest same parameter in Π to the left, i.e., $c_i < i$, $S[c_i] = S[i]$ and $S[k] \neq S[i]$ for any k such that $c_i < k < i$. If such c_i does not exist, let $c_i = i$. Now, replace $S[i]$ with $i - c_i \in N$ if $S[i] \in \Pi$, for all i: We let the obtained string in $(\Sigma \cup N)^*$ be prev(S).*

For example, prev($\mathtt{AC}x\mathtt{BC}y z y\mathtt{A}z x\mathtt{C}$) = $\mathtt{AC0BC002A38C}$. Let $S[i]$ denote the ith character of S, and $S[i..j]$ denote a substring of S that starts at position i and ends at position j. Note that $S = S[1..|S|]$. The p-suffix tree of a p-string S is the compacted trie of all the prev-encoded suffixes, *i.e.*, prev($S^+[i..n+1]$) for all positions i, where $S^+ = S\$$ and \$ is a character in neither Σ nor Π. We here consider \$ as an ordinary alphabet, not as a parameter. Baker [16, 18, 19] proposed this data structure and showed that it can be constructed in $O(n(|\Pi| + \log|\Sigma|))$ time. Kosaraju [95] improved the time by giving an $O(n(\log|\Pi| + \log|\Sigma|))$ algorithm. Note that both of the algorithms are based on McCreight's suffix tree construction algorithm [102] and that neither supports on-line computation. In chapter 6, we will give an on-line algorithm for the same task based on Ukkonen's algorithm [155].

2.2.3 The Suffix Tree of a Tree

A set of strings $\{S_1, \ldots, S_k\}$, such that no string is a suffix of another, can be represented by a common suffix tree [32, 94] (CS-tree for short), which is defined as follows:

Definition 3 (CS-tree) *In the CS-tree of a set of strings $\{S_1, \ldots, S_k\}$, each edge is labeled with a single character, and each node is labeled with the concatenated string of edge labels on the path from the node to the root. In the tree, no two edges out of a node can have the same label. Furthermore, the tree has k leaves, each of which has a different label that is one of the strings, S_i.*

Figure 2.4 shows an example of a CS-tree. The number of nodes in the CS-tree is equal to the number of different suffixes of strings. Thus, the size of a CS-tree is not larger than the sum of the lengths of the strings represented by the CS-tree. Note that the CS-tree can be constructed easily from strings in a time linear to the sum of the lengths of the strings.

The generalized suffix tree of a set of strings $\{S_1, \ldots, S_k\}$ is the compacted trie of all the suffixes of all the strings in the set. As mentioned in [94], the suffix tree of a CS-tree is the same as the generalized suffix tree of the strings represented by the CS-tree. Furthermore, the size of the generalized suffix tree is linear

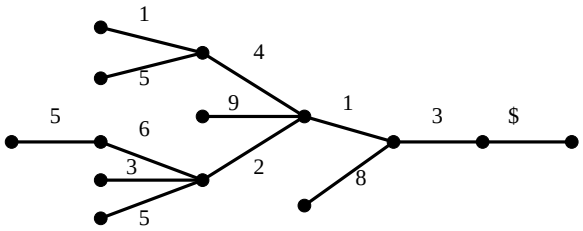


Figure 2.4: CS-tree of the strings 1413\$, 5413\$, 913\$, 56213\$, 3213\$, 5213\$, and 83\$.

to that of the CS-tree, because the number of leaves of the suffix tree is equal to the number of edges in the CS-tree. Note that an edge label of the suffix tree of a CS-tree corresponds to a path in the CS-tree, and it can be represented by the pointers to the first edge (nearest to the leaves) and the path length.

Let n_i be the length of S_i, and let $N = \sum_i n_i$. Let n be the number of nodes in the CS-tree of the strings. The generalized suffix tree can be obtained in $O(N)$ time in the case of integer alphabets (*i.e.*, $S_i \in \{1, \ldots, n\}^{n_i}$) as follows. First, we construct the suffix tree of a concatenated string of $S_1\$S_2\$\cdots\$S_k$ using Farach's suffix tree construction algorithm. Then, we obtain the generalized suffix tree by cutting away the unwanted edges and nodes. But N is sometimes much larger than the size n of the CS-tree. There exists a tree for which N is $\Theta(n^2)$ for example. This means that the $O(N)$-time suffix tree construction algorithm given above is not at all a linear time algorithm. The best-known $O(n \log |\Sigma|)$ algorithm [32] for this problem is based on Weiner's suffix tree construction algorithm [163]. In chapter 6, we will improve it in the case of integer alphabets by giving a new algorithm based on Farach's linear-time suffix tree construction algorithm.

Chapter 3

Efficient Enumeration of Alternative Multiple Sequence Alignments

Aligning two or more biological sequences to compare them is the most fundamental routine work for almost all the molecular biologists who deal with sequence data. In many algorithms, an alignment is often obtained by the alignment with the best score on some given scoring criterion between characters. But it is also known that the obtained alignment is not always the biologically best alignment. Thus a more flexible way of aligning sequences is desired. In this chapter, we propose two efficient approaches that enable us to align sequences in a very flexible way. One approach is by enumerating suboptimal solutions and the other is by doing parametric analysis.

A suboptimal alignment is an alignment whose score is close to the optimal one. In case of aligning two sequences, the suboptimal alignment enumeration problem is well-studied [37, 108, 130, 166] and used for many applications such as predicting protein structure and so on [127, 128, 161]. In the multiple alignment problem, we can see suboptimal alignments of each pair of sequences with these methods for only two sequences as in [166], but these are not the accurate suboptimal alignments of all the sequences.

Enumeration of the suboptimal alignments had not been considered as very practical even in the case of aligning two sequences [108, 166]. But such enumeration has become easier because a new efficient algorithm for the k shortest paths problem was proposed by Eppstein [49]. This algorithm enumerates the lengths of the k shortest paths in $O(k+n+m)$ time and space if we are given the shortest path tree from the source or to the destination for any graph with non-negative m edges and n vertices. Even if we have to output the paths themselves, this algorithm requires only time linear to the output size, add to the time given above. Note that the shortest path tree can be constructed with DP or the A* algorithm.

For this approach, we first discuss the method to obtain E_Δ, which can be done with some extension of the A* algorithm. E_Δ represents all aligned groups of residues in optimal and suboptimal alignments which are at most Δ worse than the optimal. Furthermore, based on this extended A* algorithm and the Eppstein algorithm, we go on to discuss the methods for the enumeration problem.

The number of suboptimal solutions is very large, and we should restrict the outputs to important solutions if we know what kind of solution is important. Hence we discuss what kind of suboptimal alignment is unnecessary to enumerate, and propose an efficient technique to enumerate only necessary alignments. This

technique is so splendid that it remains output size sensitive even though we restricted solutions only to the necessary alignments in our concept.

For the other approach of parametric analysis, we first review the basic techniques for parametric analysis [74, 83, 157, 159, 162, 165], and propose new techniques for multiple alignments. As for the techniques, we use the Eppstein algorithm to examine all the optimal solutions for one fixed parameter, and upper bounding technique for the parametric alignment. In the most of previous work, they computed only one optimal solution for one fixed parameter in parametric analysis, while we examine all optimal solutions using the Eppstein algorithm.

We do a parametric study on gap penalties, score tables and weight matrices. Related with the weight matrices, we show the (enhanced) A* algorithm is applicable for the weighted multiple alignment problem. The weighted problem is considered only when the phylogenetic tree is given, but our approach enables more flexible study of the weighted multiple alignment problem. This technique may also be useful in constructing or tuning phylogenetic trees.

In this chapter, section 3.1 describes the first approach by enumerating suboptimal solutions, while section 3.2 describes the second approach by parametric analysis.

3.1 Enumeration of Suboptimal Alignments of Multiple Sequences

In this section, we deal with the first approach to the flexible alignment, that is the enumeration of suboptimal solutions. At first, we introduce the Eppstein algorithm [49] which is very efficient for enumerating suboptimal paths in ordinary graphs. We then consider how to obtain E_Δ efficiently for the multiple alignment problem. Furthermore, we also discuss how to enumerate the suboptimal solutions, introducing the new notation of classes of suboptimal solutions.

3.1.1 Eppstein Algorithm

Eppstein [49] proposed an algorithm which finds implicitly the k shortest paths for the graph G with non-negative m edges and n vertices regardless of cycles, in $O(m + n + k)$ time after the shortest path tree is constructed. He also proposed an easier algorithm of $O(m + n \log n + k)$ time. Note that before the proposition of Eppstein, the best algorithm for the problem was $O(k(m + n \log n))$.

In the algorithm, we use $\delta(u, v)$ for the edge (u, v) as in equation (3.2). This $\delta(u, v)$ denotes how much longer the path will be using the edge (u, v) than the optimal path by way of v, and therefore this value is always non-negative.

If an edge (u, v) is on the shortest path tree, $\delta(u, v)$ is zero, otherwise, it is called a sidetrack and $\delta(u, v)$ may not be zero. If we go along an s-t path p other than the shortest path, there must be one or more sidetracks on p, and we define $sidetrack(p)$ as the nearest sidetrack from s within them.

Let $(tail(p), head(p))$ be $sidetrack(p)$. Then we can suppose a heap, in which the parent of a path p is a path which is same as p from $head(p)$ to t, but goes along the shortest path from s to $head(p)$ instead of using $sidetrack(p)$. We define $parent(p)$ as the parent of p and we call p a child of $parent(p)$. The root of the heap is the shortest path, and all the paths from s to t appear in the heap once. In this heap, p

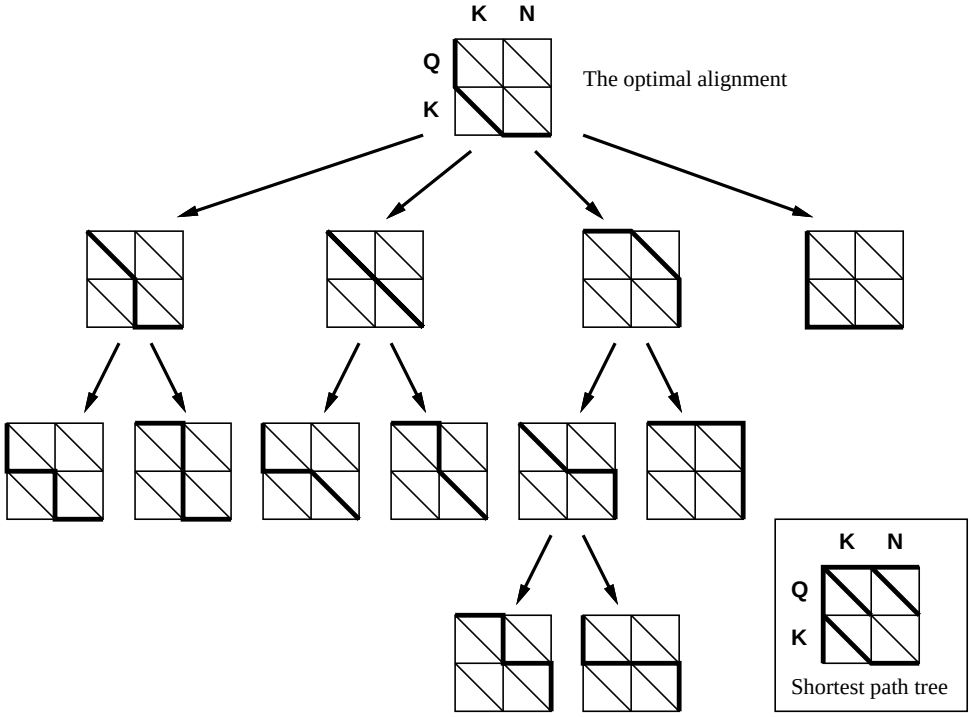


Figure 3.1: The path heap of the alignment graph of two sequences KN and QK.

is $\delta(sidetrack(p))$ longer than $parent(p)$. Figure 3.1 shows the path heap of the alignment graph of two sequences KN and QK.

We call a heap an i-heap if the node of the heap has only i children at most (it is not required to be balanced). The basic concept of the Eppstein algorithm is to modify this path heap to a 4-heap, sharing as many nodes as possible. Figure 3.2 shows an example of this compact version of the path heap of the same alignment graph in Figure 3.1, in which some of the nodes are shared and the number of nodes in it is reduced.

From this heap, we can obtain the k shortest paths in $O(k)$ time [60] or $O(k \log k)$ time in sorted form. The following is the outline of this algorithm:

Algorithm 5 (Eppstein)

1. *Construct the shortest path tree from s to all the other vertices.*

2. *For each vertex v, construct $H_G(v)$, a 3-heap of sidetracks (u', u) such that u is on the shortest path from s to v, ordered by $\delta(u', u)$. Let the length from the root of $H_G(v)$ to a node x be $\delta(u, v)$ if x represents sidetrack (u, v).*

   (a) *For each vertex v, construct $H_{out}(v)$, a 2-heap of sidetracks (v', v) ordered by $\delta(v', v)$ in which the root has only one child.*

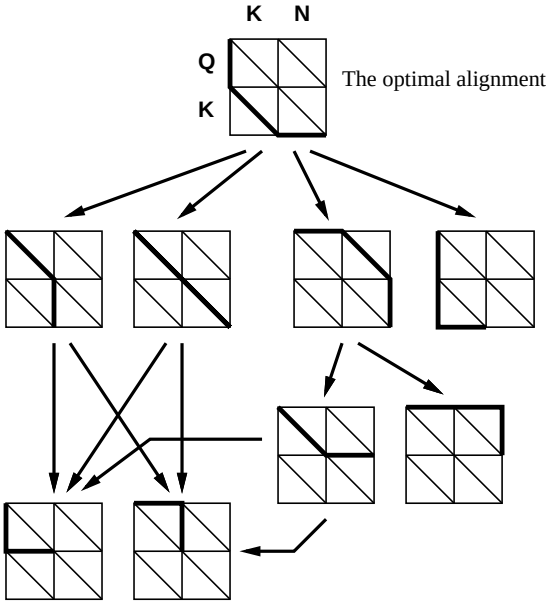


Figure 3.2: The compact path heap of the alignment graph of two sequences `KN` and `QK`. Some of the nodes are shared, and the number of nodes in it is reduced.

(b) For each vertex v, construct $H_T(v)$, a 2-heap of vertices on the shortest path from s to v ordered by the value δ of the root of the heap made in step 2-(a).

(c) Merge $H_{out}(v)$ and $H_T(v)$ to make $H_G(v)$. Then let the length of the edge from node x_1 to node x_2 in this heap be $\delta(x_2) - \delta(x_1)$.

3. For each v in G, make an edge from each node in $H_G(v)$ which represents a sidetrack (u', u) to the root of $H_G(u')$, and let the length of this new edge be the value of the root.

4. Make a new node for each v in G, and make an edge from this node to the root of $H_G(v)$. Let the length of this edge be δ of the root. Let this new graph be $P(G)$.

Then we can find a heap $H_v(G)$ in $P(G)$ for any v, considering the root as the node made in step 4 for v, and the value of a node as the length from the root to the node. There is a one-to-one correspondence between the nodes in $H_v(G)$ and the paths from v to t in G, and the k smallest nodes in this virtual heap $H_v(G)$ correspond to the k shortest paths. Moreover, we can easily restore the path from the node of the heap, which can be done in $O(n')$ time where n' is the size of the output alignment.

Note that the shortest path tree in step 1 is constructed generally by the Dijkstra method, but for problems such as the alignment problem, we can also use DP. Eppstein showed the step 2 can be done in $O(n + m)$ time with a very complicated algorithm, but we use a far more easier and practically faster algorithm of $O(n \log n + m)$ time, which is also proposed by Eppstein [49]: we make $H_G(v)$ one by one from s to the other vertices along the shortest path tree, sharing as many nodes as possible.

3.1.2 Upper Bounding Technique for Computing E_Δ

E_Δ is a set of vertices which are used by the s-t paths whose lengths are at most Δ longer than the shortest path, and it corresponds to all aligned groups of residues in optimal and suboptimal alignments in original problem. The problem to compute it is well-studied [85, 108, 130, 166]. Here we show how to compute this set E_Δ with the A* algorithm [85]. For any path p from s to t, the modified path length by the expression (2.4) is only $h(t) - h(s)$ longer than the original length. This value is not relevant to p, thus E_Δ on the modified graph is same as the original one.

Theorem 5 *Any paths from s to t on the graph in which the length of edge (u, v) or $l(u, v)$ is replaced by $l'(u, v)$ as in (2.4) are a constant shorter than those on the original graph.*

Proof: Let p be a path from s to t, and h be a dual feasible estimator for the shortest path length to t. Then the length of a path p or $length'(p)$ in the new graph is described by the length of p or $length(p)$ in the original graph as follows:

$$\begin{aligned} length'(p) &= \sum_{(u,v)\in p} l'(u,v) \\ &= \sum_{(u,v)\in p} l(u,v) + h(t) - h(s) \\ &= length(p) + h(t) - h(s) \end{aligned} \tag{3.1}$$

According to this, all the paths on the new graph from s to t are $h(s) - h(t)$, which is constant, shorter than those on the original graph. □

Note that the following corollaries can be easily derived from Theorem 5.

Corollary 1 *E_Δ related to the paths from s to t on a graph in which the length of edge (u, v) or $l(u, v)$ is replaced by $l'(u, v)$ as in (2.4) are same as that on the original graph.*

Corollary 2 *The k shortest paths from s to t on a graph in which the length of edge (u, v) or $l(u, v)$ is replaced by $l'(u, v)$ as in (2.4) are same as those on the original graph.*

Hence, first we modify the edge lengths with some dual feasible estimator, and then we can obtain E_Δ with the Dijkstra method as follows [85] on this modified graph.

Algorithm 6 (E_Δ)

1. *Search from s by the Dijkstra method until the shortest path from s to t is discovered.*
2. *Search successively until a vertex v, to which the shortest path from s is more than Δ longer than the s-t shortest path, is discovered.*
3. *Modify the length of each edge (u, v) to $\delta(u, v)$ as follows:*

$$\delta(u,v) = l(u,v) + L^*(s,u) - L^*(s,v) \tag{3.2}$$

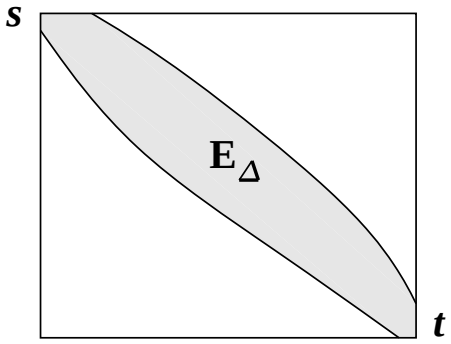


Figure 3.3: E_Δ is a set of vertices which are used by the s-t paths whose lengths are at most Δ longer than the shortest path.

4. *Apply the Dijkstra method from t until a vertex from which the shortest path to t is longer than Δ is discovered in the modified graph. E_Δ is the set of vertices searched in this step.*

A vertex in the graph for the multiple alignment has $2^d - 1$ edges going out from it, and the Dijkstra algorithm examines all the descendant vertices and keeps the information about all of them. If an upper bound $L^+(s,t)$ for the s-t shortest path is given, we can also decrease the heap size for searching as in the case of computing the optimal solution with the enhanced A* algorithm: we can ignore w such that $L^*(s,v) + l(v,w) > L^+(s,t) + \Delta$, when we examine the descendant vertices of v.

In general, such kind of an upper bound is difficult to obtain. However, we can use the actual shortest path length obtained in step 1 for the upper bound in step 2: we can ignore w such that $L^*(s,v) + l(v,w) > L^*(s,t) + \Delta$. Note that if we are given some upper bound of the solution before computing the optimal one, we can of course use it too.

3.1.3 Extending Eppstein Algorithm to Reduce Memory Space

In this subsection, we discuss how to enumerate efficiently all the suboptimal alignments whose scores are at most Δ lower than the optimal one. The original Eppstein algorithm requires searching all over the graph, and requires much memory. But it is evident that we only have to apply the Eppstein algorithm in the subset E_Δ after computing E_Δ as in the previous section, and then search the Eppstein's heap structure with the depth first method.

Moreover, if we use the easier $O(n \log n + m)$ algorithm in step 2 (b) of Eppstein algorithm (Algorithm 5), we do not have to compute E_Δ additionally. First, we must take the step 1 and 2 in the Algorithm 6, using the upper bounding technique. These procedures cannot be skipped. After these procedures, we implement the Eppstein algorithm as follows:

Algorithm 7 (Eppstein algorithm with A*)

1. *Construct the Eppstein's heap structure only on the shortest path.*

2. *Search for suboptimal solutions which are at most Δ worse than the optimal (root) from the root of $H_s(G)$ with the depth first search method. If we encounter $H_G(v)$ which has not been constructed yet, we construct the heap structures of vertices on the shortest path from s to v for which we have not constructed heaps yet.*

When we finish enumerating all the suboptimal alignments with this method, the set of vertices for which we constructed the Eppstein heap is also E_Δ.

Theorem 6 *The set of vertices S for which the Eppstein's heap structures are computed in the algorithm 7 is E_Δ.*

Proof: We construct $H_G(v)$ for all the vertices in E_Δ in the algorithm 7. Thus we can easily see that $E_\Delta \in S$. A newly encountered vertex v for which $H_G(v)$ has not constructed is in E_Δ, because we only search suboptimal paths which are Δ longer than the shortest path at most. Add to that, to obtain $H_G(v)$ for some vertex v, we only have to compute $H_G(u)$ for each vertex u on the shortest path from the source s to v in the easier method of the step 2(b) in algorithm 5, which is also trivially in E_Δ. Thus, we do not construct Eppstein's heap structures for vertices outside of E_Δ. Hence, we conclude that $S = E_\Delta$. □

Thus we do not have to compute E_Δ additionally. Notice that this technique can be also used in general graphs other than the graphs for alignments.

3.1.4 Classification of Suboptimal Alignments

In this subsection, we classify suboptimal alignments. We introduce a notion of alignment class D_i as follows:

Definition 4 *D_i is a class of alignments which have i regions different from the optimal alignment, which is in D_0.*

Figure 3.4 shows some examples of suboptimal multiple alignments of protein fragments. (a) is the optimal alignment, and (b), (c) and (d) are suboptimal alignments. The regions bounded by boxes are the regions that are different from the optimal alignment. (b) and (c) have only one such region. On the other hand, (d) has two, both of which appear also in (b) or (c). According to our notion of classification, the optimal alignment (a) is in the class D_0 (and none of the others are in this class), suboptimal alignments (b) and (c) are in the class D_1, and suboptimal alignment (d) is in the class D_2.

Considering the fact that we can easily reconstruct (d) from (b) and (c), (d) is not so important as (b) nor (c) and is sometimes unnecessary to enumerate. Thus, it will be a good news if we can efficiently enumerate only the alignments in the classes D_0 and D_1.

3.1.5 Avoiding Unnecessary Alignments

The paths to which the alignments in the class D_i correspond branch off i times from the shortest path to which the alignment in the class D_0 corresponds. Hence, we can consider a very easy branch-and-bound technique to avoid such alignments in D_i ($i \geq 2$) in enumeration of suboptimal alignments: when we search the Eppstein's heap structure, if $head(p)$ of s-t path p is on the s-t shortest path and $parent(p)$ is not the

```
REAFSQAIWRATFAQVPESRSLFKR==
ADFLV-ALF-EKFPDSANFFADFKGKS
KNG-S-LLFGLLFKTYPDTKKHFKHFD
LAAVF-TAYPDIQARFPQFAGK-DVAS
GSGVE-ILY-FFLNKFPGNFPMFKKLG
```

(a) The optimal alignment

```
REA FSQ AIWRATFAQVPESRSLFKR==
ADF LV- ALF-EKFPDSANFFADFKGKS
KNG -S- LLFGLLFKTYPDTKKHFKHFD
LAA -VF TAYPDIQARFPQFAGK-DVAS
GSG -VE ILY-FFLNKFPGNFPMFKKLG
```

(b) A suboptimal alignment

```
REAFSQAIWRATFAQVPESRS LF KR==
ADFLV-ALF-EKFPDSANFFA DF KGKS
KNG-S-LLFGLLFKTYPDTKK HF KHFD
LAAVF-TAYPDIQARFPQFAG -K DVAS
GSGVE-ILY-FFLNKFPGNFP MF KKLG
```

(c) Another suboptimal alignment

```
REA FSQ AIWRATFAQVPESRS LF KR==
ADF LV- ALF-EKFPDSANFFA DF KGKS
KNG -S- LLFGLLFKTYPDTKK HF KHFD
LAA -VF TAYPDIQARFPQFAG -K DVAS
GSG -VE ILY-FFLNKFPGNFP MF KKLG
```

(d) Unnecessary alignment to check

Figure 3.4: Examples of suboptimal alignments of multiple protein sequences.

shortest path, ignore p and its all conceptual children (defined in subsection 3.1.1). Figure 3.5 shows the concept of this technique.

Recall that one of the good features of the Eppstein algorithm is that we can obtain the solutions in the time linear to the output size after having constructed the Eppstein heap. But the above technique is not output sensitive, because it requires checking some of the nodes which are in the class D_2. If there are many such solutions, the computing time becomes rather large. Thus we modified the Eppstein algorithm to overcome this problem. Here we show how to construct a heap structure whose nodes represent only alignments in classes D_0 (the root or the optimal solution) and D_1:

Algorithm 8 (Modified Eppstein Algorithm)

1. *Construct the shortest path tree from s to all the other vertices.*
2. *For each vertex v, construct $H_G(v)$, a 3-heap of sidetracks (u', u) ordered by $\delta(u', u)$ as follows. Let the length from the root of $H_G(v)$ to a node x be $\delta(u, v)$ if x represents sidetrack (u, v).*
   (a) *For each vertex v, construct $H_{out}(v)$, a 2-heap of sidetracks (v', v) ordered by $\delta(v', v)$ in which the root has only one child.*
   (b) *For each vertex v that is not on the s-t shortest path, construct $H_T(v)$, a 2-heap of vertices on the s-v shortest path but not on the s-t shortest path ordered by the value δ of the root of the heap made in step 2-(a).*

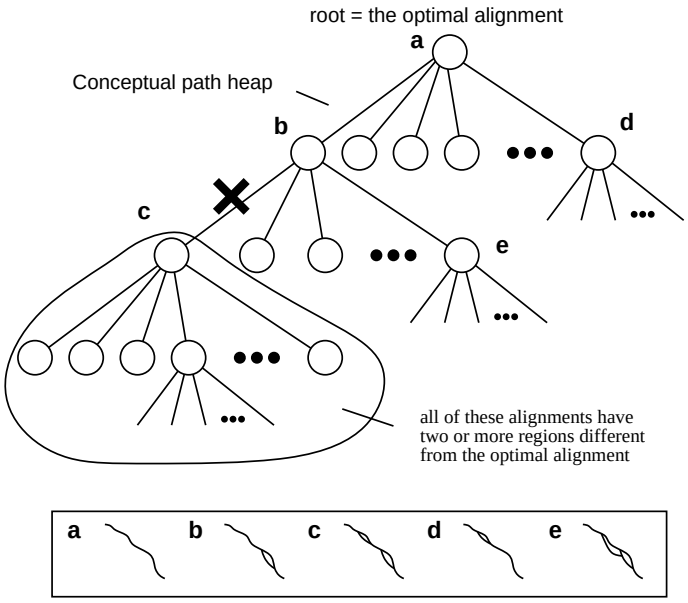


Figure 3.5: An example of a conceptual path heap. We can easily construct a heap which does not contain unnecessary alignments such as **c** and its all descendants.

(c) *For each vertex v on the s-t shortest path, construct $H_T(v)$, a 2-heap of vertices on the s-v shortest path ordered by the value δ of the root of the heap made in step 2-(a).*

(d) *Merge $H_{out}(v)$ and $H_T(v)$ to make $H_G(v)$. Then let the length of the edge from node x_1 to node x_2 in this heap be $\delta(x_2) - \delta(x_1)$.*

3. *For each v on the s-t shortest path, make a node N_v.*

4. *For each v in G, make an edge from each node in $H_G(v)$ which represents a sidetrack (u', u) to the root of $H_G(u')$ if u' is not on the s-t shortest path, and let the length of this new edge be the value of the root. Otherwise, make an edge from the node to $N_{u'}$.*

5. *Make a new node for each v in G, and make an edge from this node to the root of $H_G(v)$. Let the length of this edge be δ of the root. Let this new graph be $P(G)$.*

Once the heap was constructed, what we should do next is same as in the original Eppstein algorithm: we only have to search from the root of the virtual heap. Accordingly, we can efficiently enumerate only the alignments in class D_1, and apparently it is output sensitive algorithm once the heap is given. This algorithm can be also used with the A* algorithm using the techniques in subsection 3.1.3.

We can also extend this algorithm for enumeration of alignments in class D_i $(i \leq c)$ for any c in the same way, though the algorithm becomes much more complicated:

Algorithm 9

1. *Construct the shortest path tree from s to all the other vertices.*

2. *For each vertex v, construct heaps $H_G^{(i)}(v)$ ($0 \leq i < c$ for the vertices on the s-t shortest path, and $0 < i \leq c$ for the others), that are 3-heaps of sidetracks (u', u) as follows. Let the length from the root of $H_G^{(i)}(v)$ to a node x be $\delta(u, v)$ if x represents sidetrack (u, v).*

   (a) *For each vertex v, construct $H_{out}^{(i)}(v)$ ($0 \leq i < c$ for the vertices on the s-t shortest path, and $0 < i \leq c$ for the others), that are 2-heaps of sidetracks (v', v) ordered by $\delta(v', v)$ in which the root has only one child.*

   (b) *For each vertex v that is not on the s-t shortest path, construct $c - 2$ heaps $H_T^{(i)}(v)$ ($0 < i < c$), all of which are 2-heaps of vertices on the s-v shortest path ordered by the value δ of the root of the heap made in step 2-(a).*

   (c) *For each vertex v that is not on the s-t shortest path, construct heap $H_T^{(c)}(v)$, a 2-heap of vertices on the s-v shortest path but not on the s-t shortest path, ordered by the value δ of the root of the heap made in step 2-(a).*

   (d) *For each vertex v on the s-t shortest path, construct heaps $H_T^{(i)}(v)$ ($0 \leq i < c$), that are 2-heaps of vertices on the s-v shortest path ordered by the value δ of the root of the heap made in step 2-(a).*

   (e) *For each i and v, merge $H_{out}^{(i)}(v)$ and $H_T^{(i)}(v)$ to make $H_G^{(i)}(v)$. Then let the length of the edge from node x_1 to node x_2 in these heaps be $\delta(x_2) - \delta(x_1)$.*

3. *For each v on the s-t shortest path, make a node N_v.*

4. *For each v in G, make an edge from each node in $H_G^{(i)}(v)$ which represents a sidetrack (u', u) to the root of $H_G^{(i)}(u')$ if u' is not on the s-t shortest path, and let the length of this new edge be the value of the root. If u' is on the s-t shortest path, make an edge to the root of $H_G^{(i+1)}(u')$ if $i + 1 < c$, or to $N_{u'}$ if $i + 1 = c$.*

5. *Make a new node for each v on the s-t shortest path, and make an edge from this node to the root of $H_G^{(0)}(v)$. Let the length of this edge be δ of the root. Let this new graph be $P(G)$.*

This algorithm requires $O(c(n + m))$ time to construct the heap structure. In contrast, the branch-and-bound technique we mentioned at first in this subsection can deal with this kind of problem easily: we only have to remember how many branches from the s-t shortest path the parent path has, in searching suboptimal paths in the Eppstein heap structure. Thus, when c is large, which technique is more efficient may depend on cases. But, note that the problem whose c is large is not so important as that whose c is small, *i.e.* 1 or 2.

3.1.6 Extracting Knowledge from Eppstein Heap

As mentioned by Eppstein [49], the Eppstein heap has a good feature: some of numerical values for each suboptimal solution can be obtained in $O(1)$ time with some simple pre-process of $O(|E_\Delta|)$ time. In the case of the multiple alignment problem, these can be obtained in such an efficient way for example: the number of aligned groups in which all residues are same, the number of gaps, the score computed with another score

Table 3.1: Sequences of EF-TU and EF-1α to be aligned and their scores of pairwise sequence alignments. We use the top d sequences in this table in the experiments.

Sequences			Pairwise Scores						
Species	Protein	Length	Met	Tha	Thc	Sul	Ent	Pla	Sty
Halobacterium marismortui (Hal)	EF-TU	421	1329	1314	1221	1109	1099	1000	971
Methanococcus vannielii (Met)	EF-TU	428		1336	1247	1150	1176	1087	1045
Thermoplasma acidophilum (Tha)	EF-1α	424			1311	1261	1233	1063	1072
Thermococcus celer (Thc)	EF-1α	428				1132	1130	1049	991
Sulfolobus acidocaldarius (Sul)	EF-1α	435					1192	1131	1099
Entamoeba histolytica (Ent)	EF-1α	430						1584	1551
Plasmodium falciparum (Pla)	EF-1α	443							1636
Stylonychia lemnae (Sty)	EF-1α	446							

table, the length of the alignment, and so on. Our algorithm in the subsection 3.1.5 which enumerates only alignments in the class D_1 have the same feature too.

Let us consider the case of computing the number of gaps in each obtained suboptimal solution. Let the number of gaps p we want to know be $gaps(p)$. Then the algorithm will be like this:

Algorithm 10

1. *For each vertex v in V, compute the number of gaps contained in the part of the s-v shortest path. This can be done in $O(n)$ time, by the depth first search on the shortest path tree to t. Then let this value be $g(v)$.*

2. *For each sidetrack (u, v), compute following $\delta'(u, v)$:*

$$\delta'(u, v) = g(u) - g(v) \tag{3.3}$$

3. *When we obtained a path p (which represents some alignment), we compute the $gaps(p)$ as follows from the value of $gaps(parent(p))$.*

$$gaps(p) = gaps(parent(p)) + \delta'(sidetrack(p)) \tag{3.4}$$

In the same way, we can get many kinds of values for each solution. But note that there are some values that cannot be obtained in $O(1)$ time. For example, affine gap cost is one of them, though it is a very important value.

3.1.7 Experimental Results

In this subsection, we examine the efficiency of our approach and investigate the properties of suboptimal alignments through experiments. In the experiment, we used the PAM-250 matrix, and linear gap penalty bx where x is the gap length and b is the minimum value in the PAM-250 matrix, -8. All the experiments are done on a Sun Ultra 1 workstation with 128 megabyte memory.

Table 3.2: Searching time (sec) by the A* algorithm in the experiment on the d sequences of EF-TU and EF-1α. In case $d = 8$, the enhanced A* utilizing the optimal score is used. Note that only DP is used in case $d = 2$.

	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$
best score	1329	3970	7709	12314	18101	24912	33129
Pre-process DP	0.32	1.00	4.30	7.23	11.1	16.0	20.5
Search (optimal)	-	0.18	0.52	3.35	19.6	426	5427
Search ($\Delta = 10$)	-	0.18	0.60	3.63	20.9	439	5686
Search ($\Delta = 20$)	-	0.22	0.73	4.17	23.1	462	6735
Search ($\Delta = 30$)	-	0.27	0.93	5.00	26.9	498	8027
Search ($\Delta = 40$)	-	0.33	1.23	6.22	31.5	552	9623

Case with High Similarity

We first did experiments on a group of 8 sequences with high similarity in Table 3.1. According to it, the average scores per amino pair of these pairwise alignments are about 2.5 to 4. Add to this, the optimal score of the multiple alignment of all these 8 sequences is 33129 and its length is 456, thus the average score per amino pair of this alignment is $\frac{33129}{456 \cdot \binom{8}{2}} \approx 2.59$ (Table 3.1). These are higher than those in the next experiment. Figure 3.6 shows one of the optimal alignments of all the 8 sequences in Table 3.1. You can easily recognize the high similarity.

As for computing alignments of less than 8 sequences, we could apply the simple A* algorithm. However, for alignment of the 8 sequences, we used the upper bounding technique (enhanced A*) because 128 megabyte memory is not enough for computing with the simple A* algorithm: we used in the experiment the optimal solution as the upper bound to see the best efficiency of this enhanced algorithm. In any case, we used the upper bounding technique after the optimal solution is obtained.

According to Table 3.2, the DP takes a lot of time compared with the A* algorithm when d is small, but it is negligible when d is large. This table also shows that, the additional searching time required for computing suboptimal solutions is not so large as long as Δ is not much larger than in these experiments: it requires at most twice the time in total as in the case of computing only the optimal alignment in these experiments if $\Delta \leq 40$.

Figure 3.7, Table 3.3 and Table 3.4 show the results of enumerating the suboptimal alignments. Figure 3.7(a) shows that there are enormous number of suboptimal alignments, and the number increases exponentially as Δ increases. However, in Figure 3.7(b), we can see the number of suboptimal solutions is dramatically reduced by ignoring alignments in class D_i ($i \geq 2$). The number of the alignments enumerated in this way is only 0.0003% ($d = 4$) to 0.4% ($d = 8$) of all the alignments in case $\Delta = 30$ (see Table 3.4): it seems difficult to check significance of all the suboptimal alignments at most 10 worse than the optimal, but in our method, we can do it. Accordingly, the enumeration time is also reduced drastically(see Table 3.4).

Figure 3.9 shows an example of suboptimal alignments. The score of it is 33139, which is only 10 worse than the optimal. In this figure, * in the first line indicates the regions different from the optimal alignment

Table 3.3: Size of E_Δ and Eppstein's heap structure in the experiments on d sequences of EF-TU and EF-1α. The heap size in the table does not include the number of nodes made in step 4 of the Eppstein algorithm, which is same as $|E_\Delta|$.

Δ		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$
10	$\|E_{10}\|$	503	485	513	553	534	579	540
	heap size	437	277	411	503	454	674	404
20	$\|E_{20}\|$	1101	595	609	689	691	799	784
	heap size	7184	1010	1266	1701	1750	2604	2672
30	$\|E_{30}\|$	1447	946	817	901	864	1246	1316
	heap size	12983	4949	3417	3997	3594	7552	8539
40	$\|E_{40}\|$	2011	2528	1170	1249	1156	1973	2254
	heap size	17934	25861	8648	10036	7785	18407	23973

Table 3.4: Enumerating time (sec) when $\Delta = 30$ in the experiment on the d sequences EF-TU and EF-1α. (a) is the case enumerating all the suboptimal alignments, and (b) is the case enumerating alignments in class D_0 (optimal) and D_1. The time of constructing the Eppstein's heap structure is included in the time below.

		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$
(a)	#alignments	38047513	8804702	327522816	85923864	20689104	49633652	13857237
	time (sec)	152.87	54.98	1830.18	510.63	124.55	292.50	109.00
(b)	#alignments	6968	1695	1117	2176	1659	41791	60589
	time (sec)	0.23	0.13	0.32	0.95	2.10	9.83	29.62

in Figure 3.6. This alignment has 5 such regions, which means this alignment is in the class D_5. The existence of such alignments makes enumeration of suboptimal alignments more difficult. It is the reason that our enumeration approach which avoids enumerating such alignments is very efficient.

Observing Figure 3.7(a), the number of the suboptimal alignments seems to be similar and irrelevant to d. It is an interesting fact, but this comparison is unfair. The number must be compared between the cases which have same value of $\frac{\Delta}{\binom{d}{2}}$: we must consider Δ per amino pair. For example, it is all right to compare the case $\Delta = 28$ ($d = 8$) and the case $\Delta = 10$ ($d = 5$). In this case, the number of suboptimal alignments in the former case is $\frac{7168718}{16112} \approx 444.9$ times as that of the latter case.

According to Table 3.3, $|E_\Delta|$ and the size of the Eppstein heap for this size of Δ is not so large. Thus, the enumeration time in Table 3.4(b) is small, though it includes time for constructing the heap. In Figure 3.7(b), the number of suboptimal alignments in class D_1 increases much when $d = 2, 7, 8$ compared with other cases. The reason of this is seen in Table 3.3. The $|E_\Delta|$ in cases $d = 2, 7, 8$ is larger than the others: there may be many alignments which have a large region different from the optimal. On the other hand, in Figure 3.7(a), the number of the alignments in case $d = 4$ is large compared with others, but many of these must be

Table 3.5: Globin sequences to be aligned and their scores of pairwise sequence alignments.

globin		Length	Apl	Bus	Ct7	Ct3
Lumbricus terrestris - AIII	(Lum)	157	29	15	35	41
Aplysia limacina	(Apl)	146		126	177	140
Busycon canaliculatum	(Bus)	147			111	64
Chironomus thummi thummi - VIIA	(Ct7)	145				191
Chironomus thummi thummi - IIIa	(Ct3)	151				

combinations of small number of 'necessary' alignments.

Case with Low Similarity

We next did experiments on 5 globin sequences listed in Table 3.5. According to it, the average scores per amino acid pair of pairwise alignments of them are about 0.2 to 1.3. The score of the optimal multiple alignment of these 5 sequences is 543 and its length is 165, thus the average score per amid acid pair of this alignment is $\frac{543}{165 \cdot \binom{5}{2}} \approx 0.33$, which is lower than the previous case. Figure 3.10 shows the optimal alignment of the sequences in Table 3.5. It shows the dissimilarity compared with the case of the previous experiments.

Figure 3.8 and Table 3.6 show the result of the experiments. According to Table 3.6, the searching time by the simple A* algorithm is far longer than in the previous experiments for same d, though the length is short. It is because the estimator of the A* algorithm is not so powerful in case with low similarity.

According to Figure 3.8, the number of alignments in this experiment is also drastically reduced as in the previous experiments by ignoring alignments in class D_i ($i \geq 2$): the number of alignments in D_1 is only 0.004% of that of all the suboptimal alignments in case $d = 5$ and $\Delta = 20$.

As mentioned by Zuker [166], the alignments with low scores are not always insignificant. In general, if the lengths of sequences to be aligned are short, the size of E_Δ will be small. However, the size of E_Δ is larger than in the previous experiments for same d and Δ. Hence we can estimate that sequences we use in this experiment is not so significant as in the previous experiment. In this way, we can use the size of E_Δ as a factor of the significance of the alignment.

```
     1                                                                  80
Hal  MS-DEQHQNLAIIGHVDHGKSTLVGRLLYETGSVPEHVIEQHKEEAEEKGKGGFEFAYVMDNLAEERERGVTIDIAHQEF
Met  MAKTKPILNVAFIGHVDAGKSTTVGRLLLDGGAIDPQLIVRLRKEAEEKGKAGFEFAYVMDGLKEERERGVTIDVAHKKF
Tha  MASQKPHLNLITIGHVDHGKSTLVGRLLYEHGEIPAHIIEEYRKEAEQKGKATFEFAWVMDRFKEERERGVTIDLAHRKF
Thc  MAKEKPHINIVFIGHVDHGKSTTIGRLLFDTANIPENIIKKFE-EMGEKGK-SFKFAWVMDRLKEERERGITIDVAHTKF
Sul  MS-QKPHLNLIVIGHVDHGKSTLIGRLLMDRGFIDEKTVKEAEEAAKKLGKDSEKYAFLMDRLKEERERGVTINLSFMRF
Ent  MPKEKTHINIVVIGHVDSGKSTTTGHLIYKCGGIDQRTIEKFEKESAEMGKGSFKYAWVLDNLKAERERGITIDISLWKF
Pla  MGKEKTHINLVVIGHVDSGKSTTTGHIIYKLGGIDRRTIEKFEKESAEMGKGSFKYAWVLDKLKAERERGITIDIALWKF
Sty  MPKEKNHLNLVVIGHVDSGKSTSTGHLIYKCGGIDKRTIEKFEKEAAEMGKGSFKYAWVLDKLKAERERGITIDIALWNF
     81                                                                 160
Hal  STDTYDFTIVDCPGHRDFVKNMITGASQADNAVLVVAA-D---D-GV-QP-QTQEHVFLARTLGIGELIVAVNKMD-L-V
Met  PTAKYEVTIVDCPGHRDFIKNMITGASQADAAVLVVNVDDA--KSGI-QP-QTREHVFLIRTLGVRQLAVAVNKMD-T-V
Tha  ETDKYYFTLIDAPGHRDFVKNMITGTSQADAAILVISARDG--E-GV-ME-QTREHAFLARTLGVPQMVVAINKMDATSP
Thc  ETPHRYITIIDAPGHRDFVKNMITGASQADAAVLVVAV-T---D-GV-MP-QTKEHAFLARTLGINNILVAVNKMD-M-V
Sul  ETRKYFFTVIDAPGHRDFVKNMITGASQADAAILVVSAKKGEYEAGMSAEGQTREHIILSKTMGINQVIVAINKMDLADT
Ent  ETSKYYFTIIDAPGHRDFIKNMITGTSQADVAILIVAAGTGEFEAGISKNGQTREHILLSYTLGVKQMIVGVNKMD-A-I
Pla  ETPRYFFTVIDAPGHKDFIKNMITGTSQADVALLVVPADVGGFDGAFSKEGQTKEHVLLAFTLGVKQIVVGVNKMD-T-V
Sty  ETAKSVFTIIDAPGHRDFIKNMITGTSQADAAILIIASGQGEFEAGISKEGQTREHALLAFTMGVKQMIVAVNKMDDKSV
     161                                                                240
Hal  DYGESEYKQVVEEV-KDLLTQVRFDSENAKFIPVSAFEGDNIAEESEHTGWYDGEILLEALNELPAPEPPTDAPLRLPIQ
Met  NFSEADYNELKKMIGDQLLKMIGFNPEQINFVPVASLHGDNVFKKSERNPWYKGPTIAEVIDGFQPPEKPTNLPLRLPIQ
Tha  PYSEKRYNEVKADA-EKLLRSIGFK-D-ISFVPISGYKGDNVTKPSPNMPWYKGPTLLQALDAFKVPEKPINKPLRIPVE
Thc  NYDEKKFKAVAEQV-KKLLMMLGYK-N-FPIIPISAWEGDNVVKKSDKMPWYNGPTLIEALDQMPEPPKPTDKPLRIPIQ
Sul  PYDEKRFKEIVDTV-SKFMKSFGFDMNKVKFVPVVAPDGDNVTHKSTKMPWYNGPTLEELLDQLEIPPKPVDKPLRIPIQ
Ent  QYKQERYEEIKKEI-SAFLKKTGYNPDKIPFVPISGFQGDNMIEPSTNMPWYKGPTLIGALDSVTPPERPVDKPLRLPLQ
Pla  KYSEDRYEEIKKEV-KDYLKKVGYQADKVDFIPISGFEGDNLIEKSDKTPWYKGRTLIEALDTMQPPKRPYDKPLRIPLQ
Sty  NWDQGRFIEIKKEL-SDYLKKIWLQPRQDPFIPISGWHGDNMLEKSPNMPWFTGSTLIDALDALDQPKRPKDKPLRLPLQ
     241                                                                320
Hal  DVYTISGIGTVPVGRVETGILNTGDNVSFQPSD-V----S-GEVKTVEMHHEEVPKAEPGDNVGFNVRGVGKDDIRRGDV
Met  DVYTITGVGTVPVGRVETGIIKPGDKVVFEPAG-A----I-GEIKTVEMHHEQLPSAEPGDNIGFNVRGVGKKDIKRGDV
Tha  DVYSITGIGTVPVGRVETGVLKPGDKVIFLPAD-K----Q-GDVKSIEMHHEPLQQAEPGDNIGFNVRGIAKNDIKRGDV
Thc  DVYSIKGVGTVPVGRVETGVLRVGDVVIFEPASTIFHKPIQGEVKSIEMHHEPMQEALPGDNIGFNVRGVGKNDIKRGDV
Sul  EVYSISGVGVVPVGRIESGVLKVGDKIVFMPVG-K----I-GEVRSIETHHTKIDKAEPGDNIGFNVRGVEKKDVKRGDV
Ent  DVYKISGIGTVPVGRVETGILKPGTIVQFAPSG-V----S-SECKSIEMHHTALAQAIPGDNVGFNVRNLTVKDIKRGNV
Pla  GVYKIGGIGTVPVGRVETGILKAGMVLNFAPSA-V----V-SECKSVEMHKEVLEEARPGDNIGFNVKNVSVKEIKRGYV
Sty  DVYKIGGIGTVPVGRVETGLLKPGMVLTFAPMN-I----T-TECKSVEMHHESLTEAEPGDNVGFTVKNLSVKDLRRGYV
     321                                                                400
Hal  CGPADDPPSVA--ET-FQAQIVVMQHPSVITEGYTPVFHAHTAQVACTVESIDKKIDPSSGEVAE-ENPDFIQNGDAAVV
Met  LGHTTNPPTVA--TD-FTAQIVVLQHPSVLTDGYTPVFHTHTAQIACTFAEIQKKLNPATGEVLE-ENPDFLKAGDAAIV
Tha  CGHLDTPPTVV--KA-FTAQIIVLNHPSVIAPGYKPVFHVHTAQVACRIDEIVKTLNPKDGTTLK-EKPDFIKNGDVAIV
Thc  AGHTNNPPTVVRPKDTFKAQIIVLNHPTAITVGYTPVLHAHTLQVAVRFEQLLAKLDPRTGNIVE-ENPQFIKTGDSAIV
Sul  AGSVQNPPTVA--DE-FTAQVIVIWHPTAVGVGYTPVLHVHTASIACRVSEITSRIDPKTGKEAE-KNPQFIKAGDSAIV
Ent  ASDAKNQPAVG-CED-FTAQVIVMNHPGQIRKGYTPVLDCHTSHIACKFEELLSKIDRRTGKSMEGGEPEYIKNGDSALV
Pla  ASDTKNEPAKG-CSK-FTAQVIILNHPGEIKNGYTPLLDCHTSHISCKFLNIDSKIDKRSGKVVE-ENPKAIKSGDSALV
Sty  ASDSKNDPAKD-TTN-FLAQVIVLNHPGQIQKGYAPVLDCHTAHIACKFDEIESKVDRRSGKVLE-EEPKFIKSGEAALV
     401                                                                456
Hal  TVRPQKPLSIEPSSEIPELGSFAIRDMGQTIAAGKV--L---G---V-NE-----R
Met  KLIPTKPMVIESVKEIPQLGRFAIRDMGMTVAAGMA--I---Q---VTAKN----K
Tha  KVIPDKPLVIEKVSEIPQLGRFAVLDMGQTVAAGQC--I---D---L-EK-----R
Thc  VLRPTKPMVIEPVKEIPQMGRFAIRDMGQTVAAGMV--I---S---I-QKA----E
Sul  KFKPIKELVAEKFREFPALGRFAMRDMGKTVGVGVI--I---D---VKPRKVE-VK
Ent  KIVPTKPLCVEEFAKFPPLGRFAVRDMKQTVAVGVV--K---A---V-T------P
Pla  SLEPKKPMVVETFTEYPPLGRFAIRDMRQTIAVGIINQLKRKNLGAVTAKAPA-KK
Sty  RMVPQKPMCVEAFNQYPPLGRFAVRDMKQTVAVGVIKEVVKKEQKGMVTKAAQKKK
```

Figure 3.6: The optimal alignment of the 8 EF-TU and EF-1α sequences.

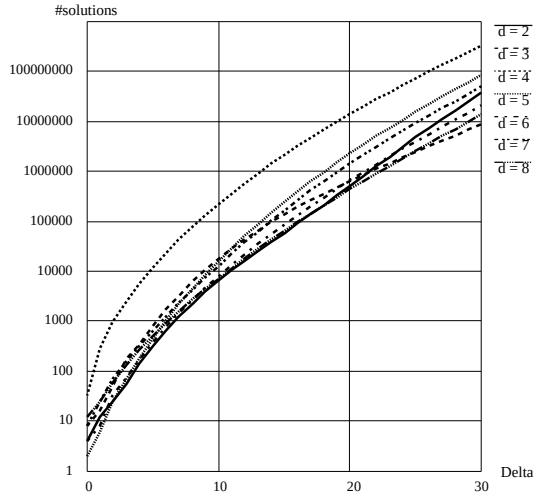
Table 3.6: The best score, searching time (sec) by the simple A* algorithm and the size of E_Δ in the experiment on the d globin sequences.

(1)

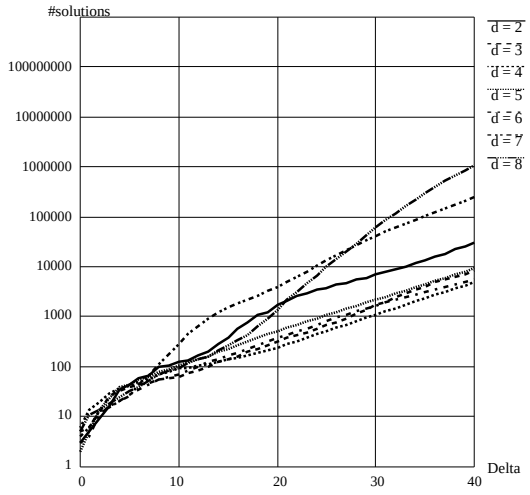
	$d=2$	$d=3$	$d=4$	$d=5$
best score	29	103	354	543
Pre-Process DP	0.05	0.27	0.52	0.83
Search (optimal)	-	0.73	7.40	837
Search ($\Delta=10$)	-	0.85	8.13	865
Search ($\Delta=20$)	-	0.93	9.43	909
Search ($\Delta=30$)	-	1.22	11.22	964
Search ($\Delta=40$)	-	1.63	14.13	1080

(2)

	$d=2$	$d=3$	$d=4$	$d=5$
$\lvert E_{10}\rvert$	357	508	380	554
$\lvert E_{20}\rvert$	776	1184	866	1159
$\lvert E_{30}\rvert$	1149	2403	1575	2448
$\lvert E_{40}\rvert$	1544	3839	2669	4569

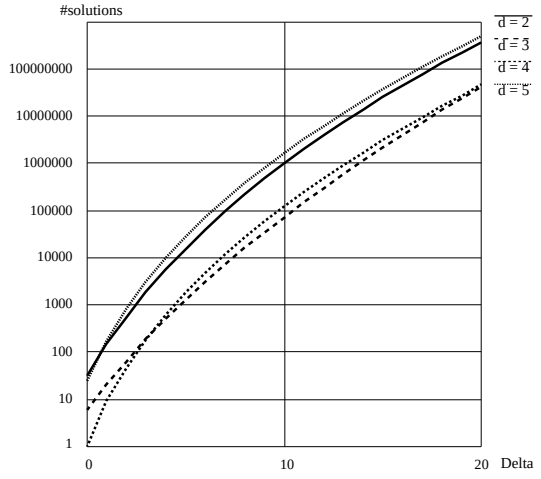


(a) enumerating all the alignments

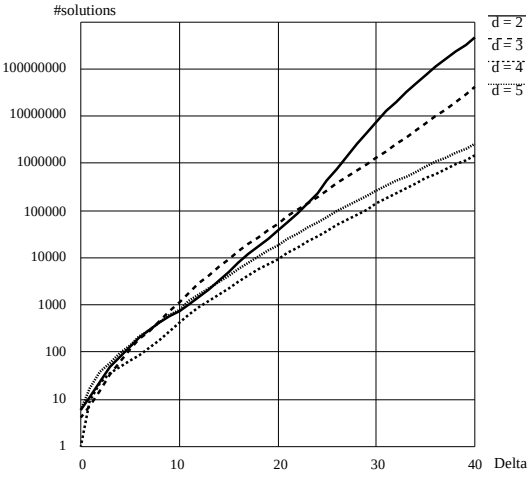


(b) enumerating alignments in classes D_0 and D_1

Figure 3.7: Number of the suboptimal alignments of d sequences of EF-TU and EF-1α whose scores are at most Δ worse than the optimal. (a) is the case enumerating all the alignments. $(0 \leq \Delta \leq 30)$ (b) is the case enumerating alignments in classes D_0 and D_1. $(0 \leq \Delta \leq 40)$



(a) enumerating all the alignments



(b) enumerating alignments in classes D_0 and D_1

Figure 3.8: Number of the suboptimal alignments of d globin sequences whose scores are at most Δ worse than the optimal alignment. (a) is the case enumerating all the alignments. $(0 \leq \Delta \leq 20)$ (b) is the case enumerating alignments in classes D_0 and D_1. $(0 \leq \Delta \leq 40)$

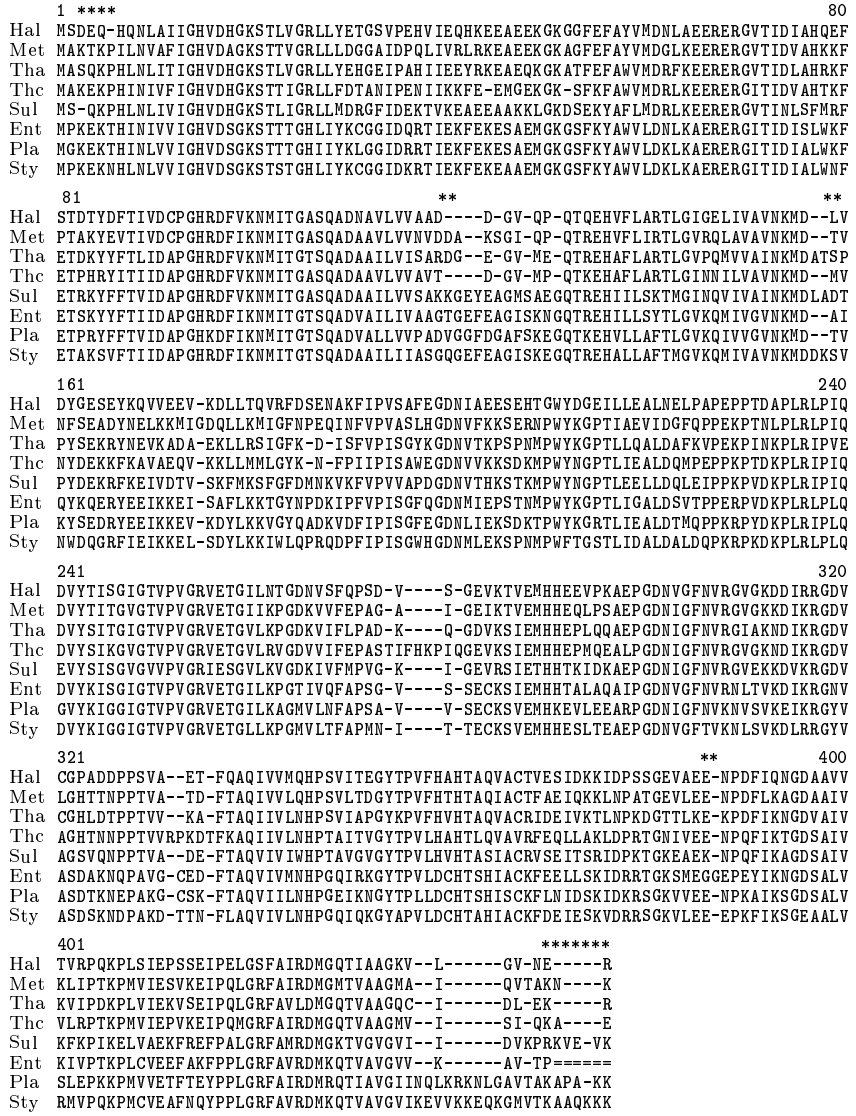


Figure 3.9: An example of suboptimal alignments of the 8 EF-TU and EF-1α sequences.

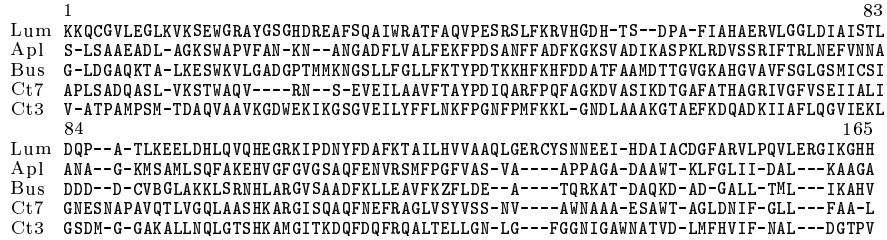


Figure 3.10: One of the optimal alignments of the 5 globin sequences.

3.2 Parametric Analysis of Multiple Sequence Alignment

In this section, we describe the techniques for parametric analysis of the multiple sequence alignment problem.

3.2.1 Basic Techniques

In this subsection, we describe basic methods to check how the optimal solution varies as the parameters such as gap penalties change. The easiest approach for this kind of problem is to change the parameter little by little and check the optimal solution, but we cannot know how little we should change the parameter. Recently the techniques for parametric analysis were developed [74, 83, 157, 159, 160, 162, 165]. Some of them did parametric analysis on more than one parameter, but algorithms for it are not so efficient as those for the one-parameter problems. Hence we deal with only one parameter at one time in this thesis. We consider the case in which the score of some alignment A_i is expressed with parameter p as follows:

$$s_i(p) = a(A_i) + b(A_i) \cdot p \tag{3.5}$$

Gap penalty satisfies this expression for example.

From now on, we explain how to divide 1-parameter (1-dimensional) space to regions in which the optimal alignments are always same. Let a_i be $a(A_i)$ and b_i be $b(A_i)$. Let p_i and p_j be the values of the parameter which satisfies $p_i < p_j$ and have different optimal solutions. Let the alignment A_i be the alignment with the smallest value of b among the optimal alignments at $p = p_i$ and A_j be the alignment with the largest value of b among the optimal alignments at $p = p_j$. Then these two alignments A_i and A_j have the same score at $p = p_{ij} = -\dfrac{a_i - a_j}{b_i - b_j}$. If the optimal score at $p = p_{ij}$ equals $s_i(p_{ij}) = s_j(p_{ij})$, there are only two regions between p_i and p_j. Otherwise, we can apply the same technique recursively (*i.e.* apply between p_i and p_{ij} and between p_{ij} and p_j) to obtain such division. Figure 3.11 shows an example of this procedure. Letting n be the number of regions which we want to obtain, we only have to compute the optimal solutions $2n - 1$ times. Thus we can efficiently do parametric analysis in the case of one parameter.

In the algorithm, the alignments with the largest or smallest value of b among the optimal alignments at some fixed parameter are required. These can be easily obtained by some lexicographical extension of DP [157, 159, 160, 162]. This technique can be applied also to the (enhanced) A* algorithm. But the aim of the parametric analysis is to examine all the optimal solutions, and it is not preferable to ignore most optimal solutions if there are many. Thus, for this purpose, we enumerate the optimal solutions by the Eppstein algorithm in the subsection 3.1.1 and pick up the solutions with the largest or smallest value of b.

3.2.2 Upper Bounding Technique for Parametric Alignment

As we stated in the subsection 2.1.2, the A* algorithm will be more efficient if some upper bounding value for the optimal solution is given (it is called the enhanced A* algorithm). In the parametric alignment problem, $s_i(p_{ij}) = s_j(p_{ij})$ in the subsection 3.2.1 is a lower bound of the score of the optimal alignment at $p = p_{ij}$. Thus it can be used as the upper bounding value for computing the optimal alignments at $p = p_{ij}$ with the A* algorithm. Note that the enhanced A* algorithm will show the best performance if the $s_i(p_{ij}) = s_j(p_{ij})$ is the optimal score at $p = p_{ij}$, which always happens at the final stage of the parametric analysis.

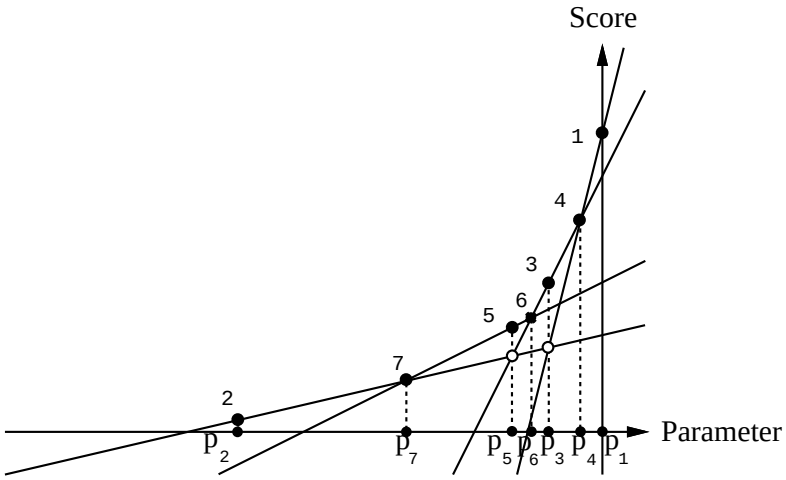


Figure 3.11: An example of division of 1-parameter space. In this case, there are 4 regions between p_1 and p_2.

3.2.3 Experimental Results

In this subsection, we do parametric analysis on groups of actual protein sequences. We use the PAM-250 matrix as the score matrix in the following experiments except for the experiments on the parametric weight matrix. The experiments on the parametric gap penalty and the parametric score matrix are done on a Sun Ultra 1 workstation with 128 megabyte memory, while the experiments on the parametric weight matrix are done on a Sun Ultra 1 workstation with 256 megabyte memory.

Parametric Gap Penalty

Previous work on parametric analysis mainly dealt with gap penalty, because it is a very important factor of the alignment problem [74, 83, 157, 159, 162]. But the previous work dealt only with the 2-alignment problem. We did parametric analysis of gap penalty using the top d sequences ($d < 7$) in Table 3.1. In general, the most popular gap penalty is the minimum value in the score matrix, which is -8 in this PAM-250 case. We did parametric analysis for d-sequence alignment ($2 \leq d \leq 6$) with the gap penalty between -2 and -16.

Table 3.7 shows the result of the experiment. In Table 3.7, the first row of each entry of d shows the boundaries of the regions, but several of the ends are not the boundaries: the ends with $-$ in #Max and #Min entry are not boundaries. The second row shows the numbers of the optimal solutions at the value. The last two rows show the numbers of optimal solutions with the largest/smallest value of b in the subsection 3.2.1. Thus, these values equal the number of the optimal solutions between the boundaries.

According to the table, it is observed that the intervals of the regions become smaller (*i.e.* the optimal solution is not stable), as the penalty increases regardless of d. It also shows that there are not so much difference between different d's, which means we can do parametric analysis as easily as in the 2-alignment case. The table also shows that there are more than 1 optimal solution in all cases in the experiments.

Table 3.7: The result of the experiment on parametric gap penalty using EF-1α sequences.

$d = 2$	Gap penalty	-16	-5	-3	-2.5	-2			
	#Solutions	4	12	24	192	576			
	#Max	-	8	16	8	32			
	#Min	-	4	8	16	8			
$d = 3$	Gap penalty	-16	-3.5	-3	-2.75	-2.5	-2.2	-2	
	#Solutions	8	16	24	32	72	48	256	
	#Max	-	8	16	16	16	32	96	
	#Min	-	8	8	16	16	16	32	
$d = 4$	Gap penalty	-16	-8	-3.83	-3.5	-2.5	-2.33	-2.25	-2
	#Solutions	16	32	32	32	32	48	160	4608
	#Max	-	16	16	16	16	32	128	384
	#Min	-	16	16	16	16	16	32	128
$d = 5$	Gap penalty	-16	-7.5	-4	-3.38	-3.17	-3	-2.88	-2.75	-2.5
	#Solutions	2	4	4	4	4	4	12	8	24
	#Max	-	2	2	2	2	2	4	4	4
	#Min	-	2	2	2	2	2	2	4	4
$d = 6$	Gap penalty	-16	-6.5	-4.5	-4	-3.5				
	#Solutions	4	16	8	8	4				
	#Max	-	4	4	4	-				
	#Min	-	4	4	4	-				

Figure 3.12 shows the number of visited nodes by the A* algorithm in computing all the optimal alignments under various gap penalties. According to this figure, the number of the visited nodes increases drastically as the gap penalty increases especially when gap penalty is larger than -4. This is the reason why we analyzed gap penalty only up to -2.5 or -3.5 when $d \geq 5$: the required space was too large to compute when the gap penalty is around -2. We suppose this increase of the search space is caused by the instability of the optimal solution. In general, it is known that the required space for the A* algorithm is large if the similarity among the group is low, probably for the same origin.

We also did experiments on 10 TNF-α sequences in Table 3.8, which are very similar to each other. Table 3.10 shows the result of it. In the experiments, we examined between gap penalties -16 and -2.5, for $d = 2, 4, 6, 8, 10$ sequences. It is between -16 and -4.5 in case of $d = 10$, because of computational difficulty of obtaining the optimal solution with gap penalties near to 0. There are fewer regions than the EF-1α case, whose reason may be the high similarity among this group.

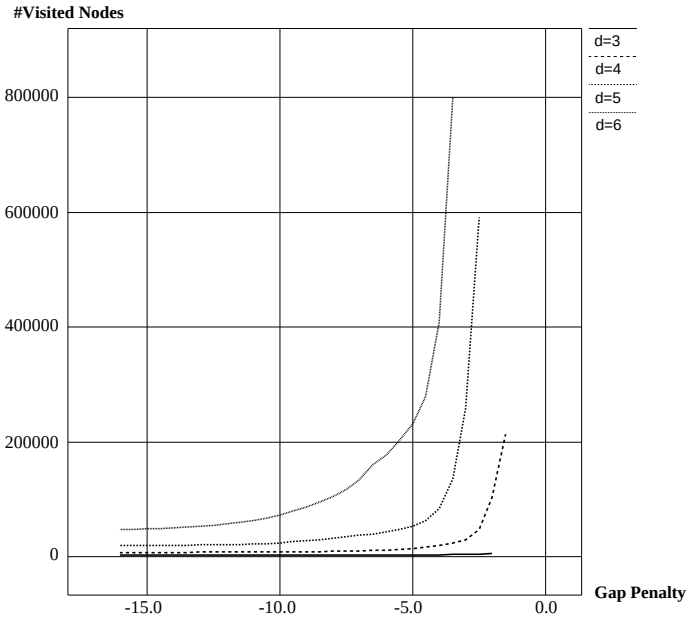


Figure 3.12: Number of visited nodes by the A* algorithm for various gap penalties.

Table 3.8: TNF-α sequences used in the experiment.

Species	Protein	Length
Homo sapiens (HSp)	tumor necrosis factor α (TNF-α) precursor	233
Mus musculus (MM)	tumor necrosis factor α (TNF-α) precursor	235
Sus scrofa (SS)	tumor necrosis factor α (TNF-α) precursor	232
Ovis orientalis aries (OOAp)	tumor necrosis factor α (TNF-α) precursor	234
Bos primigenius taurus (BPT)	tumor necrosis factor α (TNF-α) inhibitor	233
Equus caballus (EC)	tumor necrosis factor α (TNF-α) precursor	234
Oryctolagus cuniculus (OC)	tumor necrosis factor α (TNF-α) precursor	234
Rattus norvegicus (RN)	tumor necrosis factor α (TNF-α) precursor	235
Homo sapiens (HSi)	tumor necrosis factor α (TNF-α) inhibitor	233
Ovis orientalis aries (OOAi)	tumor necrosis factor α (TNF-α) inhibitor	233

Table 3.9: Scores of pairwise alignments of TNF-α sequences.

	MM	SS	OOAp	BPT	EC	OC	RN	HSi	OOAi
HSp	955	990	924	935	1013	949	943	1041	932
MM		919	872	872	945	952	1124	952	866
SS			980	990	982	906	912	974	988
OOAp				1059	916	840	856	915	1145
BPT					933	865	854	926	1067
EC						962	931	1013	906
OC							946	961	845
RN								940	850
HSi									922

Table 3.10: The result of the experiment on parametric gap penalty using TNF-α.

$d = 2$	Gap penalty	-16	-2.5				
	#Solutions	6	18				
	# Max	-	12				
	# Min	-	6				
$d = 4$	Gap penalty	-16	-6.67	-3.75	-2.88	-2.67	-2.5
	#Solutions	12	24	24	24	24	36
	# Max	-	12	12	12	12	24
	# Min	-	12	12	12	12	12
$d = 6$	Gap penalty	-16	-4.95	-3.3	-2.75	-2.5	
	#Solutions	6	12	18	24	36	
	# Max	-	6	12	12	24	
	# Min	-	6	6	12	12	
$d = 8$	Gap penalty	-16	-5.57	-4.5	-3.36	-2.92	-2.75	-2.5
	#Solutions	1	2	2	3	4	4	2
	# Max	-	1	1	2	2	2	-
	# Min	-	1	1	1	2	2	-
$d = 10$	Gap penalty	-16	-5.28	-4.5			
	#Solutions	1	2	1			
	# Max	-	1	-			
	# Min	-	1	-			

Parametric Score Matrix

We next deal with the parametric analysis of a score matrix. A score matrix is a set of very important parameters and it deeply affects the quality of the output alignment. Thus parametric analysis of it is very important and requires much care to deal with.

There are $\frac{(n+2)(n-1)}{2}$ parameters in the score matrix where n is the number of characters, thus what we can do is very limited simple analysis. In previous work, Vingron et al. [157] analyzed the varying solutions as they added some constant (which is the parameter to change) to each element in the score matrix. But it is not so interesting because those constants do not have much meaning. We should do analysis in more practical and interesting way.

We implemented a program to analyze how the optimal solutions will change as the score matrix changes linearly between two score matrices $(s_{ij}^{(1)})$ and $(s_{ij}^{(2)})$: we used $(p \cdot s_{ij}^{(1)} + (1-p) \cdot s_{ij}^{(2)})$ as the score matrix and considered p as the parameter to be changed. Note that our program can of course deal with the analysis what Vingron et al. did [157].

At first, we did experiments on 8 sequences (fragments) of the rhodopsin superfamily shown in Table 3.11. Figure 3.13 shows the optimal alignment of 5 sequences taken from Table 3.11. But, in the biologists' view, it is not the best alignment [29, 77]. According to them, the K's (lysine) which are marked with * at the first line in Figure 3.13 must be aligned.

There must be various methods to cope with this kind of problems, but one of the simplest methods is to check the optimal alignment obtained with a score table whose score for K-K is large. But, too large score for K-K makes irrelevant K's aligned and the alignment will be unnatural. Thus, we should choose as small score as possible in the constraint that we can obtain a biologically good solution. In this way, we can align reasonably these K's. In this point of view, we did parametric analysis on these sequences, letting the score for K-K be the parameter to change. In the experiment, we let the scores other than it are the same as those in the PAM-250 score matrix in Table 2.1. In the PAM-250 score matrix, the score for K-K is 5, hence we only examined in the region where its score is larger than 5.

Table 3.13 shows the result of this experiment. The first row of each entry shows the score for K-K which is a boundary of regions except for several ends with - in #Max and #Min entries. The second row shows the number of the optimal solutions and the other two rows show the number of the optimal solutions with the largest/smallest b in the subsection 3.2.1. In the table, * denotes the smallest score for K-K which induces the optimal alignment whose K's are aligned desirably. Figure 3.14 shows the alignment of 5 sequences obtained with the smallest score for K-K such that K's are aligned desirably. This alignment is biologically much better than that obtained with the ordinary PAM-250 score matrix.

Figure 3.15 shows how the number of visited nodes by the A* algorithm changes as the score for K-K changes. This figure shows the case of 4 and 5 sequences. In this figure, we can easily see that the number is large where the boundary in the parametric analysis exists. Thus we can estimate that computing the optimal solution will be more difficult if the solution is instable.

It is also interesting that in the case of the alignment of 4 rhodopsin sequences, the number of visited nodes is fixed when the score for K-K is larger than 84. It is because all the possible K's are aligned and thus letting the score larger makes no difference.

Table 3.11: Sequences of the Rhodopsin Superfamily used in the experiments.

Species		Protein	Length
Halobacterium sp.	(Hal)	Halorhodopsin precursor	39
Homo Sapiens	(HSg)	Green-sensitive opsin	48
Homo Sapiens	(HSr)	Red-sensitive opsin	48
Gallus gallus	(GGd)	Rhodopsin	55
Homo Sapiens	(HSb)	Blue-sensitive opsin	54
Homo Sapiens	(HSd)	Rhodopsin	55
Drosophila melanogaster	(DM3)	Opsin RH3	53
Drosophila melanogaster	(DM4)	Opsin RH4	53

Table 3.12: Scores of pairwise sequences of the Rhodopsin Superfamily.

	HSg	HSr	GGd	HSb	HSd	DM3	DM4
Hal	-35	-32	-79	-61	-83	-31	-26
HSg		256	108	100	111	2	-10
HSr			107	100	110	7	-5
GGd				147	275	30	18
HSb					150	34	35
HSd						33	21
DM3							277

PAM-t score table is computed based on the evolutionary permutation probability of amid acids per period proportional to t [44], and we can consider various PAM score tables such as PAM-120, PAM-250 and so on. As Altschul [4] suggested that it is better to align sequences with two or three different PAM scores based on statistical analysis, parametric analysis on t is very useful. Because PAM-t converges into some matrix as t becomes larger, parametric analysis on t can be approximated by the linear parametric analysis between PAM matrices if t is large enough. Note that the exact parametric analysis on t is difficult and remains as one of future tasks.

Table 3.14 shows the result of the experiment using 6 TNF-α sequences in Table 3.8. The first row of

```
     1                *       *                          55
Hal  ==GLALVQSVGVTSWAYS-VLDVF-AK-YVF---AF-I-LLR-WV-ANN---ER=
HSg  NPGYPFHPLMAALPAFFAKSATIYNPVIYVFMNRQFRNCILQ-LF-GKK----V=
HSr  NPGYAFHPLMAALPAYFAKSATIYNPVIYVFMNRQFRNCILQ-LF-GKK----V=
GGd  NQGSDFGPIFMTIPAFFAKSSAIYNPVIYIVMNKQFRNCMITTLCCGKNPLGDED
HSb  NRNHGLDLRLVTIPSFFSKSACIYNPIIYCFMNKQFQACIMK-MVCGKAMTDESD
```

Figure 3.13: The optimal alignment of 5 rhodopsin sequences based on PAM-250.

```
      1                    *                                   60
Hal   GLALVQSVGVTSWAYSVLDVFAK--YVF-A--FILL-R--W---VAN---NER=======
HSg   N----PGYPFHPLMAALPAFFAKSATIYNPVIYVFMNRQFRNC-ILQLF-GKK-V=====
HSr   N----PGYAFHPLMAALPAYFAKSATIYNPVIYVFMNRQFRNC-ILQLF-GKK-V=====
GGd   N----QGSDFGPIFMTIPAFFAKSSAIYNPVIYIVMNKQFRNCMITTLCCGKNPLGDE-D
HSb   N----RNHGLDLRLVTIPSFFSKSACIYNPIIYCFMNKQFQAC-IMKMVCGKA-MTDESD
```

Figure 3.14: The optimal alignment of 5 rhodopsin sequences when the score for K-K is between 89 and 124.

Table 3.13: The result of the experiment on parametric score matrices using rhodopsin sequences. * denotes the smallest score for K-K which induces the optimal alignment whose K's are aligned desirably.

$d=2$	Score of K-K	5	53*	128		
	#Solutions	1	6	5		
	#Max	-	1	-		
	#Min	-	5	-		
$d=3$	Score of K-K	5	55.5	128		
	#Solutions	2	7	5		
	#Max	-	2	-		
	#Min	-	5	-		
$d=4$	Score of K-K	5	38	58.5*	128	
	#Solutions	4	36	48	12	
	#Max	-	32	12	-	
	#Min	-	4	32	-	
$d=5$	Score of K-K	5	24	89*	124	128
	#Solutions	24	32	20	28	16
	#Max	-	8	12	16	-
	#Min	-	24	8	12	-

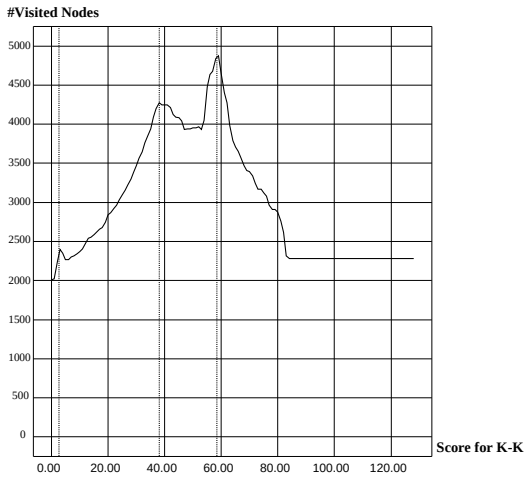
$d=6$	Score of K-K	5	20	109.5*	128
	#Solutions	24	28	6	2
	#Max	-	4	2	-
	#Min	-	24	4	-
$d=7$	Score of K-K	5	16	33.33	64
	#Solutions	4	8	6	2
	#Max	-	4	2	-
	#Min	-	4	4	-
$d=8$	Score of K-K	5	16.5	32	
	#Solutions	8	16	8	
	#Max	-	8	-	
	#Min	-	8	-	

each entry shows the value p of $(p \cdot s_{ij}^{(1)} + (1-p) \cdot s_{ij}^{(1)})$ which is a boundary of regions except for several ends with - in #Max and #Min entries, where $(s^{(1)})$ is the PAM-250 score matrix and $(s^{(2)})$ is the PAM-320 matrix. The second row shows the number of the optimal solutions and the other two rows show the number of the optimal solutions with the largest/smallest b in the subsection 3.2.1. Notice that there are no other optimal solutions other than those at both ends between the PAM-250 score matrix and the PAM-320 score matrix.

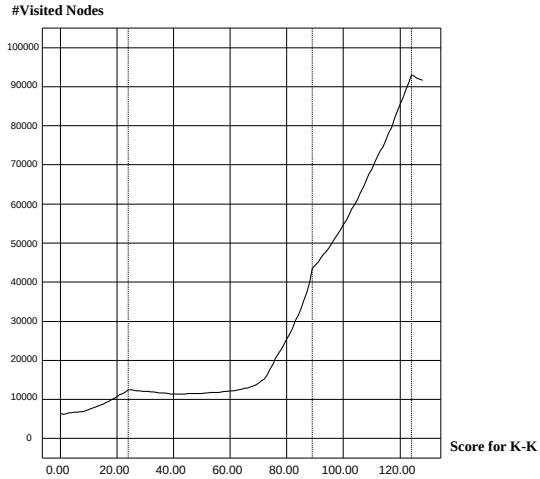
Parametric Weight Matrix

We next deal with a weighted version of the multiple alignment problem. Computing the optimal solution of this problem by the (enhanced) A* algorithm is rather easy: all we have to do is using $h(v) = \sum_{1 \le i < j \le d} w_{ij} \cdot L^*(u_{ij}, v_{ij})$ as the estimator. Thus we can use the same techniques as in the previous experiments.

A weight matrix for aligning sequences whose phylogenetic tree is known can be made if divergence between sequences are given [5]. But what should we do if the divergence are ambiguous? In such a case,



(a) Case of the alignment of 4 rhodopsin sequences



(a) Case of the alignment of 5 rhodopsin sequences

Figure 3.15: Number of visited nodes by the A* algorithm for various scores for K-K. The dotted lines denote the boundaries of the regions where the optimal solution is same.

Table 3.14: The result of the linear parametric analysis between PAM-250 and PAM-320 using 6 TNF-α sequences.

$d = 6$	p	0	0.5	1
	#Solutions	6	12	9
	#Max	-	6	3
	#Min	-	6	6

parametric analysis between reasonable two weight matrices helps. Thus, parametric analysis on weight matrices may help tuning parameters of a phylogenetic tree.

There are $\frac{(d-2)(d+1)}{2}$ parameters to change in the weight matrix, thus what we can do is very limited simple analysis. We implemented a program to analyze how the optimal solutions change as the weight matrix changes linearly between two weight matrices.

We did experiments between following two weight matrices of (w_{ij}) and $(w_{ij}^{(16,n)})$:

$$w_{ij} = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases} \tag{3.6}$$

$$w_{ij}^{(p,n)} = \begin{cases} p \cdot w_{ij} & i = n \text{ or } j = n \\ w_{ij} & \text{otherwise} \end{cases} \tag{3.7}$$

In this equation, (w_{ij}) corresponds to the simple sum-of-pairs multiple alignment, and $w_{ij}^{(p,n)}$ increases the importance of nth sequence to p times as the simple sum-of-pairs multiple alignment. If biologically good alignment is discovered in the experiment, we can estimate the importance of the sequence which was increased.

Table 3.15: The result of the experiment on parametric weight matrices using EF-1α sequences.

Hal	Weight	1	1.33	3	3.17	4.5	4.75	5	6.57	16	
	#Solutions	4	16	16	12	8	8	48	16	8	
	#Max	-	4	8	4	4	4	8	8	-	
	#Min	-	4	4	8	4	4	4	8	-	
Met	Weight	1	2.25	3	4.2	4.4	4.5	12	14.5	16	
	#Solutions	4	8	12	16	16	16	16	16	24	
	#Max	-	4	8	8	8	8	8	8	16	
	#Min	-	4	4	8	8	8	8	8	8	
Tha	Weight	1	2.33	3	5.5	7.67	8	16			
	#Solutions	4	8	8	8	8	16	4			
	#Max	-	4	4	4	4	4	-			
	#Min	-	4	4	4	4	4	-			
Thc	Weight	1	1.8	3	4	5	5.33	6	10.33	12	14.44	16
	#Solutions	4	8	8	12	12	8	8	8	8	8	12
	#Max	-	4	4	2	4	4	4	4	4	4	8
	#Min	-	4	4	4	2	4	4	4	4	4	4
Sul	Weight	1	2	3	3.75	5	6	6.43	8	10.25	14	16
	#Solutions	4	12	8	8	12	16	16	16	16	16	8
	#Max	-	4	4	4	8	8	8	8	8	8	-
	#Min	-	4	4	4	4	8	8	8	8	8	-
Ent	Weight	1	1.33	1.5	3	3.66	5	6	10.33	13	16	
	#Solutions	4	8	8	8	8	8	8	12	24	16	
	#Max	-	4	4	4	4	4	4	8	16	-	
	#Min	-	4	4	4	4	4	4	4	8	-	

Table 3.15 shows experimental results using the top 6 EF-1α sequences in Table 3.1. The first column is the name of the sequence whose importance was increased. The first row of each entry shows the value of p of $w_{ij}^{(p,n)}$ which is a boundary of regions except for several ends with - in #Max and #Min entries. The second row shows the number of the optimal solutions and the other two rows show the number of the optimal solutions with the largest/smallest b in the subsection 3.2.1.

In this experiment, we notice that the optimal solutions will change even when only $p = 1.33$ in some of the cases (cases of **Tha** and **Ent**). It means we should take more care of the weight matrix. This experiment also show that there are more than 1 optimal solution in all the cases in this experiment. In the experiment, the numbers of the regions are not too large to deal with (6 to 10 in this experiment), which means this approach is very reasonable.

We also did experiments on 5 EF-2 sequences in Table 3.16, which are very similar to each other and very long. The experiments are done also between weight matrices of (w_{ij}) and $(w_{ij}^{(16,n)})$. In the experiments, we increased importance of one of the sequences. Table 3.17 shows the results.

Table 3.16: EF-2 (translation elongation factor eEF-2) sequences to be aligned and their pairwise scores.

Sequences			Pairwise Scores			
Species	Protein	Length	CGS	DM	DD	SC
Homo sapiens (HS)	eEF-2	858	4216	3409	2463	2941
Cricetinae gen. sp. (CGS)	eEF-2	858		3392	2456	2931
Drosophila melanogaster (DM)	eEF-2	844			2416	2959
Dictyostelium discoideum (DD)	eEF-2	830				2446
Saccharomyces cerevisiae (SC)	eEF-2	842				

Table 3.17: The result of the experiment on parametric weight matrices using EF-2.

HS	Weight	1	1.33	2	3	3.14	6	9	16
	#Solutions	16	32	32	16	16	16	24	24
	#Max	-	16	8	8	8	8	16	-
	#Min	-	16	16	8	8	8	8	-

3.3 Summary

In the multiple alignment problem, it is often said that the optimal solution obtained with only the scoring criteria is not always the biologically best one. Thus, in this chapter, we took two flexible approaches to overcome this problem. One approach is by suboptimal analysis and the other is by parametric analysis.

We first investigated methods for enumerating suboptimal alignments of multiple sequences. We discussed a method to reduce the memory size for the Eppstein algorithm, an algorithm for enumerating suboptimal shortest paths. We then classified the suboptimal alignments into classes D_i based on the number of the different regions from the optimal solution. We showed that the suboptimal alignments in classes D_i ($i \geq 2$) can be easily constructed from the alignments in the class D_0 (the optimal solution) and the class D_1, and suggested that the alignments in the classes D_i ($i \geq 2$) is not necessary to enumerate. We proposed an algorithm to enumerate only the suboptimal alignments in the class D_1. Furthermore, we did experiments using actual protein sequences to see the efficiency and the property of our algorithms.

Next, we investigated parametric optimization of the multiple alignment problem. We discussed how to use an upper bounding technique of the enhanced A* algorithm in the parametric analysis. We did parametric experiments on various parameters and examined the property of the multiple alignments in the view of parametric optimization.

Chapter 4

Accurate cDNA Clustering based on Spliced Alignment

Full-length cDNA library collections are biologically important data sources because they represent the transcriptome of an organism. High-throughput sequencing, sequence clustering, and analyses enable the rapid, computational assignment of gene names and inference of functions. Yet, the detection of alternative splice candidates is not straightforward and depends on the accuracy of sequence clustering and manual inspection. Recent EST-based analyses by [89] and [106] estimate that 42%-55% of human genes are alternatively spliced. Bioinformatics-driven studies of alternative splicing is very important for a number of reasons such as frequently detected single-base substitutions in human mRNA splice junctions and the association with 1 out of 6 genetic diseases [96]. Therefore effective and accurate methods for clustering cDNA sequences into alternative splice form candidates are highly desired, which is the aim of the algorithms presented in this chapter.

Before proceeding, we introduce several basic terms. A pair of sequences is called a *splicing pair* if one sequence is produced from the other by the splicing mechanism. The sequence of the splicing pair that is spliced in the alignment to produce the other sequence is designated the *template sequence*, while the other sequence is called the *spliced sequence*. An *alternative splice form* is a group of cDNA sequences that are transcribed from the same gene region of the genomic DNA sequence.

There have been several reports on cDNA sequence clustering algorithms [30, 34, 36, 93, 104, 116]. However all of these methods are based on ordinary similarity measures and do not take the splicing mechanism into account. As a result, these methods often cluster non-splicing pairs of sequences into one group merely because they are similar. Clustering of non-splicing sequences has been frequently observed among repeat elements of eukaryotic genomes. Although heuristic filters such as the RepeatMasker [148] remove false positives, they may also filter out actual splicing pairs. The method described here takes into account the splicing mechanism and yields more accurate clusters. To do so, we use an alignment technique called the spliced sequence alignment algorithm or more conveniently the *spliced alignment algorithm*. There is no previous work on cDNA clustering that uses the spliced alignment algorithm.

Our clustering strategy is simple. First, candidate splicing pairs are detected by very accurate pairwise comparison with the spliced alignment algorithm. As an algorithm for the spliced alignment, we used

Mott's algorithm [107] to which we made a slight modification to meet our demand. As it requires excessive computation time to perform a brute-force all-pairs comparison of all the sequences of a standard cDNA library, we proposed two techniques to reduce the computation time. These techniques reduce the number of pairs of cDNA sequences to check without missing any important pairs and enables significant reduction of the total computation time without decreasing the accuracy. These algorithms are described in section 4.1. The combination of two techniques reduced the candidates to between 1/4,000 and 1/20,000 of the candidate pairs (depending on the parameters) according to the experiments in section 4.2 that use a large library of mouse full-length cDNA sequences, FANTOM 1.10 [90].

We also examined in section 4.2 the efficiency and the accuracy of our algorithms using the FANTOM cDNA library. According to the FANTOM paper [90], 21,076 cDNA sequences were manually clustered into 15,295 groups of unique genes. The clusters are called MGI clusters as they are confirmed in the MGI (Mouse Genome Informatics) database [28]. To assess the performance of our algorithm, we compared the MGI clusters against our clusters. The experiments revealed that 87-89% of the annotated MGI clusters were identical to the clusters obtained by our algorithm, although we did not use the MGI or any other database to facilitate clustering.

In the rest of this chapter, we explain about the algorithm in section 4.1 and demonstrate the experimental results in section 4.2.

4.1 Clustering Algorithm for cDNA libraries

In this section, we describe our strategy for clustering cDNA sequences in detail. First, we describe the outline of our algorithm in subsection 4.1.1. Our spliced alignment algorithm which our clustering algorithm uses is described in subsection 4.1.2. Our clustering scheme is then described in subsection 4.1.3. After that, two new techniques to reduce the total time for clustering is described in subsections 4.1.4 and 4.1.5. Finally, we add some discussions on these algorithms in subsection 4.1.6.

4.1.1 Algorithm Outline

Our basic clustering strategy is as follows (see also Figure 4.1). First, pairs of sequences that have possibilities to be candidate splicing pairs are selected by two independent filtering algorithms described in subsections 4.1.4 and 4.1.5. Then the selected pairs are checked by very accurate pairwise comparison based on a spliced alignment algorithm described in subsection 4.1.2. This spliced alignment algorithm can check whether or not a given pair of sequence is a splicing pair or not in $O(nm)$ time where n and m are the lengths of the two sequences. Clusters are then constructed by collecting the detected candidate splicing pairs based on the simple strategy described in subsection 4.1.3.

As stated above, we propose two independent filtering methods to reduce the candidates to be checked by the spliced alignment so as to reduce the total computation time. These two filters do not throw away any important pairs that will be determined as candidate splicing pairs by the spliced alignment algorithm. Therefore there is theoretically no decrease of accuracy caused by the use of these two filtering algorithms.

The first filtering method described in subsection 4.1.4 utilizes local similarities among the sequences in

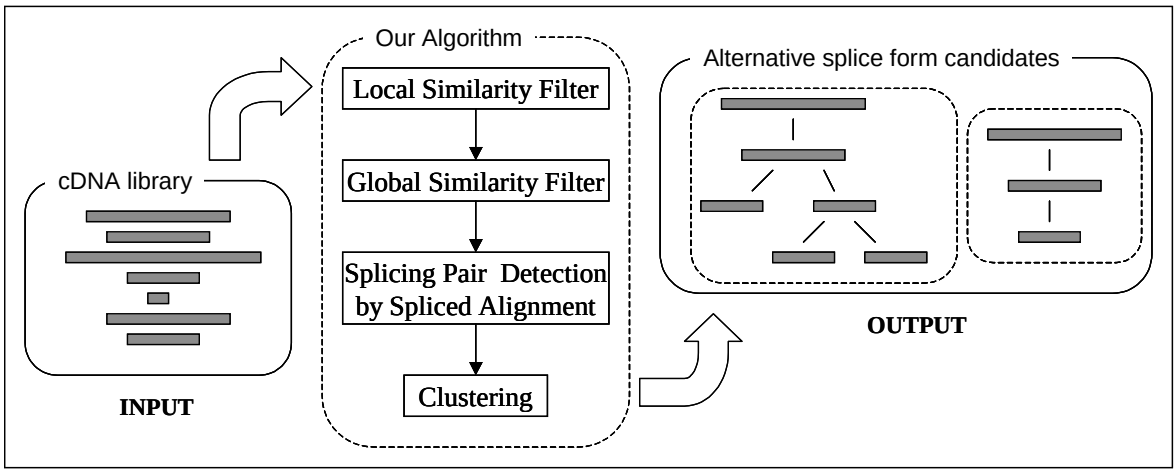


Figure 4.1: Algorithm outline.

an alternative splicing form. Note that methods similar to the first filtering algorithm are used in various applications, but it is the first case that this filtering strategy is applied to this problem. The second filtering method described in subsection 4.1.5 is a totally new filtering strategy for which we propose a new efficient longest common subsequence algorithm. It utilizes global similarities among the sequences in an alternative splicing form. Since the two filtering algorithms use different, independent information, the combination of both algorithms can dramatically reduce the number of pairs to be checked precisely by the spliced alignment algorithm and significantly reduce the total computation time, that is about 4,000 to 20,000-fold speedup. In our algorithm, the local similarity filter is used before the second global similarity filter because the global similarity filter takes much more time than the local similarity filter. Add to these, we also propose a much faster heuristic clustering algorithm by changing intentionally the parameters of these filtering algorithms.

4.1.2 Modified Spliced Alignment Algorithm

Our aim of the spliced alignment is to determine whether or not a pair of sequences is a splicing pair. Mott's algorithm is not suitable for this purpose though it can align sequences reasonably, because the alignment score obtained by his algorithm is strongly influenced by the number of splicing sites. It uses a rather large splice site penalty, *i.e.*, about 20 times to 40 times as large as the score for one mismatch, depending on the bases around the acceptor sites and the donor sites, to avoid an increase in the number of very short splice sites. This will cause difficulty in determining whether a pair is a splicing pair or not by its score, because a splicing pair with many splice sites will have a very bad score using his algorithm.

To avoid these drawbacks, we introduce a new problem formulation which we call *the spliced alignment problem with a minimum splice site length* (SAPMSSL). In the SAPMSSL, we set a minimum splice site length when we align the sequences, so that we do not have to worry about the splice site length even if we use a very small splice site penalty.

The SAPMSSL is formalized as follows. Let $P[1..n] \in \Sigma^n$ be the template sequence and $C[1..m] \in \Sigma^m$ be the spliced sequence to be aligned, and consider an alignment (P', C') of P and C. To define the score

of an alignment, we use the following functions and parameters. Note that we consider a region in C' of consecutive internal gaps with lengths larger than t described below is a splice site candidate.

- $match(a, b)$: The score function that outputs the score for matching bases $a \in \Sigma$ and $b \in \Sigma$. These scores are usually given in a table.
- gt_i, gt_e: The gap scores assigned to an internal and an external gap inserted into the template sequence, respectively. Note that the external gaps are the gaps located before the first base or after the last base of the sequence, and the internal gaps are all the other gaps.
- gs_i, gs_e: The gap scores assigned to the internal and external gaps in the spliced sequence.
- t: The minimum length of the splice site candidates.
- s: The score added to a splice site candidate that is independent from its length.
- $donor(A, i)$: The score function that outputs the score we add when a donor site begins at the position i of the sequence A.
- $acceptor(A, i)$: The score function that outputs the score we add when an acceptor site ends at the position i of the sequence A.

We assume that we can compute $donor(A, i)$ or $acceptor(A, i)$ in $O(1)$ time. We then define a score s_i for the i-th column of the alignment (P', C'). If neither $P'[i]$ nor $C'[i]$ is a gap, $s_i = match(P'[i], C'[i])$. If $P'[i]$ is an internal gap character ('-') or an external gap character, $s_i = gt_i$ or $s_i = gt_e$, respectively. If $C'[i]$ is the first gap character of a splice site candidate, the last gap character of a splice site candidate, another gap character of a splice site candidate, another internal gap character, or an external gap character, $s_i = s + donor(P, i)$, $s_i = acceptor(P, i)$, $s_i = 0$, $s_i = gs_i$, or $s_i = gs_e$, respectively. Finally, the SAPMSSL is defined as a problem of finding the alignment of P and C that have the minimum score $S(P, C) = \sum_{1 \leq i \leq l} s_i$. Note that we can translate this problem into the problem of finding the alignment with the maximum score by changing the signs of all the scores. The largest difference with the Mott's algorithm is the existence of t, the minimum length of a splice site candidate.

We describe an algorithm for solving the SAPMSSL in Figure 4.2. In the figure, $N_{i,j}$ denotes the optimal score for the spliced alignment of $P[1..i]$ and $C[1..j]$, which we compute by dynamic programming. The computation time of this algorithm is $O(nm)$, which is same as Mott's algorithm, and the required memory size is $O(n + m \cdot t)$. Note that it is easy to extend it to output the alignment result, but we do not show it here. We do not use an opening gap penalty [67], which is also easy to incorporate.

Using the above algorithm, we determine that the two sequences form a splicing pair if the obtained score is below some threshold h_0. Reasonably, the threshold should be proportional to the spliced sequence length. Let r_0 be the ratio of the threshold over the spliced sequence length m, *i.e.*, $r_0 = h_0/m$. Using information about the lower bound of similarities in the aligned regions (*i.e.*, regions except for splice sites and external gaps), we can estimate the appropriate r_0 value and consequently we can compute h_0. Note that two very similar sequences without any splice sites might also have a score below the threshold, and therefore will be determined as a splicing pair. There are many such examples in the experiment we will

for $(i = 0;\ i \leq n;\ i++)\ N_{i,0} = i \cdot gt_e$;
for $(j = 0;\ j \leq m;\ j++)\ N_{0,j} = j \cdot gs_e$;
for $(i = 1;\ i \leq n;\ i++)$ {
  for $(j = 1;\ j \leq m;\ j++)$ {
    if $(i < n)\ g_{template} = gt_i$; else $g_{template} = gt_e$;
    if $(j < m)\ g_{spliced} = gs_i$; else $g_{spliced} = gs_e$;
    $N_{i,j} = \min\{N_{i-1,j} + g_{spliced}, N_{i,j-1} + g_{template}, N_{i-1,j-1} + match(P[i], C[j]),$
      $S_{i-t,j} + acceptor(P, i)\}$; // Let $S_{l,x} = +\infty$ if $l < 0$
    $S_{i,j} = \min\{N_{i,j} + s + donor(P, i+1), S_{i-1,j}\}$;
  }
}
Output $N_{n,m}$ as the optimal alignment score.

Figure 4.2: Algorithm for the spliced alignment problem with a minimum splice site length.

describe in Section 4.2, but this does not matter because they can be considered to be variants of a sequence due to sequencing errors.

Given a threshold, we can reduce the computation time. We do not have to compute values of $N_{v',j}$ and $S_{v',j}$ when $v' > v$ if all of the $N_{v,j}$s and $S_{v,j}$s are not small enough to achieve a score within the threshold. Moreover, if the gap penalties for the template sequences have positive values and all of the other parameters (match scores, gap scores, splice site scores, donor site scores, and acceptor site scores) have non-negative values, we can reduce the time further. In such cases, we can see that the number of gaps in the spliced sequence is at most $w = h_0 / \min\{gt_i, gt_e\}$. Hence we do not have to compute the values of $N_{i,j}$ and $S_{i,j}$ of the algorithm in Figure 4.2 when $j - i > w$ or $i - j > n - m + w$. This will effectively reduce computation time if w is small and n is close to m.

Figure 4.3 shows an example of this spliced alignment. This is the optimal alignment of two mouse cDNA sequences that encode *Plp2* (proteolipid protein 2) and a potential isoform. The template *Plp2* sequence has the DDBJ accession number of AK012816 while the potentially spliced sequence has the accession number AK003522. In the alignment, '`-`' means a normal gap and '`=`' means a splice site. In this alignment example, we set the score for an exact match to 0, a mismatch to 1, a gap in the template sequence to 1, an internal gap in the spliced sequence to 1, an external gap in the spliced sequence to 0, the minimum splice site length to 40, and the splice site score to 2. Note that we gave a non-`GT` donor site and a non-`AG` acceptor site an additional penalty of 1 each while computing, but they do not appear in this alignment.

4.1.3 Simple Clustering Scheme

Now we have a tool to determine whether or not a given pair of sequences is a splicing pair. It will induce a directed graph in which a node represents a sequence and an edge represents the relationship of the template sequence and the spliced sequence. As the above spliced alignment algorithm can detect splicing pairs very accurately, we can construct alternative splice forms by just clustering into transitive closures as follows. If

Figure 4.3: An example of the spliced alignment.

we determine the sequences A and B form a candidate splicing pair, then we combine them into one group. If another sequence C forms a candidate splicing pair with any sequence in the group (*i.e.*, A or B in this case), add C to this group, and do the same for all the other sequences. This can be done in a linear time to the size of the graph.

This simple strategy is possible because we compute the alignment accurately taking the splicing mechanism into account. But it will take a large computation time to use this strategy if we do the all-pairs comparisons naively. In the next two subsections, we will discuss how to effectively reduce the total computation time.

4.1.4 Filtering based on Local Similarity

Except for splice sites, the members of a splicing pair of sequences are very similar to each other, and there are many common substrings between the two sequences. Thus we can reduce the number of candidate pairs which do not have any common substrings that are long enough before examining the candidates with the spliced alignment algorithm in section 4.1.2. Similar techniques are used in the previous work of [156], who searched for regions with common substrings of a fixed length before doing spliced alignments, though their setting of the length has no theoretical justification. A similar technique has also been used for approximate string matching problems [109, 124], but not for the spliced alignment problem.

Without loss of generality, we can assume that the score for exactly matching bases is 0. Consider the minimum value among the scores for mismatching bases, scores for splice sites, and gap scores for the template sequence. Let this be p_1. We assume that p_1 is a positive value, which is true in most cases. Let $h_1 = r_1 \cdot m$ be the threshold of the alignment score for examination where m is the spliced sequence length, and $q = \lfloor 1 + h_1/p_1 \rfloor$. Then divide the spliced sequence C into q or more substrings. Let them be $C_1, C_2, \ldots$. It is clear that one of them must exactly match some substring of the template sequence; otherwise the alignment score of the pair exceeds the threshold h_1. Therefore we can use this information for filtering candidates: We do not have to check a candidate pair of sequences if there are no such exactly matching substrings. As we mentioned in the last subsection, the threshold h_1 is proportional to the spliced sequence length m. Therefore, if we divide the sequence into q substrings with roughly equal lengths, the lengths of the substrings are about the same regardless of any pairs.

To find sequences with substrings that exactly match one of these divided substrings, some indexing structure is needed. Several known indexing structures are suitable for this purpose. The simplest structure is a hash table. A hash table that stores all the substrings of length l in a cDNA database of size N (*i.e.*, the sum of the lengths of all the sequences) can be built in $O(N)$ to $O(lN)$ time (depending on hashing algorithms), and the query time is $O(l)$. The suffix array [100] is another candidate indexing structure. In general, we can construct this structure in $O(N \log N)$ time. The query time is $O(l \log N)$ time. Note that we can reduce the query time bound to $O(l + \log N)$ time or $O(l)$ time if we can use an additional data structure. The advantage of this data structure over the hash table is that this structure can deal with various substring lengths, while a hash table can deal with only a fixed substring length. Note that there are also compressed versions of suffix arrays [53, 71, 123, 124], but they need more construction time and query time. The suffix tree [50, 73, 102, 155, 163] is another candidate for indexing. It requires $O(N)$ time to construct, and the query can be done in $O(l)$ time, but this data structure requires much larger memory space. We should use the most appropriate data structure for our purpose. In previous work, [156] used suffix arrays, [109] used suffix trees, and [124] used compressed suffix arrays. Considering the advantages and drawbacks of these indexing structures as stated above, we will use a simple hash table in the experiment section to reduce the computation time and conserve memory space at the same time. Note that the suffix array would also be a good choice.

The value r_1 should be the same as the value of r_0 in Section 4.1.2, because we will miss some of the actual splicing pairs if $r_1 < r_0$. However, we can reduce the number of splicing pair candidates by using a smaller r_1 value, and consequently reduce the total computation time. Thus a smaller r_1 value can be used as a setting for a heuristic clustering algorithm. We will examine the performance of this heuristic approach in the experiment section.

According to the experiments in the next section, this filtering technique is very effective, but the computation time is still too large for practical use. In the next subsection, we will describe another filtering algorithm to further reduce the number of candidates.

4.1.5 Filtering by Simplified Spliced Alignment

When we align a splicing pair, we can easily see that there are many fragments of the spliced sequence in the template sequence in the same order. We can use this fact to further filter out hopeless candidate pairs. As in the previous subsection, we can assume that the score for exactly matching bases is 0 without loss of generality. Let p_2 be the minimum value among the scores for mismatching bases and gap scores for the template sequence, m be the spliced sequence length and $h_2 = r_2 \cdot m$ be the threshold of the score to distinguish splicing pairs. We can easily see that at least $m - h_2/p_2$ bases of the spliced sequences appear in the template sequence in the same order; otherwise the alignment score of the pair will exceed the threshold. Let $k = h_2/p_2$. The sequence of bases that appears in a given sequence in the same order is called a subsequence of that given sequence. We can eliminate the candidate pairs before examining them with the time-consuming spliced alignment algorithm if the length of the longest common subsequence (LCS) of the pair is smaller than $m - k$. Note that we can align two sequences so that their LCSs are aligned. Figure 4.4 shows an alignment that corresponds to the LCS of two example sequences being `CGCGCATGAACAAACGCTGGAGCTCAGGATTCATCTCGGA` and `GCTGAGAAGAGGTTTCATCT`. Therefore computation of the LCS can be considered to be a simplified spliced alignment algorithm. Note also that this alignment is far from a biologically correct alignment, though we can use it for filtering as we described.

The LCS of a pair of sequences is known to be computable in $O(nm)$ time [73, 79], where n and m are the lengths of the two sequences. However, this time bound is the same as that of the spliced alignment algorithm, and it is not effective to use such slow algorithms for filtering. Therefore, we propose an algorithm for examining whether or not the LCS of the pair is longer than $m - k$, which runs in time $O(n + k \cdot m)$ and space $O(k + n + m)$. This algorithm is much faster than the ordinary LCS algorithms if k is small enough. Figure 4.5 shows the algorithm. In the figure, $P[1..n]$ and $C[1..m]$ are the sequences to examine. The function $next\_char\_position(P, i, c)$ returns the first position of the base c in P after the position i, which can be computed in a constant time if the alphabet size is constant, as is the case for DNA sequences. In this algorithm, we compute $positions[i]$ that stores the smallest d such that the LCS length of $P[1..d]$ and $C[1..j]$ is $j - i$, using a dynamic programming technique.

In contrast to the filtering algorithm based on local similarity, the algorithm in this subsection utilizes global similarity. The two filtering algorithms use totally different information, and the set of candidate pairs that can be filtered out by the algorithm in this subsection is different from those filtered out by the algorithm in the last subsection. Thus we expect that we can filter out candidate pairs very effectively if we use both of these filtering algorithms simultaneously. As the filtering method based on the local similarity is faster than the method based on simplified spliced alignment, we should use the local similarity algorithm

```
Template sequence candidate:    CGCGCATGAACAAACGCTGGAGCTCAGGATT-CATCTCGGA
Spliced sequence candidate:     -GC---TGA-GAA--G----AG----G--TTTCATCT----
Longest common subsequence:      GC   TGA  AA  G    AG    G  TT CATCT
```

Figure 4.4: An example of the longest common subsequence.

```
for (i = 0; i ≤ k; i++) positions[i] = 0;
k_min = 0;
for (j = 1; j ≤ m; j++) {
  k_max = min{k, j};
  for (i = k_max; i ≥ k_min; i--) {
    if (i == 0) {
      positions[i] = next_char_position(P, positions[i] + 1, C[j]);
    } else {
      positions[i] = min{next_char_position(P, positions[i] + 1, C[j]),
                         positions[i − 1]}
    }
    if (positions[i] > n) k_min = i + 1;
  }
  if (k_min > k) { return("LCS length is smaller than m − k."); }
}
return("LCS length is m − k_min, which is not smaller than m − k.");
```

Figure 4.5: An algorithm for longest common subsequence problem.

first and then use the other algorithm afterwards. In the next section, we will show how effectively we can reduce splicing pair candidates using these algorithms.

We should set the r_2 value to the same value as r_0, because otherwise we might miss some of the actual splicing pairs through filtering if $r_2 < r_0$. Similar to the heuristic scheme we described in the previous subsection, we can use a smaller r_2 value as a setting in a heuristic version of the clustering algorithm. We will also examine the performance of this heuristic scheme.

4.1.6 Discussions on Our Algorithm

The algorithm proposed in this section is not for clustering ESTs (expressed sequence tags). ESTs are substrings of the full-length cDNAs, and two ESTs that overlaps each other in the genomic template is not always detected as a splicing pairs by our algorithms. The easiest way to deal with ESTs is to add the genomic templates to the set of ESTs as previous methods do in previous work [30, 34]. Then we can easily deal with the ESTs.

In the previous work of cDNA clustering, tools for masking low-complexity regions like RepeatMasker [148] is often used before clustering. It is because general sequence comparison tools like BLAST that they use are not good at distinguishing splicing pairs with repetitive similar patterns, and consequently they cannot detect splicing pairs accurately. We don't apply RepeatMasker to filter out repeats before applying our algorithm because it can detect splicing pairs accurately. However we use RepeatMasker to visualize repeat elements by small letters in the splicing pairs to facilitate better biological interpretation.

4.2 Computational Experiments

In this section, we demonstrate the performance of our clustering algorithms through experiments using sequences of the mouse cDNA library FANTOM1.10 [90], which can be obtained from the RIKEN ftp site (ftp://fantom.gsc.riken.go.jp/fantom/1.10/fantom1.10.seq.gz). Note that an updated version FANTOM 1.20 has been released recently at the ftp site (ftp://fantom.gsc.riken.go.jp/fantom/current). This library contains $21,076$ sequences. Thus there are $21,076 \times 21,075 = 444,176,700$ pairs to check, because we must distinguish the template sequence with the spliced sequence when we compute the alignment score. All of the experiments were done on a single IBM RS64III processor with a clock speed of 450MHz.

In our experiments we set the score for an exact match to 0, a mismatch to 1, a gap in the template sequence to 1, an internal gap in the spliced sequence to 1, and an external gap in the spliced sequence to 0. We let the minimum splice site length be 40. We set the splice site score to 2 and we gave non-`GT` donor sites and non-`AG` acceptor sites an additional penalty of 1 each. These are the same settings used in Section 4.1.2 for computing the spliced alignment example.

4.2.1 Performance of the Filtering Algorithms

Table 4.1 shows the performance of our algorithm with the settings above. In the table, the *ratio* column shows the ratio of the threshold over the spliced sequence length, *i.e.*, $r_0 = r_1 = r_2 = ratio$. (See Section 4.1 for the definition of r_i's.) We did experiments based on 5 different ratios. The average accuracy of sequences in FANTOM 1.10 is 99.1% according to the report [90]. Thus we estimate that the average ratio of the spliced alignment score over the spliced sequence length for an actual splicing pair is around 0.01. Therefore we believe that reasonable *ratio* values are around 0.02 or 0.03. The N column indicates the number of candidate splicing pairs detected with the setting of the corresponding *ratio*. The N_s column shows the number of candidate splicing pairs that have apparent splice sites, *i.e.*, internal gaps whose lengths are larger than 40. Note that here we do not count shorter gaps as splice sites, though such splice sites exist. The T_{total} column shows the total computation time in seconds, including the filtering stage, the alignment stage, and the clustering stage. The T_1, T_2 and T_3 columns show the computation time in seconds for the filtering algorithm based on the local similarity, that for the filtering algorithm based on the simplified spliced alignment, and that for the spliced alignment algorithm, respectively. The N_1 column shows the number of remaining candidates to check after doing the first filter based on the local similarity, and the N_2 column shows the number of those that passed through both of the filters.

According to the table, the time for the filtering stages can be ignored, and the total computation time is essentially determined by the number of splicing candidates that passed through both of the filters. We can also see that, if we let *ratio* be larger, the number of candidate splicing pairs in the output will increase and it will be difficult to filter out candidates, which makes the computation time longer. The computation time of the first filtering stage will decrease as the *ratio* increases, because the window size of the hash will become smaller. The number of candidate pairs to check using the spliced alignment algorithm is only about 1/4,000 to 1/20,000 (depending on the setting of the *ratio*s) of all the $444,176,700$ candidate pairs, which means a 4,000 to 20,000-fold speedup against a very naive brute-force algorithm.

Table 4.1: Performance on FANTOM 1.10 with various thresholds.

ratio	N	N_s	T_{total}	T_1	T_2	T_3	N_1	N_2
$r = 0.01$	12,676	205	46177.65	294.83	65.07	45137.20	40,135	23,514
$r = 0.02$	15,713	432	54530.04	254.55	257.60	53330.12	68,335	27,521
$r = 0.03$	17,237	574	73771.73	233.84	994.07	71840.75	165,727	35,444
$r = 0.04$	18,300	684	89184.62	227.53	2061.80	86211.47	244,814	41,834
$r = 0.05$	19,123	814	285925.44	222.50	15897.57	269146.94	1,451,698	119,993

4.2.2 Comparison with MGI Clusters

Table 4.2 shows the numbers of clusters of various sizes in the output of our algorithm and in the set of the MGI clusters. The *ratio* column shows the ratio of the threshold over the spliced sequence length, as in Table 4.1, except for the 'MGI' row that shows the numbers of the MGI clusters. Notice that about 2/3 of the cDNA sequences in the library do not form any candidate splicing pairs with other sequences. The 'Comparison (total)' columns show the comparisons of our clusters with the MGI clusters including singleton clusters. The 'Comparison (size$\geq$ 2)' columns show the comparisons of our clusters to the MGI clusters without including singleton clusters. The ID, CB, SP, and OV columns indicate the numbers of our clusters that are identical to the MGI clusters, those that are combined with others to form the MGI clusters (over-split), those that are split in the MGI cluster set (over-clustered), and those that overlap with some of the MGI clusters, respectively. According to these numbers, most of our clusters are identical to the MGI clusters. The identical clusters are about 87.5% (13,384/15,295 when $ratio = 0.01$) to 89% (13,625/15,295 when $ratio = 0.05$) of the MGI clusters (that are 77% to 86% of our output). Within these, 69% ($ratio = 0.01$) to 75% ($ratio = 0.05$) of the MGI clusters with more than 1 sequence are identical to our results according to the table. Note that many of non-identical clusters are partially identical to the MGI clusters. When $ratio = 0.05$, about 12% of our clusters are over-split clusters compared to the MGI clusters. There are only a small number of over-clustered clusters and overlapping clusters in our results. This is because we do not consult either the template mouse genome sequences or any known genes in public databases, and alternative splicing sequences without their common template sequence in the cDNA library cannot be clustered with our algorithm. Even if we do not have reference information (*e.g.*, genomic sequences or pre-existing clusters), our clustering algorithm mis-clusters only 11 out of 100 human-annotated clusters. This performance can significantly enhance the expert annotation or sequence analysis of large sequence sets.

Let us look in detail at two of the MGI clusters that were annotated by human experts. There is a cluster of 4 cDNA sequences that were annotated as *Plp2* (proteolipid protein 2) [103]. The spliced alignment of two of them appears in Figure 4.3. Three of the sequences have the accession numbers AK003522, AK011282, and AK012816. The remaining 1 sequence (2810425P20) has not been submitted to DDBJ yet but was designated by MGI as *Plp2* marker. We succeeded in clustering these sequences correctly, regardless of the *ratio* settings in Table 4.1. Note that the common template of this cluster seems to be the *Plp2* sequence AK012816, according to our results. Another cluster is known to be related to a gene called *Tex9* (testis expressed gene 9), which contains 3 cDNA sequences [38] with the DDBJ accession numbers AK018568,

Table 4.2: Clustering results for FANTOM 1.10 with various thresholds.

ratio	Cluster size						Comparison (total)				Comparison (size$\geq$ 2)			
	1	2	3	4	5	≥ 6	ID	CB	SP	OV	ID	CB	SP	OV
0.01	15,213	1,371	441	158	62	91	13,384	3,851	80	21	1,472	550	80	21
0.02	14,046	1,580	545	209	88	102	13,636	2,728	182	24	1,881	437	182	24
0.03	13,569	1,670	593	225	100	106	13,680	2,311	249	23	2,038	384	249	23
0.04	13,246	1,743	627	234	105	108	13,687	2,050	301	25	2,131	360	301	25
0.05	13,028	1,781	639	243	114	111	13,625	1,903	360	28	2,161	339	360	28
MGI	12,064	1,986	702	287	138	118	-				-			

Table 4.3: Heuristic clustering results.

Algorithm	N	N_s	T_{total}	T_1	T_2	T_3	N_1	N_2
$r_1 = 0.01$	19,118	814	53959.52	299.90	443.73	52500.42	40,135	27,875
$r_1 = 0.02$	19,121	814	60617.79	253.25	760.48	58929.68	68,335	30,485
$r_1 = 0.03$	19,123	814	81240.73	231.97	1823.75	78510.80	165,727	38,450
$r_1 = 0.04$	19,123	814	94833.48	231.22	2695.67	91224.10	244,814	43,889
$r_2 = 0.01$	14,175	550	150229.89	222.50	2573.08	146777.86	1,451,698	72,971
$r_2 = 0.02$	16,552	682	184948.25	222.50	5476.10	178609.89	1,451,698	85,990
$r_2 = 0.03$	17,803	757	218017.16	222.50	8681.78	208474.89	1,451,698	97,483
$r_2 = 0.04$	18,608	800	251564.05	222.50	12166.08	238538.28	1,451,698	108,670
Exact	19,123	814	285925.44	222.50	15897.57	269146.94	1,451,698	119,993

AK012189, and AK008505. Our algorithm clustered these 3 sequences into 2 clusters regardless of the *ratio* settings. One cluster contains only AK018568 while the other contains AK012189 and AK008505. The reason why our result is different from the MGI cluster of *Tex9* is that the FANTOM library does not contain any common template sequence of these 3 sequences: Our algorithm would cluster the 3 sequences into one cluster if a common template sequence was present.

4.2.3 Performance of the Heuristic Scheme

Next, we clustered the sequences using the heuristic version of our algorithm. We varied the r_1 and r_2 values used in the filtering stages while we did not change the r_0 value used in the final spliced alignment dynamic programming algorithm. Table 4.3 shows the results. In all the experiments in this table, we fixed the r_0 value at 0.05. An $r_1 = x$ row indicates that we used the threshold that corresponds to $r_1 = x$ as the threshold for the filtering algorithm based on local similarity. In these experiments, we fixed r_2 at 0.05. Similarly, an $r_2 = x$ row indicates that we used the threshold that corresponds to $r_2 = x$ as the threshold for filtering algorithm based on simplified spliced alignment, while r_1 was fixed at 0.05 at the first filtering stage. The 'Exact' row is the same experiment as the *ratio* = 0.05 row in Table 4.1.

The results listed in the above table demonstrate that varying the thresholds of the local similarity filter

is very effective. The accuracy does not decrease significantly, even if we set $r_1 = 0.01$. Only five splicing pair candidates were missed with this setting. The algorithm with the setting $r_1 = 0.01$ is about 5 times faster than the exact algorithm. Thus, we can effectively reduce the computation time by changing the threshold of the local similarity filter with only a very small loss of accuracy. On the other hand, varying the threshold of the simplified spliced alignment filter negatively affects the accuracy.

4.3 Summary

We have described a new efficient and accurate method to cluster sequences of full-length cDNA libraries based on an accurate spliced alignment algorithm. To decrease the computational time of the algorithm, we proposed two methods to filter out candidates that do not meet the threshold settings before applying the time-consuming spliced alignment algorithm. With these techniques we achieved 4,000 to 20,000-fold speed-ups without a loss of accuracy compared to a naive brute-force approach. Most of our clusters turned out to be identical to the annotated clusters. We also developed an effective heuristic algorithm that is several times faster than the exact algorithm. The accuracy of the heuristic algorithm is very close to the exact algorithm.

Alternative splice form detection, mutation analyses or DNA motif analyses from huge cDNA-derived data sets are dependent on the clustering methods. Therefore, both fast and accurate algorithms are required to support efficiently biological evaluation and interpretation. Future tasks include the parallelization of the algorithm to enhance the computation speed and the construction a secondary databases of splice site motifs derived from the FANTOM1.10 sequences to improve the specificity of potential splice candidate detection.

Chapter 5

Dictionary-driven Prokaryotic Gene Finding

As a testimony to the accelerated pace of genome sequencing projects, almost eighty complete genomes have been deposited in the public databases to date, whereas many more genomes are currently at various stages of sequencing. Consequently, the automated identification of the protein coding regions in a newly sequenced genome is attracting increasing attention.

Accurate gene prediction is of relevance to many biological applications. For example, the predicted coding regions can be used to generate probes for a DNA microarray; they can form the basis for knockout experiments; the candidate proteins corresponding to these predicted genes might be used as new drug targets, etc. In this chapter, we focus on the prokaryotic gene identification problem. Gene identification is also known as 'gene discovery', 'gene recognition' or 'gene finding' – the latter is the term we use in this discussion.

With the exception of a handful of reported instances in archaeal organisms, splicing does not generally occur in prokaryotes. Thus, the problem of gene identification in these organisms is generally considered to be simpler than its eukaryotic counterpart. The schemes which have been proposed over the years have permitted great advances in the *in silico* prediction of genes in prokaryotic genomes but, arguably, have shortcomings. As such, the demand for increasingly accurate prediction schemes continues.

Over the years, a large number of methods have been proposed that address the gene finding problem. These methods can be largely divided into two categories. The methods in the first category make use of the statistics of DNA sequences to determine the location of genes. In fact, it was observed very early that the statistical properties of nucleotide usage differs inside DNA regions which code for genes and outside those regions: the concept of the CpG island [27] is a demonstration of such a difference in statistical behavior. Among the gene identification methods which make use of this observation those which are based on Markov models are the most popular to date [31, 46, 98, 125].

The second category comprises methods that are based on similarity search in large databases of genomic information [13, 14, 62, 63, 66, 122]. These methods carry out database searches in an effort to determine DNA regions (resp. amino acid sequences) that share similarities with the DNA regions (resp. amino acid translations) of open reading frames from the genome under consideration. For details on such techniques, the reader is referred to one of the many review papers on the topic; several of these papers also address the gene identification problem in eukaryotic organisms [33, 35, 40, 41, 54, 55].

Despite the notable success of the methods that have been developed over the years, each of these two basic strategies has its own shortcomings. Statistical methods such as those based on Markov models can identify coding regions whose statistical behavior is similar to that of the used training set. If no appropriate training set is available, one resorts to using sets derived from database searches, or simply assumes that very long open reading frames do code for genes. The statistics of coding regions often differ from organism to organism and, ideally, if one wishes to achieve high prediction ratios using such approaches then one ought to employ models with organism-dependent parameters. Essentially, a different Markov model must be built for each targeted genome. Moreover, short genes (*e.g.* fewer than 60-80 a.a.) cannot be predicted reliably using statistical methods. Finally, genes that are statistically distinct from other genes of the same organism, *e.g.* genes that are the result of horizontal transfer [91, 111] typically represent challenges for methods based on statistical schemes.

Unlike statistical methods, similarity-based approaches are more effective in finding short genes or genes that are statistically distinct from the majority of the genes in the organism being studied. The implicit assumption here is that similar genes or similar proteins are already present in the databases that are searched. Clearly, there is no dependence on training sets since no such sets are needed. Problems can arise if the shared similarity between a candidate gene and its database counterpart is very low and not detectable. In general, similarity-based methods have an improved ability to determine the correct location of genes over statistical methods, a very desirable property for gene finding tools.

Given that the best characteristics of these two categories complement one another, genome sequencing projects typically use representative methods from both categories to generate results [57].

In what follows, we present a new method which borrows the best attributes from similarity searches while at the same time relying on implicit sequence statistics as in the case of Markov models. Our method builds on a new paradigm that describes amino acid sequences with the help of patterns that are present in these sequences. The patterns are derived by processing very large public databases of amino acid sequences with the help of an unsupervised discovery algorithm. It is worth noting that our method does not make any use of additional evidence, *e.g.* ribosome-binding sites, in order to decide the presence, absence and location of genes. Clearly, incorporating such information provides additional constraints that can further increase the quality and accuracy of the results generated by a gene finding algorithm. Nonetheless, in this first installment of our approach, we have decided against incorporating such modules. We made this decision because of our desire to present a clear assessment of our method's potential as a generic, alternative scheme to gene finding. We plan to incorporate such additional modules in follow-up work.

In section 5.1 we describe our approach in detail. Section 5.2 contains details on the experimental setup as well as a presentation and analysis of the results obtained from applying BDGF, our algorithm's implementation, to 17 archaeal and bacterial genomes.

5.1 Methods and Algorithms

In this section, we present methodological details and give information on the algorithms which we have employed.

5.1.1 Notation and Definitions

Let Σ denote the alphabet of all 20 amino acids. When processing an input dataset containing a collection of strings from Σ^+ with the Teiresias algorithm [117, 118], we can succinctly capture the patterns that can be discovered with the regular expression $\Lambda(\Lambda \cup \{\text{`.'}\})^*\Lambda$ where $\Lambda = (\Sigma \cup [\Sigma\Sigma^*\Sigma])$, and '.' is a "don't care" character which stands for any character in Σ. In other words, the generated patterns can either be a single alphabet symbol, or strings that begin and end with a symbol or a bracket with two or more characters, and contain an arbitrary combination of zero or more residues, brackets with at least two alphabet characters, and don't care characters. A bracket denotes a "one of" choice; *i.e.* `[CPM]` denotes exactly one of `C`, `P` or `M`. Also, a bracket can have a minimum of 2 (two) alphabet characters but obviously not more than $|\Sigma| - 1$.

A pattern t is called an $< L, W >$ pattern (with $L \leq W$) if every substring of t of length W which begins and ends with a literal comprises L or more positions that are occupied by literals. The smallest length of an $< L, W >$ pattern is obviously equal to L whereas its maximum length is unbounded. Any given choice for the parameters L and W has direct bearing on the degree of remaining similarity among the instances of the sequence fragments that the pattern captures: the smaller the value of the ratio L/W, the lower the degree of local similarity. Also associated with each pattern t is its support which is equal to the number of t's instances in the processed input database. Finally, K denotes the minimum required support and represents the minimum number of instances that a pattern t must have before it can be reported.

5.1.2 Bio-Dictionary

The concept of the Bio-Dictionary is introduced in detail in previous work [119, 120, 121]. The Bio-Dictionary is a collection of patterns that we refer to as seqlets (for 'small sequences') and which completely describes and accounts for the sequence space of natural proteins at the amino-acid level. The seqlets are derived by processing a large public database of proteins and fragments, using the Teiresias algorithm [117, 118] and for an appropriate choice of L, W and K. In [119], the computation of the Bio-Dictionary from the GenPept release from February 10, 1999 using $L = 6$, $W = 15$ and $K = 2$ is described. The processed input contained ~387,000 sequences amounting to a grand total of ~120M amino acids. The computation resulted in a Bio-Dictionary that at the time comprised ~26M seqlets and accounted for (i.e. covered) 98.12% of the amino acid positions in the processed input. The reader is referred to that publication for details regarding the computation, example seqlets with discussion, and an extensive description of possible applications. Indeed, the availability of such a complete collection of seqlets permits one to effectively and successfully tackle a number of tasks that among other include similarity searching [58], functional annotation [121], phylogenetic domain analysis [120], gene identification, and other. Below, we describe gene identification in detail.

5.1.3 The Key Idea Behind Dictionary-driven Gene Finding

As mentioned already, the Bio-Dictionary concept seeks to substitute a given database of proteins and fragments such as GenPept or SwissProt/TrEMBL [15] by an equivalent collection of regular expressions (=seqlets) each of which represents combinations of amino acids that appear two or more times in the processed input. To the extent that the input sequences in such a public database correspond to a representative

sampling of the sequence space of natural proteins, the seqlets of the Bio-Dictionary represent an exhaustive collection of intra- and inter-family signals that are discovered in an unsupervised and exhaustive manner. Our computation of intra- and inter-family signals becomes possible due to the fact that during processing we consider all publicly available sequences without any of the filtering that one encounters in databases such as PROSITE [80], Pfam [22] or PRINTS [11], and which is based on the sequences' known functional behavior. The properties of the Teiresias algorithm guarantee that all $< L, W >$ patterns will be discovered and that they have the maximum possible extent and specificity.

Two requirements need to be fulfilled for the Bio-Dictionary approach to be successful in tackling the kinds of problems in which we are interested. First and foremost, the input to be processed should be a large and diverse collection of proteins and fragments (see [120] for details), a condition that can be satisfied given the large number of completed and ongoing genome sequencing projects which contribute to the public databases. Second, the pattern discovery process should be able to generate patterns that are specific enough, not accidental, and account for as much of the processed input as possible. As we have demonstrated in [119] this is indeed possible, thus the second requirement is satisfied as well.

Given this description, our strategy for gene finding should now be evident. First, we compute all possible ORFs in each of the three reading frames and for both the forward and reverse strands of the given DNA sequence. Clearly, the number of true coding regions will be a proper subset of this collection of ORFs. Then, for each ORF we generate its amino acid translation: if the ORF under consideration is indeed a coding one, then we should be able to locate instances of many of the Bio-Dictionary's seqlets across the span of the ORF's translation, and vice versa. If the number of seqlets that we can locate exceeds a predetermined threshold, we report the ORF as a putative gene. We discuss the details of thresholding below; as a rule of thumb, the higher the number of Bio-Dictionary seqlets that can be found in a given ORF, the more likely it is that the ORF is coding for a gene.

5.1.4 Fast Seqlet Search Scheme for Bio-Dictionary

The Bio-Dictionary we use in our experiments contains approximately 30 million seqlets (see the following section for more details). All of these seqlets need to be checked against a large number of amino acid translations from all ORFs on both strands of a given complete genome. It is thus important that this operation be carried out as efficiently as possible so as to reduce the overall computational requirements of our method. There exist many efficient linear-time algorithms for searching exact strings [73], but since seqlets include don't care characters these algorithms are not applicable here. We can address the problem of determining whether a seqlet matches a given amino acid sequence in one of two generic ways: we can either preprocess the amino acid sequence, or we can preprocess the Bio-Dictionary seqlets with which we will be searching.

Most of the work that has appeared in the literature revolved around the preprocessing of the sequence which is assumed to be fixed [73]. Moreover, the number of patterns whose instances were sought in the sequence was substantially smaller than the collection of patterns we are interested in. The suffix tree [50, 73, 102, 155, 163], which is the compacted trie of all the suffixes of a given string, is a very efficient and useful data structure for handling this task. When using a suffix tree, the query time is not linear in the

size of the problem, but the approach affords rather efficient searches. The disadvantage of the suffix tree is its large size compared to the size of the string. Note that if the number of characters in Σ is very small, there exist other methods like fast Fourier transform [52, 56] and the shift-and method [24, 73], but when $|\Sigma| = 20$ as in our case, these methods are not appropriate.

Note that in our work, the set of seqlets is fixed, known in advance, and very large. On the other hand, the amino acid sequences that we need to examine are numerous and not known in advance. Moreover, the average length of such a query sequence is just several hundred amino acids. It thus appears more effective to preprocess the contents of the Bio-Dictionary. One approach would be to build hash indices out of seqlets so as to reduce the number of seqlets that need to be examined at each position of the amino acid sequence; however, it is not immediately clear how to build such indices for patterns that have variable numbers of don't care characters as is the case of the seqlets.

Similarly to what is done in traditional dictionary searches without don't care characters, we can build the index with the help of the first few characters of a seqlet. For example, the seqlet `G..G.GK[ST]TL` could be indexed through `G..G` if we consider indices that are built using the first 4 characters of the seqlet. If one of the positions used to build the index is occupied by a bracket expression like `[ST]` it is much simpler to replace the contents of that position by a don't care character. This method is rather effective and can in fact be improved further. In the case of `G..G.GK[ST]TL`, it would seem more appropriate to use one of `GK.T` or `K.TL` as the seqlet's 4-character-induced index instead of the first characters `G..G`. Since `GK.T` and `K.TL` contain fewer don't care characters, they should generally appear fewer times than `G..G` in an arbitrary protein.

Keeping these observations in mind, we next propose a novel effective scheme for generating indices out of seqlets. Let us define an (l, w) subpattern of a seqlet t as follows: first, we replace all brackets in the seqlet by don't care characters and let t' denote the modified seqlet. The (l, w) subpattern is a minimal substring of t' such that its length is less than w and it contains l characters that are not don't care characters. Notice that we cannot always find such (l, w) subpatterns; if one such subpattern exists, we select it and form the index for t. If such a subpattern does not exist we find the largest value l' such that an (l', w) subpattern exists in t' and use this subpattern to form the seqlet's index: for example, `GK.T` is a $(3, 4)$ subpattern of `G..G.GK[ST]TL`.

If a seqlet t is indexed by an (l, w) subpattern, we have to check this seqlet approximately $n/|\Sigma|^l$ times when we consider a random protein of length n that has uniform amino acid bias. We can of course reduce searching time by setting l to a large value. But, at each position, we must examine patterns that are indexed by any of $\binom{n}{k}$ possible (l', w) subpatterns for all $l' \leq l$. The total number of subpatterns to be checked at any one position is thus $O(2^w)$; if we set w and l to large values, the search will be slow. In section 5.2.2 we will describe how to choose appropriate values for l and w. Note that if there exist seqlets without any fixed character (*e.g.* `[LK]..[ST][GKT]..[AERST][ELK]..[AVL]`) we must check them at each position in addition to the seqlets hashed by subpatterns as above.

If we know the frequency of appearance of individual amino acids in query sequences, we can use it to estimate the probability of appearance for each subpattern. In our case, we can use it to further improve the performance. For example, in the case of `G..G.GK[ST]TL`, if we know that the amino acid `G` appears

less frequently than the amino acid L, we can assume that GK.T is also rarer than K.TL, and thus we should hash the seqlet with the help of GK.T. There can be cases where some (l', w) subpattern is estimated to be less frequent than an (l, w) subpattern although $l' < l$. In such cases, we should use the former, more rare subpattern. We can easily search for such a subpattern in time that is linear to the size of the seqlet. In the experiments below, and for given values of l and w, we use as a hash index the least frequent (l', w) subpattern, with $l' \leq l$, as can be estimated from a given known amino acid bias.

5.1.5 Incorporating A Weighting Scheme

The basic strategy for gene identification which we just described is straightforward. It is easy to see that in addition to the number of seqlets that can be located within the translation of an ORF, the very composition of these seqlets can have an impact when deciding whether the ORF codes for a gene. In general, any two Bio-Dictionary seqlets that match an amino acid translation affect this final decision differently. One can think of each seqlet being associated with a specific score: by summing up the scores of the individual seqlets that match an ORF, we can compute a quality measure that will allow us to determine whether to report the ORF as a putative gene or to discard it. Next, we examine how to appropriately weigh each of the seqlets.

Let $T = \{t_1, t_2, \ldots, t_n\}$ be the complete collection of seqlets in a given Bio-Dictionary. Let us consider the amino acid translation s of an ORF from a given DNA sequence and let l be the length of s. We say that a seqlet matches at position j of the amino acid sequence s if an instance of the seqlet can be found beginning at the j-th location of s. For example, G..G.GK[ST]TL matches the sequence MTHVLIKGAGGSGKSTLAFW beginning at position 8 of the sequence. Let T_{s_j} denote the set of those seqlets that match beginning at position j of s, and let T'_{s_j} be the set $T \backslash T_{s_j}$. Also let $T_s = \{t_{v_1}, t_{v_2}, \ldots, t_{v_m}\}$ denote the concatenated list of T'_{s_j}'s for all j $(1 \leq j \leq l)$. Similarly, let T'_s denote the concatenated list of T'_{s_j}'s for all j $(1 \leq j \leq l)$. Note that T_{s_j}'s for different j's can contain the same seqlet, thus T_s is in general a multiset. Similarly, T'_s can also be a multiset.

Let p_i be the probability that seqlet t_i matches a database protein or fragment at a fixed location, and q_i be the probability that t_i matches the amino acid translation of a non-coding ORF at a fixed location. If all the seqlets in the Bio-Dictionary are assumed to be statistically independent then the probability that a given ORF corresponds to an actual gene will be equal to $rP_s/(rP_s + (1-r)Q_s)$ where $P_s = \Pi_{j=1}^{m} p_{v_i} \cdot \Pi_{i=1}^{m'}(1 - p_{u_i})$, $Q_s = \Pi_{i=1}^{m} q_{v_i} \cdot \Pi_{i=1}^{m'}(1 - q_{u_i})$, and r is the ratio of ORFs within the given ORF set that correspond to actual genes.

Let us examine this independence assumption a little further by considering two specific seqlets that contain don't care characters and have several instances in the database from which they were derived. It is easy to see that because of the don't care characters the seqlets can be overlapping and matching the sequence under consideration (the 'query') even though they are derived from two distinct groups of unrelated sequences – in such a case, the seqlets cannot really be considered dependent. Of course, it can also happen that the seqlets are overlapping and matching the query and they are also derived from related groups of sequences in the processed database – in this case, the two seqlets would be dependent. Frequently though, two seqlets would be dependent only as a result of a small subset of sequences that is shared by the two groups of input sequences that gave rise to the two seqlets in the first place. It is conceivable that one could keep track of such situations for all groups of seqlets whose instances overlap within some sequences of

the database. Something like this would of course require exceptionally demanding bookkeeping (in terms of required space and time) and would make the application of our approach prohibitive for problems of practical size. It should be clear that the combinations of seqlets that ought to be considered dependent due to overlaps of some of their database instances are substantially fewer than the combinations of seqlets that are independent. Moreover, a given amino acid position of the query is typically 'covered' by 5-10 seqlets, a small number compared to the total number of seqlets that will match somewhere within it. Thus, any seqlet dependence that manifests itself in the outlined gene finding process will involve only a small number of seqlets each time and only because of a small fraction of the total number of their instances in the database from which they were originally derived. Consequently, the assumption of independence is a reasonable simplifying approximation that also allows for speedy computations and, as evidenced by the results we present in the next section, does not have any noticeable adverse impact on our results.

We can use the ratio $R_s = P_s/Q_s$ to compare the relative likelihood that two candidate ORFs correspond to genes. Note that the probability will be larger than 0.5 if $R_s > 1$ and $r = 0.5$. Let $N = \Pi_{i=1}^{n}(1 - p_i)$ and $N' = \Pi_{i=1}^{n}(1 - q_i)$. N denotes the probability that no seqlet matches at a fixed position of an actual protein, and N' the probability that no seqlet matches at a fixed position of the amino acid translation derived from a non-coding ORF. Recall that the Bio-Dictionary is derived from a database of proteins and fragments, thus N is smaller than N' in most cases. Considering that the number of seqlets that match at a given position is much smaller than the number of seqlets not matching at the same position, and that the p_i's and q_i's are very small numbers, we can approximate the terms $\Pi_{i=1}^{m'}(1 - p_{u_i})$ and $\Pi_{i=1}^{m'}(1 - q_{u_i})$ by N^l and N'^l respectively. Thus we can use $R'_s = (P'_s/Q'_s) \cdot (N/N')^l$ instead of R_s, where $P'_s = \Pi_{i=1}^{m} p_{v_i}$ and $Q'_s = \Pi_{i=1}^{m} q_{v_i}$. Note that, if $N \approx N'$, we can use $R''_s = P'_s/Q'_s$ instead of R'_s.

By definition, ORFs do not contain any stop codons internally, and consequently long ORF-like stretches are not likely to be random. Let L denote the probability that a stop codon is observed at a fixed position of a random DNA sequence. Since there exist 3 stop codons among the 64 possible codons and assuming that all 4 possible nucleotides appear with equal probability in the random sequence, then L is equal to 3/64. With this in mind, we can use $R'''_s = R'/(1 - L)^l = (P'_s/Q'_s) \cdot M^l$ where $M = N/N' \cdot (1 - L)$ instead of R'_s. If $M \approx 1$, we can use $R''_s = P'_s/Q'_s$ instead of R'''_s.

Let $w_i = \log p_i - \log q_i$ be the weight associated with seqlet t_i. Let us also consider the sum of weights of the seqlets matching anywhere in the translation s of an ORF as the measure W_s that is characteristic of the coding quality of the ORF under consideration. It is easy to see that we can write the following equation for the coding quality measure of an ORF: $W'_s = W_s + l \cdot \log M = \log R'''_s$. If M cannot be ignored, we can instead use the following expression $W'_s = W_s + l \cdot \log M = \log R'''_s$ to define the coding quality of an ORF. In actuality, the value of $l \log M$ is far smaller than the value of W_s, and we can safely ignore the term during the actual computations.

At times, we are faced with a situation where we have multiple start codons matching the same stop codon and must decide which start/stop pair to report. Our solution amounts to picking the start codon which will result in the highest value for the coding quality measure. Due to the fact that seqlets can also have negative associated weights, and even if we ignore the $\log M$ term, it should be evident that selecting the start codon in such a way will not necessarily result in the reporting of the longest ORF as coding.

On a related note, ATG is the most frequently used start codon but it is not the only one. Consequently, it is inappropriate to treat the different start codons in a uniform manner. Let $\{c_1, c_2, \ldots, c_k\}$ denote the set of possible start codons. Let f_i be the probability that c_i is the start codon of a randomly chosen coding region, f_i' be the probability that c_i is observed in non-coding regions, and g_i be $\log f_i - \log f_i'$. We can then use $W_s + g_i$ instead of W_s as the measure of coding quality for the amino acid translation s of an ORF that is initiated by the start codon c_i.

In order to compute the coding quality measure, we need the values for p_i's and q_i's. The most natural way to obtain these values is to compute them with the help of actual genes and non-coding ORFs. We can calculate the actual seqlet occurrences in the regions annotated as coding in a given training set and derive the needed p_i values; we can then compute each seqlet's occurrences in ORFs that are not designated as coding in a training set and derive the q_i values. The values for the f_i's and f_i''s can be computed in a similar manner.

But how can these values be obtained in the absence of a training set? For the p_i's we can use the probabilities computed with the help of the protein database from which the Bio-Dictionary is derived. Alternatively, we can compute them using very long ORFs instead of actual coding regions. For the q_i's we can use non-ORF regions, or we can estimate the probability of random occurrence based on an appropriately chosen amino acid bias.

Once we have attached a corresponding coding-quality measure to each ORF we can decide which ORFs correspond to putative genes by appropriately setting a threshold value. The higher the value of the measure we associate with a given ORF the more likely it is that the ORF is a coding one.

5.1.6 Removing Encapsulated Genes Coded in Different Frames

Sometimes one encounters ORFs whose span completely encapsulates other ORFs in one or more of the remaining five reading frames. Occasionally, our method will give comparable, high scores to a pair of ORFs where one of the ORFs completely includes the other. It is believed that not both members of such pairs of coding regions can correspond to actual genes. In these situations, we use the ORF score to sub-select, and report the one with the higher score. Note that most of the time, the ORFs selected and reported in this manner correspond to the longer member of the pair. A similar approach is also employed by Glimmer [46].

5.2 Experimental Details and Results

In this section, we describe and report on the results we obtained with BDGF, the implementation of our gene-finding algorithm. BDGF was applied to several complete archaeal and bacterial genomes. We begin by describing the building of the Bio-Dictionary that we use and the parameter choices for our search scheme that determines possible matches of a given seqlet in the amino acid translation of an ORF.

5.2.1 Generation of the Bio-Dictionary

With the help of the Teiresias algorithm [117, 118], we computed an instance of the Bio-Dictionary for the June 12, 2000 release of the SwissProt/TrEMBL [15] database. The processing was carried in the manner

Table 5.1: Statistics for the seqlet-derived pattern collections used in our experiments.

Pattern Collection	#Patterns	Average on CDS	Average on non-CDS
BD-2	3,951	2,478.23	1,965.55
BD-3	309,478	151.62	104.01
BD-4	5,726,316	13.62	8.25
BD-5	17,509,665	1.98	1.02
BD-6	22,523,439	0.39	0.14

that is outlined in [119]. As a matter of fact, we used Teiresias with a setting of $L = 6$, $W = 15$ and $K = 2$. The justification for this choice of values for L and W is the result of earlier extensive analysis and was described in [58]. The Bio-Dictionary that resulted from this processing contained 29,397,880 seqlets. The instances of these seqlets accounted for 98.10% of the amino acid positions in the processed database. It is this collection that we used in our experiments.

Some of the seqlets in the employed Bio-Dictionary have rather long spans. These are typically important patterns, generated by putative proteins that are coded by genes in distinct genomes from the same phylogenetic domain. Because these patterns appear infrequently, it is difficult to compute their weights in the absence of an extraordinarily large training set. For our experiments, we handle this situation as follows: we replace seqlets that have k fixed characters by $k - s + 1$ seqlets each of which contained s fixed characters, for some choice of the value s (see below). Let $S\{i..j\}$ denote a subpattern of a seqlet S that starts at the i-th fixed character and ends at the j-th fixed character. We replace S by subpatterns $S\{1..s\}, S\{2..(s+1)\}, \ldots, S\{(k-s+1)..k\}$. Any duplicate seqlets that appear in the resulting collection are removed before further processing. Heretofore, and for simplicity purposes, we will use the shorthand notation BD-i to refer to the derived pattern collections that are constructed as above by setting s to i. The collections that we used in our experiments were BD-4 and BD-6.

Table 5.1 shows statistics for collections BD-i with i assuming values between 2 and 6 inclusive. In particular, for each BD-i the table lists the following items: a) the number of seqlet-derived patterns contained in BD-i - as expected, and because identical patterns are removed after splitting, the number is small for small values of i; and, b) the average numbers of instances of a derived pattern per 1M amino acids, computed from experiments against coding sequences (column 3) and non-coding sequences (column 4) from the seventeen genomes we used for our experiments - as expected, the number of instances is much higher in coding sequences than in non-coding ones.

The total lengths for the coding and non-coding sequence sets for the seventeen genomes that we used in our experiments are ∼30Mbp and ∼94Mbp respectively. Note that the seemingly large sizes of these datasets are due to the existence of 6 reading frames. If we make use of all seventeen organisms for training, the corresponding coding and non-coding sequence sets have sufficiently large sizes to permit the computation of representative weights, even for the patterns in the BD-6 collection. On the other hand, if only one or a handful of genomes are available, the corresponding sizes for these two sets are not substantial to permit the generation of representative weights for the BD-5 and BD-6 collections. Thus, a given choice for the training

Table 5.2: Actual query time (in seconds) using (l, w)-subpattern-based indexing.

	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$	$l = 7$	$l = 8$	$l = 9$	$l = 10$
$w = 2$	10297.62								
$w = 3$	7538.57	4716.27							
$w = 4$	7688.03	2248.43	2242.25						
$w = 5$	7730.57	1037.28	840.77	887.23					
$w = 6$	7905.72	760.68	465.05	414.75	433.82				
$w = 7$	7892.13	757.52	307.85	258.97	265.95	265.90			
$w = 8$	7921.85	750.30	172.25	148.58	154.12	153.68	153.20		
$w = 9$	7887.63	755.35	121.23	93.10	100.70	103.12	106.92	106.88	
$w = 10$	7778.68	769.63	100.92	73.25	86.15	95.50	105.55	103.28	103.82
$w = 11$	7971.78	758.70	98.90	68.13	96.98	129.92	143.38	146.43	152.45
$w = 12$	8189.87	760.25	111.12	79.83	141.28	201.75	243.38	267.60	274.33
$w = 13$	7901.72	758.78	114.60	100.78	213.30	328.28	450.63	522.22	556.00
$w = 14$	7903.68	759.87	130.92	137.07	320.48	547.75	815.77	1002.45	1152.20
$w = 15$	7889.48	764.33	140.85	184.83	465.45	914.70	1407.35	1846.40	2193.10

set indirectly dictates which collections BD-i can be used to carry out any planned gene finding experiments.

5.2.2 Performance of the Seqlet Search Scheme

As we have already mentioned, it is important that we be able to quickly determine which of the seqlets of the Bio-Dictionary are contained in the amino acid translation of a given ORF. To this end, we present experimental results on the performance of the search scheme that we described in the previous section, and for different combinations of the parameters l and w.

For the purposes of benchmarking the method's performance, we used the a) original unmodified Bio-Dictionary that contained 29,397,880 patterns and b) 100 proteins from *E. coli* as queries. The average span of the seqlets in this Bio-Dictionary is 13.15 positions whereas the average number of fixed characters in these seqlets is 7.41. The average length of the 100 query proteins is 340.61 with the shortest and longest sequences having lengths of 21 and 1,073 amino acids respectively. The goal of the experiment is to find all the Bio-Dictionary seqlets that have instances in the 100 query proteins. As it turns out, a total of 302,349 Bio-Dictionary seqlets appear in the cumulative collection of 100 proteins, with each amino acid position participating in the instances of 8.88 distinct seqlets, on average.

Table 5.2 shows the computation time required to determine all these matches and for the parameters l and w assuming values in the range $2 \leq l \leq 10$, $2 \leq w \leq 15$ ($l \leq w$). All of the experiments are carried out on a single IBM RS64III processor with a clock speed of 450MHz. For these experiments, we constructed a hash index assuming an amino acid bias that results from a uniformly distributed random sequence of nucleotides. The best performance is obtained when $l = 5$ and $w = 11$: with these settings, we can process a 500 amino acid query and determine the subset of the Bio-Dictionary's approximately 30 million seqlets that have instances in the query in approximately 1 second.

Table 5.3 shows the average number of seqlets that are examined for instances in the query sequence.

Table 5.3: Average number of seqlets that need to be examined per query unit length.

	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	$l=10$
$w=2$	201782.33								
$w=3$	124952.44	76371.22							
$w=4$	121615.54	36029.50	34732.23						
$w=5$	120480.81	15522.02	12612.06	12566.68					
$w=6$	119775.41	10576.61	5730.41	5649.36	5647.23				
$w=7$	119441.63	10036.59	3442.44	3295.59	3293.04	3292.91			
$w=8$	119260.62	9909.72	2037.39	1791.63	1788.00	1787.88	1787.85		
$w=9$	119187.04	9836.12	1321.24	954.16	948.41	948.29	948.26	948.34	
$w=10$	119162.34	9797.54	1004.48	510.67	501.35	501.22	501.20	501.27	501.26
$w=11$	119158.53	9783.17	883.61	273.85	259.01	258.88	258.86	258.93	258.92
$w=12$	119158.53	9779.73	850.41	152.38	129.47	129.34	129.31	129.39	129.38
$w=13$	119158.53	9779.73	844.78	95.72	61.46	61.34	61.31	61.38	61.38
$w=14$	119158.53	9779.73	844.78	79.28	30.18	30.05	30.03	30.10	30.09
$w=15$	119158.53	9779.73	844.78	79.28	10.97	10.84	10.81	10.89	10.88

Note that only a subset of the Bio-Dictionary's seqlets will actually match somewhere in the query. As anticipated, the number of seqlets that need to be examined decreases as l and w grow larger, an expected result. Also, the search time does not decrease when $l > 5$ or $w > 11$. The explanation can be found in Table 5.4 where we show the maximum number of subpatterns that need to be checked at each position of a given protein: this number increases as l and w increase. As a matter of fact, the actual search times are roughly proportional to $1.6 \times x + y$, where x is the number of the checked subpatterns at each position and y is the average number of the actually checked seqlets at each position. Note that x is sometimes larger than y because subpatterns exist that must be checked but for which no seqlet has been hashed. We can easily compute the expected number of checked seqlets against random sequences with a given amino acid bias, and thus can estimate appropriate values for l and w in this manner. Table 5.5 shows the estimated average number of seqlets that would be examined for instances in a random protein sequence of 1Mbp length with a uniform amino acid bias. The numbers in this table are much smaller than those in Table 5.3 because the sequences used in Table 5.3 are actual protein sequences (not random ones) and there should be much more occurrences of seqlets than estimated (see also Table 5.1). But we can use the estimated values in Table 5.5 because they are roughly proportional to those in Table 5.3. The graph of Figure 5.1 shows relationship between the search time and $0.4 \times x + \bar{y}$ where $\bar{y}$ denotes the estimated number of checked seqlets at each position in Table 5.5. We can predict the optimal l and w in this manner.

For completeness purposes, we also carried out experiments where we searched using hash keys derived from the prefixes of seqlets. The results are shown in Table 5.6 for various values of the prefix length. This scheme does not perform as well as our (l, w)-based hashing scheme and the best result of 237 seconds that is obtained for a prefix length of 11 is far too slow to be useful; thus, we abandoned this prefix scheme idea.

Table 5.4: Maximum number of subpatterns to be checked at each position.

	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	$l=10$
$w=2$	2								
$w=3$	3	4							
$w=4$	4	7	8						
$w=5$	5	11	15	16					
$w=6$	6	16	26	31	32				
$w=7$	7	22	42	57	63	64			
$w=8$	8	29	64	99	120	127	128		
$w=9$	9	37	93	163	219	247	255	256	
$w=10$	10	46	130	256	382	466	502	511	512
$w=11$	11	56	176	386	638	848	968	1,013	1,023
$w=12$	12	67	232	562	1,024	1,486	1,816	1,981	2,036
$w=13$	13	79	299	794	1,586	2,510	3,302	3,797	4,017
$w=14$	14	92	378	1,093	2,380	4,096	5,812	7,099	7,814
$w=15$	15	106	470	1,471	3,473	6,476	9,908	12,911	14,913

5.2.3 Gene Finding Results on Archaeal and Bacterial Genomes

In this section, we outline and discuss the capabilities of our gene-finding method by reporting the results we obtain from processing seventeen complete genomes with BDGF.

Genome Identities

Of the seventeen genomes we used in our experiments, 4 were archaeal (*A. fulgidus, M. jannaschii, M. thermoautotrophicum, P. abyssi*) whereas the remaining thirteen were bacterial. Table 5.7 shows relevant information for these genomes including the genome length in nucleotides, the number of all identifiable ORFs that are longer than eighteen nucleotides (*i.e.* 6 amino acids), and the number of annotated coding regions that have been reported in the public databases for each genome. We should mention at this point that each of these genomes may contain additional actual coding regions that to date have remained unreported. Also, one should not loose sight of the fact that the annotated (= reported) coding regions are for the most part putative and have typically been reported without verification *via* wet laboratory experiments.

Quantifying the Quality of our Predictions

There exist several ways in which one can evaluate the performance of a gene finding algorithm. But the algorithm's sensitivity and specificity remain the most important measures. Sensitivity, often referred to as the *prediction rate*, is defined as the ratio of the number of genes predicted by the algorithm over the number of genes that has been reported in the public databases. Specificity is defined as the ratio of the number of predicted genes that are also reported in the public databases over the number of all genes that the algorithm has predicted.

Clearly, one can generate very appealing, large values for the sensitivity of the algorithm simply by

Table 5.5: Estimated average number of seqlets that need to be examined per query unit length.

	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	$l=10$
$w=2$	134198.37								
$w=3$	73494.70	38788.61							
$w=4$	73494.70	17347.44	16506.10						
$w=5$	73494.70	6334.03	4905.88	4876.20					
$w=6$	73494.70	3846.45	1744.38	1705.71	1704.36				
$w=7$	73494.70	3674.74	974.07	920.29	918.85	918.81			
$w=8$	73494.70	3674.74	537.83	463.00	461.37	461.32	461.32		
$w=9$	73494.70	3674.74	319.13	219.56	217.58	217.54	217.54	217.54	
$w=10$	73494.70	3674.74	226.35	101.89	99.39	99.35	99.35	99.35	99.35
$w=11$	73494.70	3674.74	193.16	46.67	43.45	43.41	43.41	43.41	43.41
$w=12$	73494.70	3674.74	184.85	22.11	17.94	17.90	17.90	17.90	17.90
$w=13$	73494.70	3674.74	183.74	11.94	6.54	6.50	6.50	6.50	6.50
$w=14$	73494.70	3674.74	183.74	9.19	2.29	2.25	2.25	2.25	2.25
$w=15$	73494.70	3674.74	183.74	9.19	0.46	0.42	0.41	0.41	0.41

Table 5.6: Searching time (in seconds) when seqlets are hashed with prefixes of various lengths.

Prefix length	2	3	4	5	6	7
Search time	50539.95	31194.35	19217.67	11505.07	6492.57	3374.05
Prefix length	8	9	10	11	12	13
Search time	1646.15	755.43	359.83	236.75	305.20	559.95

lowering the employed decision thresholds. But this is typically done at the expense of introducing false positives in the output which will in turn lead to decreased values for specificity. The opposite situation is also possible: one can choose thresholds in a way that will result in high specificity values at the expense of sensitivity; *i.e.* many actual genes will not be reported. Sensitivity and specificity are competing goals and, ideally, any proposed algorithm must aim at achieving simultaneous high values for both of these measures.

In addition to an algorithm's specificity and sensitivity, also of interest is the cardinality of the collection of genes that have been predicted by the algorithm and satisfy the following two conditions:

1. the predicted genes are not among the genes that have been reported in the public databases; and,
2. the predicted genes have substantial similarity to one or more protein/cDNA sequences contained in the public repositories.

Naturally, this collection forms a subset of the results that would otherwise be characterized as "false positives."

The existence of genes in several distinct genomes that also exhibit similarity to a gene predicted by a given algorithm adds support to the hypothesis that this gene is indeed correctly predicted. In what follows we use the term 'hits' to refer to the members of the special subset of predicted genes that also satisfy

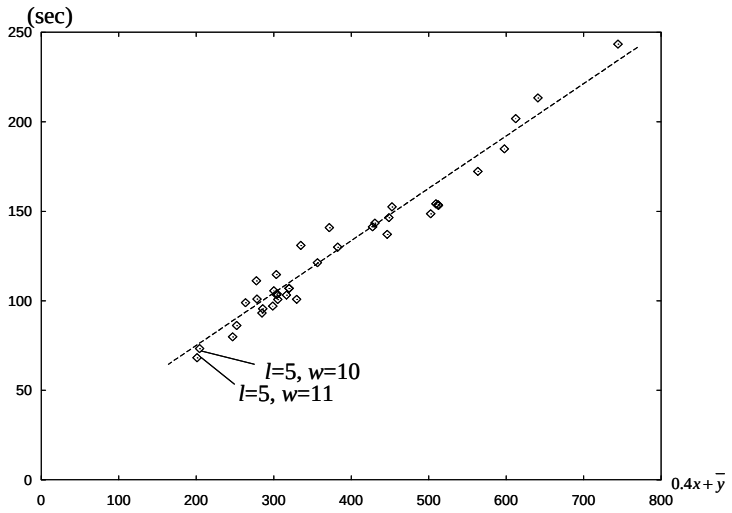


Figure 5.1: Estimation of seqlet search speed of various (l, w)-subpattern-based indexing.

conditions 1 and 2 above. In our experiments, we determined whether a predicted gene satisfied condition 2 by using both the FASTA [114] and the BLAST [8] algorithms: with default threshold and matrix settings we carried our similarity searches against the release of SwissProt/TrEMBL [15] from September 21, 2001. A query was considered to generate a hit in the searched database if one or more of the reported results had associated E(.) values that were 1.0e-4 for FASTA and 1.0e-3 for BLASTP. In addition to running FASTA and BLAST, we carried out a CD search for conserved domains using rpsblast and the Conserved Domain Database from Feb 28, 2002 [101]: the E(.) value threshold we used here was equal to 1.0e-4.

In all cases that are described below, we quantified the performance of our approach by simultaneously reporting the values of the following three measures: 'sensitivity,' 'specificity' and 'hits.'

How we Built and Used our Training and Test Sets

As we explained previously, an appropriate training set is needed in order to compute weights for all the seqlets in the Bio-Dictionary. The experiments that we carried out were meant to mimic the very wide spectrum of situations that a researcher may encounter in a real-world setting.

First, we divided each genome into two equal-length parts: we used the second half of the genome as a training set and the first half as a test set (Case 1). In spirit, this is a test similar to what has been previously reported in the literature [46, 98]. However, it should be stressed that what we use in this case to derive weights for our seqlets is a mere 50% of a genome whereas we test our prediction capability on the remaining 50% of it. The size of the training set we use in this set of experiments is much smaller than what has been typically employed to train previously reported methods. This was as intentional decision meant to showcase our system's capabilities.

We also carried out experiments without using any *a priori* knowledge for the training sets. This is discussed in detail in case 2 below. The purpose of these experiments was to determine how well our

Table 5.7: Details on seventeen genomes used in our experiments.

Type	#	Organism	Abbr.	Length	#ORF	#CDS	
						Total	<300nt
Archaeal genomes	1.	*Archaeoglobus fulgidus* DSM4304	AF	2,178,400	73,238	2,407	319
	2.	*Methanococcus jannaschii* DSM2661	MJ	1,664,970	74,456	1,715	174
	3.	*Methanobacterium thermoautotrophicum* delta H	MT	1,751,377	64,726	1,869	223
	4.	*Pyrococcus abyssi* GE5	PA	1,765,118	64,436	1,765	72
Bacterial genomes	5.	*Aquifex aeolicus* VF5	AA	1,551,335	50,591	1,523	33
	6.	*Borrelia burgdorferi* B31	BB	910,724	40,403	850	78
	7.	*Bacillus subtilis* 168	BS	4,214,814	167,735	4,101	471
	8.	*Campylobacter jejuni* NCTC11168	CJ	1,641,481	72,016	1,635	149
	9.	*Chlamydia pneumoniae* CWL029	CPc	1,230,230	50,872	1,052	84
	10.	*Chlamydia pneumoniae* AR39	CPa	1,229,853	50,840	1,110	146
	11.	*Chlamydia trachomatis* serovar D	CT	1,042,519	42,338	894	61
	12.	*Escherichia coli* K12-MG1655	EC	4,639,221	163,600	4,285	376
	13.	*Haemophilus influenzae* KW20	HI	1,830,138	83,944	1,709	161
	14.	*Helicobacter pylori* 26695	HP	1,667,867	67,227	1,567	174
	15.	*Rickettsia prowazekii* Madrid E	RP	1,111,523	53,656	835	61
	16.	*Synechocystis sp.* PCC6803	SS	3,573,470	141,204	3,169	257
	17.	*Thermotoga maritima* MSB8	TM	1,860,725	57,584	1,846	160

algorithm works in the absence of such information.

The next group of experiments (Cases 3a and 3b) was designed to examine the performance of our method in the case where the seqlets' weights were not genome-specific. For each test genome, we derived weights for the seqlets by training with a collection of several complete genomes that did not include the genome under consideration; we then applied our method on this test genome. These jack-knifing experiments are typically too severe for statistical methods such as those based on Markov models. It is for this reason that many web implementations of previously reported, statistics-based methods often provide several parameter settings derived from training on various genomes: users are asked to select the appropriate settings to be used by the algorithm. The experiments for cases 3a and 3b were carried out using BD-4; *i.e.* the derived pattern collection was constructed by setting the parameter s to 4 (see discussion in subsection 5.2.1).

In order to determine the impact of the different pattern collections on the results, we repeated the experiments of cases 3a and 3b using BD-6, *i.e.* the collection constructed by setting the parameter s to 6. Cases 4a and 4b correspond to this set of experiments and are the counterparts of cases 3a and 3b respectively.

Finally, for our last group of experiments (Case 5) we used all seventeen genomes to assign weights to the seqlets, then tested the resulting non-genome-specific system by processing each of the genomes in turn.

As we noted in the introduction, in this first incarnation of our system we do not make use of any additional information (for example: promoter information that can either be computed or retrieved from the public databases) to constrain the discovery of genes and start sites. Nonetheless, as evidenced by the results that we report in cases 4 and 5, our start site prediction rates are comparable in quality, and at times superior to those that have been previously reported in the literature [61, 75, 98, 147]. What is more,

our results are achieved using a system that is generic and not genome specific. We will revisit the topic of start-site prediction at the end of this section.

Format of the Reported Results

In the tables that follow, and for each processed genome, we report the total number of genes predicted by our algorithm in column 2; we also indicate how many of these genes are shorter than 300 nucleotides in column 3. In our studies, we threshold at that score value for which the number predicted genes is equal to the number of annotated genes in the public databases – clearly this threshold value is different for each genome. Note that in this case, the exhibited sensitivity and specificity are equal; this common value is shown in column 4. For a subset of the genes that are predicted by our algorithm there is no corresponding database entry characterizing them as such. Column 5 shows how many of the predicted genes fall in this category, whereas column 6 indicates how many of these genes are shorter than 300 nucleotides. With the help of FASTA, BLAST and CD-search we report how many of these genes are in fact hits in columns 7, 9 and 11 respectively; the number of hits which correspond to gene predictions that are shorter than 300 nucleotides is listed in columns 8, 10 and 12 respectively. Finally, in each of the result tables we also report on our ability to correctly predict the start sites for the reported genes through comparison with the existing database annotations. In column 13, we show the number of genes whose start site is correctly predicted; and, in column 14 we list the same figure as a percentage of reported genes.

Prediction Results on the Various Genomes

We now report on the results of our method in five experimental settings. The experiments were designed so as to mimic the type of situations that a real-world researcher is likely to encounter, and showcase the performance of our approach across a wide spectrum of settings.

Case 1 (BD-4 & Weights Derived From The 2nd Half Of Each Genome Only.) BD-4 was used in this case. For each genome, the seqlets' weights were obtained by training on the second half of the genome. Gene prediction was carried out on the first half. Table 5.8 shows the results for this experiment: in all seventeen cases, the sensitivity/specificity value ranged between 90.1% and 95.1%. Also, approximately 33% of the additional putative genes reported by our method corresponded to hits, *i.e.* we could find statistically significant similarities with proteins in the September 21, 2001 installment of the SwissProt/TrEMBL database. The implication of this is that the actual gene prediction rate is likely to be even higher than what we report here. With respect to the start site prediction rate, the rates of correctly predicted start sites in this case range from 55.6 to 84.2%. Recall that we currently make no use of any promoter information.

Case 2 (BD-4 & Weights Derived From Using Long ORFs Only.) The previous experiment assumes the availability of annotations for at least some of the actual coding genes of a target genome. But what if such information is not available? In such a situation the only recourse is to derive the seqlet weights by restricting ourselves to the very long ORFs that can be identified in the genome which is being processed; the implicit assumption here is that long ORFs are more likely than short ORFs to be coding for genes and can thus be used as training sets. Similar heuristics have been employed by other groups as well [12, 46].

Table 5.8: Gene Prediction Results for Case 1.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	1,113	121	92.9	85	39	41	14	42	15	26	6	805	72.3
MJ	866	92	93.7	58	37	21	8	24	9	8	0	711	82.1
MT	886	80	93.7	60	23	4	3	5	4	1	0	746	84.2
PA	857	30	94.7	48	41	12	10	11	9	1	1	579	67.6
AA	712	11	93.0	54	37	18	15	23	17	16	12	578	81.3
BB	398	21	91.7	36	34	6	4	6	4	1	0	240	60.3
BS	1,831	143	94.7	102	80	33	25	38	27	9	6	1017	55.6
CJ	769	60	94.2	47	36	16	9	19	12	10	7	529	68.8
CPc	473	16	92.0	41	20	23	6	23	6	12	2	332	70.2
CPa	511	30	90.1	56	40	14	3	14	3	6	0	362	71.0
CT	430	20	94.1	27	22	4	3	4	3	0	0	270	62.8
EC	2,048	139	94.2	127	53	30	17	32	18	5	2	1406	68.7
HI	802	46	95.1	41	17	27	5	26	4	17	1	572	71.3
HP	708	62	92.4	58	43	23	10	23	10	8	0	435	61.5
RP	403	22	92.9	31	24	5	0	7	1	5	0	283	70.2
SS	1,544	98	95.0	82	64	14	10	15	11	3	3	984	63.7
TM	847	28	95.0	45	6	18	3	21	5	16	4	571	67.4

In this case, we used as a training set for the coding regions all the ORFs which were longer than 600 nucleotides and which were not included within other longer ORFs. As a training set for the non-coding regions we used all the ORFs that were shorter than 200 nucleotides and which occasionally (and incorrectly) include *bona fide* coding ORFs. We again used BD-4 as the collection of patterns for which to derive weights. As will become evident after we have described our complete set of experiments, it is not necessary to carry out this kind of training when dealing with a new genome – we have simply included case 2 for the purpose of completeness of description.

Table 5.9 shows the results for this experiment. In this case, the sensitivity and specificity value ranged from 89.9% to 95.6%. Similarly to case 1, approximately 32% of the additional predicted genes, have significant similarities with other database entries. It is notable that despite the fact that the amount of information we used for training purposes was substantially less than the one we used in case 1, the performance levels remained essentially unchanged.

Case 3a (Leave-Many-Out: BD-4 & Weights Derived From Fixed 4 Of 17 Genomes.) Another realistic situation is the one where users will carry out gene prediction on newly sequenced genomes for which little or no information is yet available. The situation can be facilitated if a phylogenetically similar genome already exists in the public databases but this is not always going to be the case. This particular experiment is meant to simulate the situation where the genomes that are already available in the public databases are few and rather distant from the ones that are being examined. To this end, we used the union of the full CDS lists that were reported for *Bacillus subtilis*, *Campylobacter jejuni*, *Helicobacter pylori*, and *Rickettsia prowazekii* to derive the weights for BD-4's patterns and subsequently tested our method by predicting genes for the remaining thirteen genomes of our genome collection.

Table 5.9: Gene Prediction Results for Case 2.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,239	200	93.0	168	47	66	17	69	18	43	7	1641	73.3
MJ	1,636	145	95.4	79	47	35	11	38	11	18	1	1295	79.2
MT	1,761	130	94.2	108	30	7	4	7	5	1	0	1492	84.7
PA	1,674	56	94.8	91	75	21	17	23	19	5	4	1130	67.5
AA	1,425	18	93.6	98	60	28	20	38	26	21	16	1148	80.6
BB	792	49	93.2	58	52	4	2	5	2	1	0	482	60.9
BS	3,897	321	95.0	204	158	74	55	85	62	16	11	2134	54.8
CJ	1,567	122	95.8	68	48	28	13	29	14	12	7	1126	71.9
CPc	984	51	93.5	68	42	30	11	31	11	14	2	692	70.3
CPa	998	67	89.9	112	83	29	8	30	8	16	1	711	71.3
CT	854	49	95.6	39	29	6	5	8	5	0	0	585	68.5
EC	3,974	182	92.7	311	64	58	24	61	23	16	5	2707	68.1
HI	1,621	108	94.9	88	36	56	10	55	9	32	2	1171	72.2
HP	1,465	130	93.5	102	72	48	21	49	21	20	3	872	59.6
RP	795	48	95.3	39	22	14	4	16	5	12	4	595	74.8
SS	3,017	184	95.2	152	119	24	19	27	23	6	6	1945	64.5
TM	1,726	70	93.5	120	8	44	3	46	5	32	3	1119	64.8

Case 3b (Jack-knifing or Leave-One-Out: BD-4 & Weights Derived From 16 Of 17 Genomes.) Here we study the jack-knife variation of case 3a: prior to carrying out gene prediction for each of the seventeen genomes of the collection, we used the union of the full CDS lists from the remaining sixteen genomes to derive the weights for BD-4's patterns.

Tables 5.10 and 5.11 show the results for cases 3a and 3b. As one would intuitively expect, the weights that are derived from 16 genomes are more representative (case 3b). Consequently, the prediction rates reported in Table 5.10 are superior to those reported in Table 5.11. The sensitivity/specificity value was in the range of 86.3% to 94.7%, exceeding the 90% mark for the majority of the genomes. Approximately 23% of the additional putative genes reported by our algorithm correspond to hits.

This set of experiments (*i.e.* cases 3a and 3b) is particularly interesting in light of observations made previously in the literature according to which the performance of statistical methods typically deteriorates if the method is used with parameters derived from a phylogenetically distant genome. This deterioration reflects merely the overall difference in the statistical properties of the respective genomes.

Unlike statistical approaches, our method depends largely on the statistical properties of verified and putative proteins and exhibits resilience to statistical variability. Thus, the conclusion of this set of experiments is that if the weights of the seqlets have been appropriately derived we expect to be able to reach very high prediction levels in a manner that will be relatively independent of the studied organism. This is indeed corroborated by the results shown in Tables 5.10 and 5.11.

Case 4a (Leave-Many-Out: BD-6 & Weights Derived From Fixed 4 Of 17 Genomes.) In this experiment, we repeat the experiment of case 3a but now using the BD-6 collection. In general, with smaller values for s one expects that the derived collection of patterns will be more sensitive but will capture less

Table 5.10: Gene Prediction Results for Case 3a.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,210	202	91.8	197	71	66	16	70	18	42	6	1528	69.1
MJ	1,636	135	95.4	79	57	28	9	28	7	18	1	1297	79.3
MT	1,713	124	91.7	156	76	6	3	7	5	1	0	1291	75.4
PA	1,669	51	94.6	96	77	24	19	27	22	6	5	1075	64.4
AA	1,414	18	92.8	109	69	29	21	39	27	22	17	1105	78.2
BB	779	47	91.6	71	66	4	2	6	2	1	0	474	60.8
CJ	1,565	114	95.7	70	48	25	9	26	10	13	8	1123	71.8
CPc	951	50	90.4	101	78	30	11	30	11	12	1	542	57.0
CPa	968	64	87.2	142	118	23	4	23	4	14	1	567	58.6
EC	3,941	212	92.0	344	127	53	27	54	26	16	5	2352	59.7
HI	1,601	100	93.7	108	59	58	12	58	12	32	3	1116	69.7
SS	2,889	149	91.2	280	208	16	12	19	14	3	3	1685	58.3
TM	1,696	78	91.9	150	25	40	4	44	7	32	4	933	55.0

'structure.' On the other hand, larger s values will give rise to a situation where potentially fewer seqlets match the putative amino acid translations and there is an associated increased difficulty to appropriately compute the seqlets' weights. Table 5.12 shows the results of this experiment.

Case 4b (Jack-knifing or Leave-One-Out: BD-6 & Weights Derived From 16 Of 17 Genomes.) Here we repeat the experiment of case 3b but this time using the seqlets from the BD-6 collection; results are shown in Table 5.13.

When we compare the results from cases 3a/3b with those from cases 4a/4b we conclude that for most of the genomes in our collection, and for the training carried out as described above, the BD-4 collection will result in better performance; a notable exception is represented by the *Chlamydophila* and *Chlamydia* species for which BD-6 gives better results.

An additional observation is that the number of genomes for which BD-6 performs better than BD-4 increases as the size of the training set increases. The very important ramification of this is that, if we have access to a rather large training set, our method will exhibit better prediction performance when used in conjunction with BD-s sets corresponding to larger s values.

Case 5 (BD-6 & Weights Derived From All 17 Genomes.) The four experimental cases above provided sufficient information that permitted us to generate optimal results with our method. In particular,

- we should use the BD-6 collection since it is bound to perform better than BD-4 as more and more information is deposited in the public databases; and,
- during training, the weights should be derived from all of the previously published coding/non-coding information for all available genomes.

With these two observations at hand, we carry out this last set of experiments noting that it is indicative of the levels of prediction quality we can expect when we use a good training set, *i.e.* one derived from many complete genomes.

Table 5.11: Gene Prediction Results for Case 3b.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,225	212	92.4	182	69	66	19	70	21	42	7	1571	70.6
MJ	1,642	135	95.7	73	53	27	9	27	7	17	1	1362	82.9
MT	1,724	144	92.2	145	77	6	4	7	6	1	0	1312	76.1
PA	1,671	51	94.7	94	78	21	17	23	19	5	5	1113	66.6
AA	1,415	17	92.9	108	69	30	22	41	28	24	18	1115	78.9
BB	777	48	91.4	73	68	4	2	6	2	1	0	495	63.7
BS	3,837	331	93.6	264	222	70	50	83	59	15	11	1995	52.0
CJ	1,570	115	96.0	65	45	22	8	22	8	11	7	1131	72.1
CPc	936	47	89.0	116	99	25	10	25	10	10	1	520	55.6
CPa	958	64	86.3	152	135	19	4	19	4	11	1	544	56.8
CT	805	53	90.1	88	77	5	4	5	4	0	0	413	51.3
EC	3,898	218	91.0	387	144	58	30	57	28	15	4	2281	58.5
HI	1,596	98	93.4	113	63	57	10	59	12	32	3	1091	68.4
HP	1,455	113	92.9	112	79	46	19	46	18	19	2	904	62.2
RP	772	43	92.6	62	48	13	4	17	7	11	4	537	69.6
SS	2,875	168	90.7	294	238	20	16	21	17	4	4	1572	54.7
TM	1,696	81	91.9	150	26	41	5	44	8	32	4	951	56.1

The results for this experiment are shown on Table 5.14: the achieved sensitivity/specificity value is in the 93.9% to 97.0% range with most of the genomes exceeding the 95% mark. Invariably, and similarly to all of the above experiments, a substantial percentage of the additional putative genes that our method reports correspond to hits, *i.e.* they can be corroborated with the help of sequence similarities to entries in the public databases.

Another very notable result that is reported in Table 5.14 has to do with our ability to correctly predict the start sites of genes. As can be seen, the ratio of correctly predicted start sites ranges between 80% and 90% across almost all studied genomes. More importantly, this ratio is achieved simultaneously with very high specificity and sensitivity values and without making use of any promoter information.

We conclude by stressing a very important point: our method derives and uses a single set of weights for the seqlets in the Bio-Dictionary and these weights are not genome-specific.

On the Prediction of Start Sites

As we mentioned already, the accurate prediction of start codon sites is a notoriously difficult problem. To derive the various ratios for our experiments, we made use of the gene starts that are reported in the annotated database entries for each processed genome. Although this kind of information is generally correct, errors are known to exist. Thus, we also carried out a verification step of our start site prediction rates using experimentally validated genes. In particular, we focused on the 1,248 experimentally validated genes from *B. subtilis* [75] and computed the ratio of start sites that were correctly predicted by our algorithm when using the pattern collections and weights described in cases 4b and 5 above. Of the 1,248 genes, we correctly determined the start sites for 797 (case 4b) and 898 genes (case 5) respectively, or 63.9% and 72.0% of

Table 5.12: Gene Prediction Results for Case 4a.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,159	164	89.7	248	140	58	13	61	12	33	1	1617	76.1
MJ	1,598	108	93.2	117	113	33	10	35	9	18	1	1268	80.6
MT	1,686	113	90.2	183	129	14	7	16	9	1	0	1227	73.4
PA	1,630	39	92.4	135	114	31	21	33	23	5	4	1163	71.6
AA	1,414	13	92.8	109	79	24	19	32	23	19	15	1171	83.6
BB	759	31	89.3	91	84	5	3	6	3	1	0	558	73.5
CJ	1,539	91	94.1	96	92	24	10	23	9	9	6	1252	82.4
CPc	967	40	91.9	85	76	28	13	28	13	12	2	701	73.3
CPa	985	54	88.7	125	105	24	8	24	8	13	1	726	74.0
EC	3,867	150	90.2	418	326	81	43	84	46	16	6	2754	73.0
HI	1,588	75	92.9	121	109	59	15	55	11	32	2	1222	78.9
SS	2,851	117	90.0	318	314	21	18	24	19	3	3	2032	72.6
TM	1,694	76	91.8	152	50	39	5	43	8	30	5	1154	68.1

the processed set respectively. These numbers closely match the corresponding entries for the entire *B. subtilisgenome* in Tables 6 (case 4b) and 7 (case 5) respectively; in fact, they are slightly higher than what is listed in these Tables. This verification lends more support to the correctness of our start site predictions.

We conclude the discussion on start sites by noting that our gene prediction algorithm can also be used for start site localization. In Figure 5.2 we show a graph that depicts for each position i in the neighborhood of an *E. coli* coding sequence the local sum of the weights for the seqlets that match starting at position i. To obtain the cumulative score corresponding to position i, the local sums from i through the stop codon position need to be added together. The seqlet weights used here are the ones from case 5 above and correspond to what we consider to be an optimal setting. The true start site of the coding region as well as alternative start sites are shown in this plot. Note how the cumulative score that our method computes is very low just prior to the true start codon then jumps abruptly to a much higher value immediately after it. These score jumps can be exploited to predict the start sites of predicted genes. And as the results for case 5 have showed, we can achieve high ratios of correct start site prediction simultaneously with high levels of specificity and sensitivity. We are in the process of incorporating information from promoter regions to our system and expect that the overall prediction capability of our method will improve further.

Web Server for this Gene Finding Algorithm

BDGF, the implementation of our gene finding algorithm has been made available via the World Wide Web. The implementation uses the patterns of the BD-6 collection and optimal weight settings derived from seventeen genomes. The server can be accessed by visiting the BDGF web site [23] and is operational around the clock; the server runs on a single IBM RS64III processor with a 450 MHz clock, and can process 250,000 nucleotides in all 6 reading frames in a little over 60 seconds. Upon completion of the computation, the results are presented to the user via a graphical user interface an instance of which is shown in Figure 5.3. The predicted genes are color-coded depending on the score they have been assigned. The interface allows

Table 5.13: Gene Prediction Results for Case 4b.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,181	178	90.6	226	124	60	15	62	14	35	2	1652	76.7
MJ	1,612	113	94.0	103	98	31	9	32	8	18	1	1341	84.3
MT	1,713	129	91.7	156	108	12	8	14	10	1	0	1299	76.1
PA	1,640	41	92.9	125	104	29	20	31	22	5	4	1177	71.9
AA	1,419	12	93.2	104	76	23	18	32	23	19	15	1165	82.8
BB	758	34	89.2	92	83	7	5	8	5	1	0	552	72.6
BS	3,738	238	91.1	364	360	51	36	58	40	12	8	2217	60.0
CJ	1,539	93	94.1	96	95	23	11	21	9	9	6	1228	80.6
CPc	997	45	94.8	55	46	25	14	25	14	7	1	777	78.2
CPa	1,017	67	91.6	93	78	20	9	20	9	8	1	804	79.1
CT	821	46	91.9	72	75	8	7	8	7	0	0	536	66.5
EC	3,891	160	90.8	394	299	74	40	79	43	16	6	2647	69.8
HI	1,599	82	93.6	110	87	54	10	52	8	32	2	1228	78.2
HP	1,435	91	91.6	132	130	42	14	42	14	21	3	1038	73.9
RP	777	32	93.2	57	66	12	3	15	5	8	1	592	78.3
SS	2,862	132	90.3	307	310	23	20	25	21	4	4	1940	69.0
TM	1,706	82	92.4	140	48	38	5	41	8	31	5	1137	66.5

the user to navigate around the processed DNA sequence and to zoom in/out of the regions of interest. Once an ORF has been selected with the help of the mouse, its location on the processed sequence is reported together with its nucleotide composition, length and amino acid translation. The amino acid translation of a selected ORF can be annotated interactively. Also, the minimum length of an ORF that will be reported is controlled by the user: its default value is 50 amino acids (*i.e.* 150 nucleotides). The complete list of ORFs that have been predicted by BDGF together with their position, length and associated score can also be downloaded from the same page.

5.3 Summary

In this chapter, we described a new method for solving the gene identification problem. Our method begins with the Bio-Dictionary, a collection of patterns that is generated from processing very large public databases with the help of the Teiresias algorithm. The collection accounts completely for the processed input, and discovers genes by making use of this set of patterns alone. The method is augmented by associating each of the used patterns with automatically-derived weights. These weights are genome-independent and thus remain fixed across genomes.

Through a series of carefully designed experiments we extensively explored various settings that mimicked real-world situations, and determined the optimal settings for our gene finding approach. As evidenced by reported experimental results from seventeen archaeal and bacterial genomes, our method can predict genes very accurately. The method achieves sensitivity and specificity values that are simultaneously very high while at the same time achieving a high rate of correctly predicted start sites. Notably, no promoter or other

Table 5.14: Gene Prediction Results for Case 5.

Abbr.	#Reported Genes		Sns. = Spc.	Additional Genes								Start Site	
				#		FASTA		BLAST		CD Search		Exact	Ratio
	Total	<300		Total	<300	Total	<300	Total	<300	Total	<300		
AF	2,294	209	95.3	113	76	52	10	54	11	34	2	1835	81.5
MJ	1,662	127	96.9	53	52	23	5	24	4	17	1	1476	89.9
MT	1,809	160	96.8	60	50	5	3	6	5	1	0	1499	83.9
PA	1,693	41	95.9	72	72	16	13	17	14	2	1	1306	77.7
AA	1,457	11	95.7	66	48	18	15	23	18	15	12	1279	88.3
BB	799	43	94.0	51	42	4	3	5	3	1	0	672	83.7
BS	3,955	305	96.4	147	175	36	24	42	29	10	7	2643	67.8
CJ	1,585	110	96.9	50	42	21	10	18	7	7	5	1375	87.2
CPc	1,020	51	97.0	32	32	23	14	23	14	6	1	838	83.0
CPa	1,042	82	93.9	68	58	15	7	15	7	6	1	888	85.2
CT	865	43	96.9	28	29	5	4	5	4	0	0	662	76.9
EC	4,157	214	97.0	128	120	45	23	45	22	16	6	3256	80.2
HI	1,655	95	96.8	54	38	45	6	44	5	27	1	1384	84.9
HP	1,503	110	96.0	63	53	35	9	35	9	18	1	1222	82.2
RP	808	41	96.9	26	22	9	2	11	3	6	1	688	85.7
SS	3,063	159	96.7	106	140	12	9	16	12	2	2	2423	80.5
TM	1,770	96	95.9	76	30	35	6	38	9	29	6	1300	73.9

information is brought to bear during our determination of the genes and/or start sites.

We demonstrated the capabilities of our method to extrapolate by intentionally relying upon a Bio-Dictionary that was built from the June 12, 2000 release of SwissProt/TrEMBL, *i.e.* a public collection of sequences that is by now more than a year and a half old. We nonetheless applied the resulting system to genomes whose ORF translations were included in SwissProt/TrEMBL either only in part or not at all, with exceptional results.

We should note that in addition to correctly discovering and reporting those ORFs that have already been listed in the public databases as putative genes, our method determines additional candidate genes in essentially all of the genomes that were used in the experiments: for a substantial fraction of these previously unreported genes, and with the help of FASTA, BLAST and CD-search, we determined similarities with amino acid sequences contained in a very recent release of SwissProt/TrEMBL. Such similarities further support the hypothesis that these ORFs ought to have been reported as putative genes in the first place.

We are currently in the process of pursuing several related topics that include: the determination of the performance impact from using Bio-Dictionaries built from selected collections of proteins and not from full-size datasets; the determination of a compact collection of seqlets for the express purpose of efficient gene identification; extending our strategy to the case of eukaryotic genomes, etc.

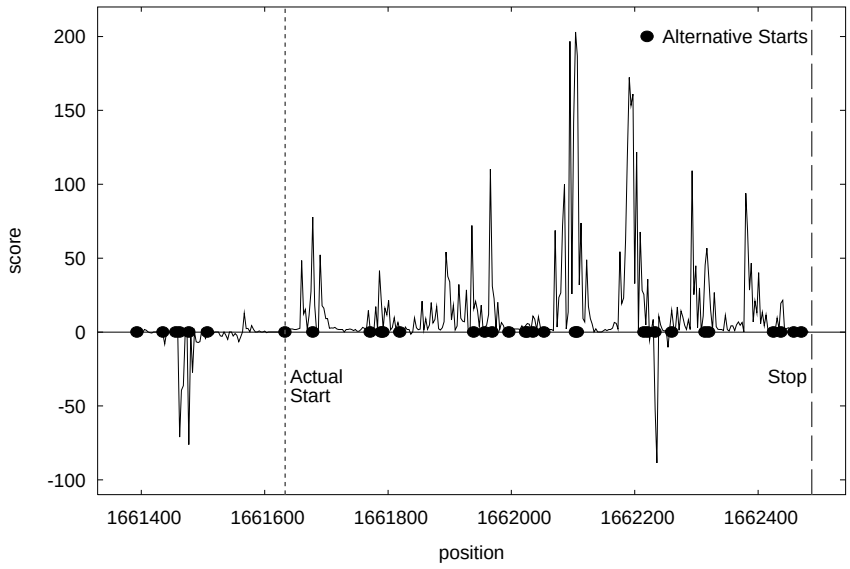


Figure 5.2: Example of start site prediction using a coding sequence *E. coli*.

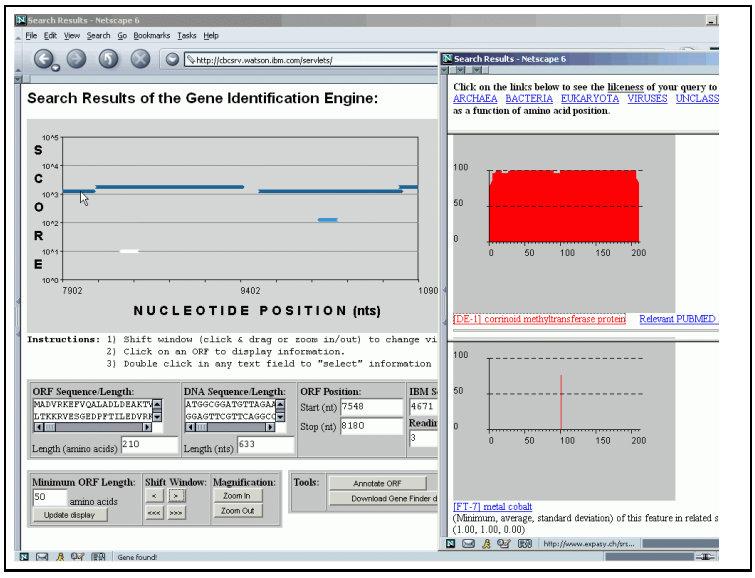


Figure 5.3: Result Page of the Web-based graphical user interface.

Chapter 6

Suffix Tree Data Structures for RNA Structure Analyses

The 3-D structure of a biological sequence plays a major role in determining its functions and properties, and sequences that have similar structures often have similar functions, even if the sequences themselves are not similar. But it is very difficult to predict and/or analyze the structure of a given sequence correctly and efficiently. Hence it seems to be still harder to find structurally similar regions among several biological sequences or to find a set of frequently appearing and structurally similar regions in a given sequence. Thus molecular biologists often search for only similar, or highly conserved regions from DNA, RNA or protein sequences to find regions with similar functions, because similar sequences have tendency of having the same structure. Though many such methods are very fast, they do not detect regions that are structurally similar to each other but not similar in the string sense.

Our data structure and our algorithm described in section 6.1 enables mining unknown important RNA structures from a large set of sequences efficiently in a linear time, by generalizing the p-suffix trees [16, 18, 19] into what we call the s-suffix trees. Furthermore, we propose an $O(n(\log|\Sigma| + \log|\Pi|))$ on-line algorithm for constructing the s-suffix tree where n is the sequence length, $|\Sigma|$ is the size of the normal alphabet, and $|\Pi|$ is that of the alphabet called "parameter," which is related to the structure of the sequence. Our algorithm achieves a linear time when it is used to analyze RNA and DNA sequences. Furthermore, as an algorithm for constructing the p-suffix trees, it is the first on-line algorithm, though the computing bound of our algorithm is same as that of Kosaraju's best-known algorithm [95], and moreover it is much simpler than Kosaraju's algorithm. The results of computational experiments using actual RNA and DNA sequences are also given to demonstrate our algorithm's practicality.

It is known that an RNA secondary structure can be described with a tree data structure [160]. Thus finding frequent patterns from a set of trees is an important issue in the analyses of RNAs. The suffix tree of a CS-tree we introduced in subsection 2.2.3 is a very useful data structure for such work. This can also be used for tasks such as minimizing sequential transducers of deterministic finite automata [32] and tree pattern matching [94]. Kosaraju [94] mentioned that the generalized suffix tree of a CS-tree can be constructed in $O(n \log n)$ time where n is the size of the CS-tree. Breslauer [32] improved this bound by giving an $O(n \log|\Sigma|)$ algorithm. Note that both of the algorithms were based on Weiner's suffix tree

construction algorithm [163]. But this algorithm becomes $O(n \log n)$ when Σ is large. In section 6.2, we improve their bound by giving an optimal $O(n)$ algorithm for integer alphabets.

In section 6.3, we deal with a new data structure called a Bsuffix tree, which is a generalization of the suffix tree of a string. Using the suffix tree of a CS-tree, we can find a given path in a tree very efficiently. The Bsuffix tree is a data structure that enables us to query any given completely balanced k-ary tree pattern from a k-ary tree or forest very efficiently. It also enables finding patterns from trees, and is very useful for RNA structural studies. Note that the concept of a Bsuffix tree is very similar to that of an Lsuffix tree [9, 65, 92], which enables us to query any square submatrix of a square matrix efficiently. We will show that this data structure can be built in $O(n)$ time for integer alphabets. Bsuffix trees have many useful features in common with ordinary suffix trees. For example, using this data structure, we can find a pattern (a completely balanced k-ary tree) efficiently in a text k-ary tree. Moreover, we can enumerate common completely balanced k-ary subtrees in a linear time. Considering that general tree pattern matching requires an $O(n \log^3 n)$ time [43], these results mean that a Bsuffix tree is a very useful data structure.

6.1 Generalization of a Suffix Tree for RNA Structural Pattern Matching

In this section, we propose a new data structure called an s-suffix tree by generalizing the p-suffix tree. We also discuss how to describe structural patterns of RNA or DNA here. Using the s-suffix tree, we can efficiently find some set of substrings in some given sequence(s) that might be structurally similar, query substrings that might be structurally similar to another given string, and so on. We also propose an efficient on-line algorithm for constructing an s-suffix tree based on Ukkonen's algorithm. Finally, we give the results of simple computational experiments using several HIV RNA complete sequences and very large DNA sequences of *E. coli* (*Escherichia coli*).

6.1.1 RNA Structural Matching

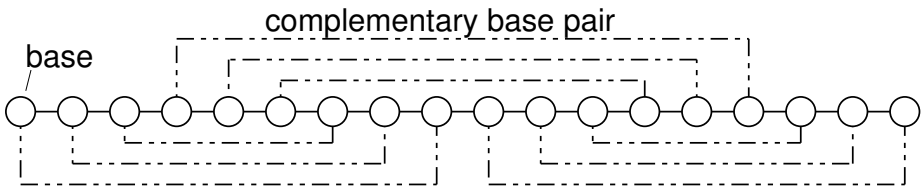
RNA sequences consist of four kinds of bases: `A` (adenine), `U` (uracil), `C` (cytosine), and `G` (guanine). Note that in DNA, `T` (thymine) is present instead of `U`. `A` and `U` (`T` for DNA) are said to be complements of each other, and `C` and `G` are also complementary bases. RNA and single-stranded DNA sequences often form some structures by combining two complementary base pairs. These combining pair of bases are called Watson-Crick pairs. It is known that double-stranded DNA sequences sometimes form such structures by becoming single-stranded locally. Note that a base sometimes combines with more than one complementary base: The triplex structure is the famous example. Many computational studies have been done to predict RNA structures, comparing a new sequence with a known RNA structure, searching a known RNA or DNA structures from large databases, and so on [2, 73, 97, 99, 129, 153, 158, 160]. But there has been no appropriate method that can mine an unknown important RNA structure from a large data set efficiently in a linear time, which is the aim of the algorithm presented in this section.

Let us consider the two RNA sequences in Figure 6.1 (1). The two sequences are not at all similar to each other: there are no identical bases in identical positions. In sequence 1, `A`'s are located at the 1st, 3rd, 8th, and 15th positions. In sequence 2, `C`'s are located at the same position as `A`'s in sequence 1. Similarly,

Sequence 1: AUAUCGUAUGGCCGAGCC

Sequence 2: CGCGUAGCGAAUUACAUU

(1) Example sequences



(2) Candidate structure

Figure 6.1: Examples of sequences that have high possibility to have a same structure.

A's, U's, and G's in sequence 2 are located at the same positions as G's, C's, and U's in sequence 1, respectively. Recall that A and U can combine with each other, and that C and G can also combine with each other. We then notice the following fact: If two bases in one of these sequences can combine with each other, then in the other sequence, two bases at same two positions are also able to combine with each other. This implies that a structure that can be formed by one of the sequences can also be formed by the other sequence. Thus there is a strong possibility that these two sequences have the same structure, and consequently may have similar properties. For example, Figure 6.1 (2) shows one of the structures that can be formed by sequence 1. It is easy to see that it can also be formed by sequence 2. The algorithm we propose from now on enables us to find such a set of substrings from a given sequence.

6.1.2 s-Strings and s-Suffix Trees

In this section we define s-strings and s-suffix trees, which are generalizations of p-strings and p-suffix trees.

Definition 5 *Let Σ and Π be disjoint finite alphabets. We call the characters in Σ the "fixed symbols" and those in Π the "parameters." Some of the characters in Π have one-to-one correspondences to other characters in Π, and two characters that correspond to each other are called complementary characters or complements of the other. No two characters can be complements of one same character. A string in $(\Sigma \cup \Pi)^*$ is called a structural string, or s-string for short. Two s-strings S and S' are said to s-match if they satisfy the following two conditions: (1) there exists a one-to-one mapping from Π to Π such that S becomes S' as a result of applying it, and (2) if x is mapped to y in the mapping, then the complement of x is also mapped to the complement of y in the mapping.*

For example, if $\Sigma = \{\mathtt{A},\mathtt{B}\}$, $\Pi = \{x, y, z, w\}$, and x and y are complements of z and w, respectively, then ABxBYyAzwz and ABwBXxAyzy s-match, but ABxBYyAzwz and ABwBXxAzyz do not. Note that if there are no complementary pairs in Π, an s-string is the same as a p-string. Note also that the complement of a given

character can be accessed in $O(\log|\Pi|)$ time if the information is stored in a balanced tree data structure, which can be constructed in $O(|\Pi|\log|\Pi|)$ time. If Π can be used as an index to a table, the complement can be obtained in $O(1)$ time.

The problem of the RNA (or DNA) structural matching described in section 6.1.1 is the problem of s-matching in the following situation: $\Sigma = \phi$, $\Pi = \{\texttt{A, U, C, G}\}$, and `A` and `C` are complementary characters of `U` and `G`, respectively. If two RNA sequences s-match with each other, it can be said that there is a high possibility that the two sequences have the same structure and that they may have similar properties as a result. For example, the two sequences in Figure 6.1 (1) s-match.

The following two encodings are useful for determining s-matching of two sequences. One is prev(S) that is already defined in Definition 2. The other is compl(S) defined as follows:

Definition 6 *Let N be the set of nonnegative integers ($N \notin \Sigma \cup \Pi$). Consider a string $S[1..n] \in (\Sigma \cup \Pi)^*$. If $S[i] \in \Pi$, let c_i be the index of the nearest complementary parameter in Π to the left, i.e., $c_i < i$, $S[c_i] = x_i$ and $S[k] \neq x_i$ for any k such that $c_i < k < i$, where x_i is the complement of $S[i]$. If such c_i does not exist, let $c_i = i$. Now, replace $S[i]$ with $i - c_i \in N$ if $S[i] \in \Pi$, for all i: We let the obtained string in $(\Sigma \cup N)^*$ be compl(S).*

For example, compl(`AB`x`B`y`A`zwz) = `AB0B0A436` if $\Sigma = \{$`A`,`B`$\}$, $\Pi = \{x, y, z, w\}$, and x and y are complements of z and w, respectively. Notice that this definition is very similar to that of prev encoding. We can compute prev and compl encodings for string S of size n in $O(n \cdot \min(\log n, \log|\Pi|))$ time and $O(n)$ space by means of a balanced tree structure, which can be computed on-line. If Π is known and can be used as an index to a table of $|\Pi|$, it is easy to see that these encodings can be computed in $O(n + |\Pi|)$ time and space.

These two encodings are related to finding s-matches as follows: s-strings S and S' are an s-match if and only if prev(S) = prev(S') and compl(S) = compl(S'). Furthermore, it is easy to see the following lemma. Let prev(S)[i] and compl(S)[i] denote the ith characters of prev(S) and compl(S) respectively.

Lemma 1 *Consider a situation in which $S[1..i]$ and $S'[1..i]$ are an s-match. In this situation, if prev(S)[i+1] = prev(S')[$i+1$] $\neq 0$, compl(S)[$i+1$] = compl(S')[$i+1$]. Similarly, if compl(S)[$i+1$] = compl(S')[$i+1$] $\neq 0$, prev(S)[$i+1$] = prev(S')[$i+1$].*

This means that, when we check s-matches of strings, we do not have to see the other encoding if one of the encodings encodes a character as a non-zero number. Using this lemma, we can check s-matching by using the following s-encoding:

Definition 7 *For a given string S, compute prev(S) and compl(S). If prev(S)[i] = 0, replace it with $-$compl(S)[i], which is a nonpositive value. We call this new encoded string in $(\Sigma \cup I)^*$ (I: integer) as a structural encoding of S, or an s-encoding for short.*

The structural suffix tree of string S, or the s-suffix tree of S for short, is the compacted trie of the s-encoded strings of all the suffixes of $S^+ = S\$$, where \$ is a character that is in neither Σ nor Π. We here consider \$ as an ordinary alphabet, not as a parameter, as in the case of a p-suffix tree. Let sencode(S) denote the s-encoding of S. The s-strings S and S' are an s-match if and only if sencode(S) = sencode(S'). Let ssuffix$_i$(S) = sencode(S[i..n]), and let ssuffix$_i$(S)[j] be the jth character of ssuffix$_i$(S). Notice that

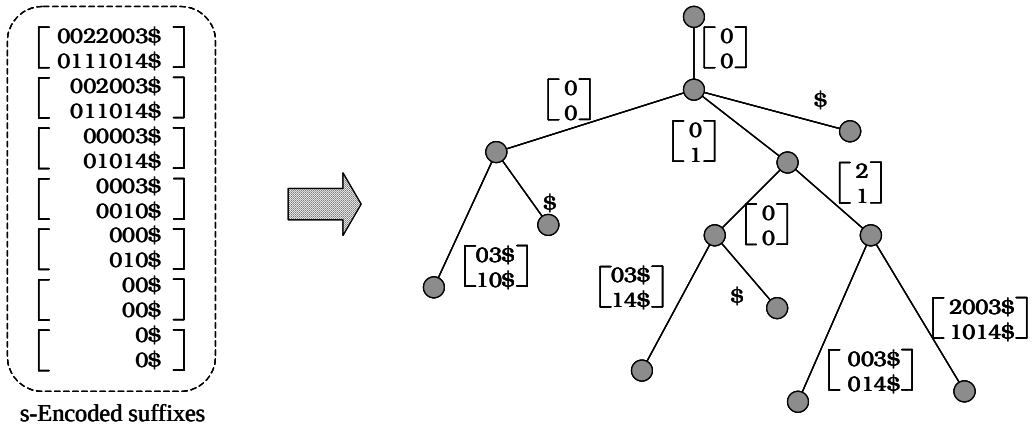


Figure 6.2: The s-Suffix Tree for RNA sequence '`AUAUCGU`.'

$\text{ssufix}_i(S)[j]$ is sometimes different from $\text{sencode}(S)[i+j-1]$: if $|\text{sencode}(S)[i+j-1]| > j$, $\text{ssuffix}_i(S)[j] = 0 \neq |\text{sencode}(S)[i+j-1]| > j$. Notice also that, if we have $\text{prev}(S)$ and $\text{compl}(S)$, we can obtain the value of $\text{ssuffix}_i(S)[j]$ for any i and j in a constant time.

Figure 6.2 shows an example of a s-suffix tree in the case of RNA structural matching. Using the s-suffix tree, we can find some set of substrings in some given sequence that s-match each other in $O(n)$ time or query a substring that matches another given string in $O(n \log(|\Sigma| + |\Pi|))$ time. We can also find common s-substrings of given two sequences in a linear time.

6.1.3 Algorithm for Constructing s-Suffix Trees

We first describe a basic method for constructing the s-suffix tree based on Ukkonen's algorithm.

The implicit s-suffix tree of S is the compacted trie of all the s-encoded suffixes of S, and a label for an edge that ends at a leaf is represented by only the first index of the label in it. Let T_i denote the implicit s-suffix tree of $S^+[1..i]$ for an integer i $(0 < i \leq |S|+1)$. Let $node(S)$ denote the node with label of s-encoded string of S in this section. Like Ukkonen's algorithm, our basic algorithm consists of $n+1$ phases, and in the ith phase, we construct an implicit s-suffix tree T_i from T_{i-1}.

As in Ukkonen's algorithm, we construct a new node $u = node(S[j..i])$ for all $1 \leq j \leq i$ in this order if there is no locus for $S[j..i]$ in the tree in the ith phase. We call this procedure for single j the jth extension as in the description of Ukkonen's algorithm. Ukkonen's algorithm speeds up each phase by ignoring unnecessary extensions. In this s-suffix tree case, the unnecessary extensions are all the same as Ukkonen's case.

The major problem in constructing s-suffix trees by applying Ukkonen's algorithm is that, as in Baker's p-suffix tree construction algorithm, a node of an s-suffix tree does not always have explicit suffix links to another node. Consider a node $u = node(c\alpha)$ in an s-suffix tree, where c is a single parameter in Π and α is some s-string. It is possible that the locus for α is not a node but a point on an edge. In this case, we let u's suffix link $sl(u)$ be this edge and call such a link an implicit suffix link.

The implicit suffix links cause two problems. One is how to keep these implicit suffix links correct

thorough the algorithm: the implicit suffix links must be updated if the corresponding edge is split. The other is how to analyze the number of scanned nodes in the algorithm. First, we deal with the former problem, and after that we discuss the latter problem.

It is easy to see the following lemma related to implicit suffix links:

Lemma 2 *Let u be a node with an implicit suffix link and $d = |\sigma_u|$. Then the first s-encoded character of the label of any of the outgoing edges from u must be one of d, 0, and $-d$. Furthermore, if it is d, its corresponding compl value must be* 0.

We use the term 'zero-node' for a node with more than one outgoing edge that has a label starting with either of d, 0, or $-d$, where d is the label length of the node, regardless of whether its suffix link is implicit or not. We also call edges the first s-encoded character of whose labels are d, 0 and $-d$, a "positive zero-edge", a "normal zero-edge" and a "negative zero-edge," respectively.

The following lemmas related to zero-nodes and zero-edges can be easily seen:

Lemma 3 *A positive zero-edge cannot be an ancestor of another positive zero-edge. Similarly, a negative zero-edge cannot be an ancestor of another negative zero-edge. There are at most $|\Pi|$ normal zero-edges on a path from the root to a leaf.*

Lemma 4 *On a path from the root to a leaf, there are at most $|\Pi| + 1$ zero-nodes.*

It is easy to find the implicit suffix links of newly constructed nodes in the algorithm, as in Ukkonen's algorithm. Consider the situation in the jth extension of the ith phase of the algorithm, when $u_j = node(S^+[j..i])$ is constructed. Note that $u'_j = node(S^+[j..i-1])$ may be constructed at the same time if necessary. As in the case of constructing an ordinary suffix tree, we will check the locus of $S^+[j+1..i-1]$ in the next extension in the same phase, so we will soon find the suffix links of u_j and u'_j. Hence we can conclude that every node other than the last leaf inserted and its parent has either an ordinary suffix link or an implicit suffix link. The problem is how to maintain these implicit suffix links. In the algorithm, we often split edges to add a new node. Thus we have to update each implicit link if the edge it links to is split into two edges by inserting a new node. We call a set of nodes a zero-chain if the nodes form a chain on a path from the root to a leaf in the tree and all the edges between them are same kind zero-edges, (*i.e.*, if one edge is a normal zero-edge, then the others are also normal zero-edges, for example).

We obtain the following theorem related to implicit suffix links:

Theorem 7 *For any edge e, the set of nodes having implicit suffix links to e forms at most $2|\Pi|$ zero-chains in the tree. Furthermore, the length of each zero-chain is at most $|\Pi|$.*

Proof: Let v and v' be two nodes with implicit suffix links to the same edge. If v is an ancestor of v' and there is a node u between v and v', it is obvious that $sl(u)$ is also between $sl(v)$ and $sl(v')$.

If neither of these two nodes is an ancestor of the other, let w be the lowest common ancestor of v and v' in the suffix tree. Note that w is not the root, because both of the first s-encoded characters of the labels of v and v' must be 0. Let $\text{suffix}_i(S)$ denote $S[i..|S|]$. Since one of the s-encoded strings of $\text{suffix}_2(\sigma_v)$ and

$\mathrm{suffix}_2(\sigma_{v'})$ must be a prefix of the other, the outgoing edges to v and v' must be zero-edges. Thus w must be a zero-node.

Lemma 3 implies that, under the negative or positive zero-edge out of w, there is only one zero-chain formed by the set of nodes having implicit suffix links to e. Furthermore, there are at most $|\Pi|$ normal zero-edges on a path to a leaf from the root, according to Lemma 3. One zero-node can have three outgoing zero-edges at most. Two of the edges can have only one such zero-chain under each edge, and the other node have at most $|\Pi| - 1$ zero-node under the edge. This means that there are at most $2|\Pi|$ such zero-chains. Also according to Lemma 4, it is obvious that the lengths of the zero-chains are at most $|\Pi|$. □

Note that, in the case of the p-suffix tree (*i.e.*, when there are no complementary character pairs), such nodes form only one zero-chain. According to this theorem, there are at most $2|\Pi|^2$ implicit suffix links to one edge. Hence when we split an edge, it takes $O(|\Pi|^2)$ time to update all the corresponding implicit suffix links if we do it naively, which causes a problem if $|\Pi|$ is large. From now on, we consider how to reduce the updating time to $O(\log |\Pi|)$.

Consider $O(n)$ sets of nodes which are empty at first. We maintain nodes which have implicit suffix links to a same edge in a same set. We perform two types of procedures for the sets. One is inserting a node into one of the sets, and the other is splitting one of the sets into two sets according to the label lengths of the nodes in the set, as follows: One of the two sets newly constructed by splitting is the set of nodes whose label lengths are larger than a specified length, and the other is the set of nodes whose label lengths are smaller than the same specified length. The upper bound of the size of a set is $p = O(|\Pi|^2)$, and that the total number of nodes inserted is $O(n)$. Using balanced tree structures like concatenable queues [1], or c-queues for short, we can achieve the following time bounds for these two procedures:

1. A new item can be inserted into a set in $O(\log p)$ time.
2. S can be split into two sets as above in a time linear to the size of the smaller set of them. More precisely, it takes $O(\log p)$ time to split the c-queue, and requires a time linear to the size of the smaller set of them to update their implicit suffix links.

It is easy to see that the total time taken by procedure 1 is $O(n \log p)$, and that the total time taken by procedure 2 is also $O(n \log p)$. Thus we can maintain the implicit suffix links by this procedure in $O(n \log |\Pi|)$ time, because $p = O(|\Pi|^2)$.

From now on, we discuss the number of nodes scanned in the algorithm. Note that it is $O(n)$ in Ukkonen's algorithm [73, 155]. Note also that the analysis for it is almost the same as that of the number of rescanned nodes in Baker's algorithm for p-suffix trees [18, 19].

Theorem 8 *The number of scanned edges that are not normal zero-edges is at most n.*

Proof: In constructing the s-suffix tree of a string S, consider that an edge (u, v) $(u = \mathit{parent}(v))$ that is not a normal zero-edge is scanned when we search for the locus of $S^+[j..i]$ in the jth extension of the $(i+1)$th phase. Let $u = \mathit{node}(S^+[j..k-1])$.

Consider the locus of $S^+[j'..k-1]$ for any $j' < j$. Note that we do not perform the j''th $(j'' \geq j)$ extension in the i'th phase $(i' < i)$ of the algorithm. If there exists a node (w) for the locus, then it must

have an explicit suffix link to some node, because $\mathrm{ssuffix}_{j'}(S^+[1..k])[k-j'+1]$ is not 0. This means that w cannot be scanned in the algorithm. Accordingly, we conclude that the total number of such edges is at most n. □

According to Lemma 4, the number of normal zero-edges that are scanned in a single phase is at most $|\Pi|$. Thus the total number of nodes that have implicit suffix links and are scanned in the algorithm is at most $n|\Pi|$. Note that the outgoing normal zero-edge from a node can be accessed in $O(1)$ time. Thus the total scanning time will be $O(n(|\Pi| + \log|\Sigma|))$.

Thus we conclude that the total computing time of our algorithm is $O(n(|\Pi| + \log|\Sigma|))$. If $|\Pi|$ and $|\Sigma|$ are constant, it is $O(n)$. In fact, in the problem of RNA/DNA structural matching ($|\Sigma| = 0$ and $|\Pi| = 4$), this basic algorithm is efficient enough.

Finally, we improve the algorithm to $O(n(\log|\Pi| + \log|\Sigma|))$. Note that our algorithm is simpler than Kosaraju's best-known algorithm [95]. A maximal zero-chains is a set of nodes in the zero-chain will not be a zero-chain, if we add any other node in the tree. We maintain all the normal maximal zero-chains in the tree using c-queues. Let $ll(e)$ be the label length of the node at the end of the edge e. For each normal zero-chain, we maintain a c-queue using $ll(e)$ as the key of each edge e. This structure can be split in $O(\log|\Pi|)$ time when a new node is inserted among the chain, and a new zero-edge can be inserted into a chain in also $O(\log|\Pi|)$ time. Furthermore, there are $O(n)$ normal zero-chains at most. Hence total time for maintaining them is $O(n\log|\Pi|)$.

Consider the situation that we have just constructed a new node by splitting an edge (u, v) ($u = parent(v)$) by inserting a node t. In the next extension, we will scan a path from $sl(u)$ to $sl(v)$ to find the locus of $sl(t)$. Note that, in Ukkonen's algorithm, we do not have to maintain suffix links of leaves, but we maintain these suffix links of leaves to know $sl(v)$ even if v is a leaf. Thus we already have both suffix links $sl(u)$ and $sl(v)$ in any case. Now let e be a zero-edge encountered in scanning from $sl(u)$ to $sl(v)$. Let C be the set of zero-edges in the zero-chain which includes e. We can find the zero-edge e' in C nearest to a leaf such that $ll(e')$ is not larger than the label length of $sl(t)$ in $O(\log|\Pi|)$ time, using the above c-queue. According to [42], we can compute the lowest common ancestor (LCA) of two nodes of a suffix tree in a constant time even while we are constructing the tree. Thus we can find the LCA w of the edge e' and the node $sl(v)$ in a constant time. Then what we have to do is to start scanning from w, because w must be an ancestor of $sl(t)$ or $sl(t)$ itself.

We can easily obtain the following theorem from Theorem 8:

Theorem 9 *The number of encountered maximal normal zero-chains in scanning is at most* $n + 1$.

Proof: There must be at least one edge that is not a normal zero-edge between two maximal normal zero-chains encountered in scanning, due to the definition of maximal zero-chains. Moreover, the number of scanned edges that are not normal zero-edges is at most n, according to Theorem 8. Thus we can conclude that the number of encountered normal zero-chains in scanning is at most $n + 1$. □

Thus the total time for scanning zero-edges is $O(n\log|\Pi|)$. Now, we conclude that the total computing time of our algorithm is $O(n(\log|\Pi| + \log|\Sigma|))$.

Table 6.1: Number of nodes in suffix trees and s-suffix trees.

Sequence	(A)	(B)	(C)	(D)	(E)	(F)	(G)
Length	9719	9748	8981	1000000	1000000	1000000	1000000
Suffix Tree	16135	16217	14710	1640492	1635995	1638043	1638008
s-Suffix Tree	16033	16132	14666	1631525	1628821	1630104	1628923

6.1.4 Computational Experiments

Using the s-suffix tree of a string we can perform tasks such as the following:

- Given a long sequence of length n and some constant l and r, we can find a set of more than r substrings that s-match with each other and are longer than l in an $O(n(\log|\Sigma| + \log|\Pi|) + T_{output})$ time, where T_{output} is the output size.
- Given more than one sequence, we can find the longest common s-encoded pattern of these sequences in $O(n(\log|\Sigma| + \log|\Pi|) + T_{output})$ time, where T_{output} is the output size and n is the sum of the lengths of the input sequences.

Note that, if the size of the alphabet is constant, both of these tasks can be completed in a linear time. In this section, we describe experiments on RNA and DNA sequences, in which we constructed the s-suffix tree of DNA sequences, where $\Sigma = \phi$, $\Pi = \{$`A`,`U`,`G`,`C`$\}$, `A` is the complement of `U` and `G` is the complement of `C`. (In DNA sequences, `T` is present instead of `U`.)

We conducted experiments on three HIV (human immunodeficiency virus) RNA complete sequences: (A) a sequence of length 9719 (accession number: K03455), (B) a sequence of length 9748 (accession number: X01762) and (C) a sequence of length 8981 (accession number: AF067156). We also use four very long DNA sequences of *E. coli*, each of which has the same length, 1 Mbp = 1,000,000 bp. The length of the full genome sequence of *E. coli* is about 4.64 Mbp, and these four sequences are the following regions of the sequence: (D) 1 bp–1,000,000 bp, (E) 1,000,001 bp–2,000,000 bp (F) 2,000,001 bp–3,000,000 bp, and (G) 3,000,001 bp–4,000,000 bp.

First, we compare the size of the s-suffix tree with that of the normal suffix tree of the same sequences. Table 6.1 shows the numbers of nodes in the suffix trees and the s-suffix trees of the seven sequences. According to the table, the sizes of the s-suffix trees are slightly smaller than those of the normal suffix trees in all cases, but the numbers of nodes in them are almost the same regardless of the length of the sequence. For any sequence, both the number of nodes in the suffix tree and that of the s-suffix tree are about 1.6 to 1.7 times the length of the sequence. Thus we can say that the s-suffix trees are very compact and that it is as reasonable to build them as to build the normal suffix trees.

Consider that a structural pattern α of length l appears r times, but any pattern that is constructed by extending it to the right, such as αc ($c \in (\Sigma \cup \Pi)$) appears less than r times. We call such a pattern α a "maximal structural pattern." We now give the experimental results of an experiment to find maximal structural patterns which are longer than l and repeated more than r times for some given l and r. Table 6.2

Table 6.2: Examples of maximal structural pattern.

(1)

Position	Sequence
646095	CCCGCTTCGGCTTCA
703617	GGGCGTTGCCGTTGA
779110	TTTATGGTAATGGTC
888469	TTTATCCTAATCCTG

(2)

Position	Sequence
371484	ACTGCGCCATGAAGATGAC
884639	GACTATAAGCTGGTGCTGA

Table 6.3: Number of structural/normal patterns.

(1) HIV RNA sequences

l	Pattern	(A)	(B)	(C)
≥ 5	Structural	5329	5061	4887
	Normal	1381	1147	1000
≥ 10	Structural	670	451	282
	Normal	479	363	126
≥ 15	Structural	336	123	4
	Normal	336	123	3

(2) *E. coli* sequences

l	Pattern	(D)	(E)	(F)	(G)
≥ 10	Structural	495371	499205	498728	497701
	Normal	90968	85899	88681	90298
≥ 15	Structural	4723	4140	4466	4529
	Normal	2402	1728	2095	2147
≥ 20	Structural	330	106	192	192
	Normal	330	103	192	190

shows two examples of maximal structural patterns found in *E. coli* sequence (D): (1) is a set of patterns of length 15 that appears four times, and (2) is a set of patterns of length 19 that appears 2 times in the sequence. Every sequence is different from the others, but these sequences s-match with each other.

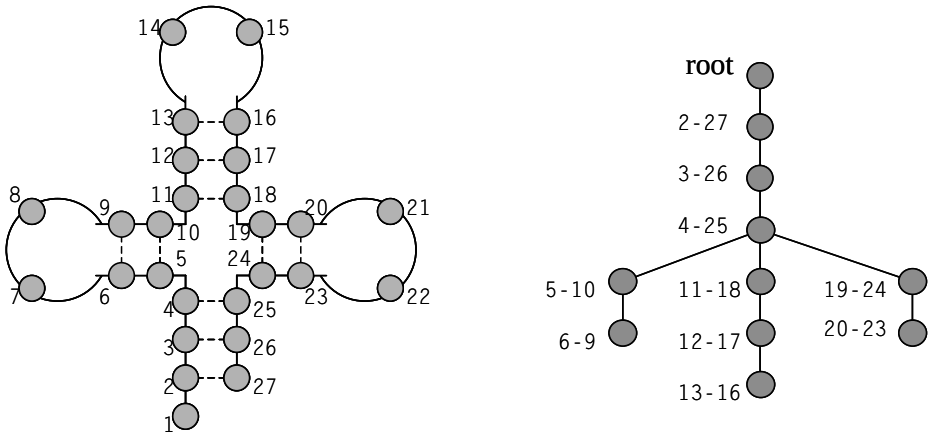
Table 6.3 shows the number of maximal patterns whose lengths (l's) are larger than some given length. In the table, a "normal pattern" means an ordinary string pattern that can be found with an ordinary suffix tree. Notice that the structural patterns include the normal patterns. According to the table, the proportion of normal patterns increases with the lengths of the patterns.

6.2 The Suffix Tree of a CS-Tree with a Large Alphabet

In this section, we describe a new algorithm for computing the suffix tree of a CS-tree which is useful for tree pattern matching.

6.2.1 Tree Representation of RNA Secondary Structures

A secondary structure of an RNA sequence $S[1..n]$ is a structure that satisfies the following condition: if $S[i]$ makes a Watson-Crick pair with $S[j]$ and $S[k]$ makes a pair with $S[l]$ with $i < k < j$, then $i < l < j$ also. Note that the graph that represents the sequence and the Watson-Crick pairs of a secondary structure can be displayed as a planar graph. Note that the structure shown in Figure 6.1 (2) is not a secondary structure



(1) A secondary structure. (2) Tree representation.

Figure 6.3: An Example of Tree Representation of an RNA Secondary Structure.

because it does not satisfy the condition above. It is known that a secondary structure can be represented by a tree data structure [160]. In the tree representation, each node of the tree represents a Watson-Crick pair, except for its root. If Watson-Crick pair a is between pair b, then we let a be a descendant of b. If there is no pair that includes pair a, then we let a be a child of the root. Figure 6.3 shows an example of an RNA secondary structure and its tree representation. Hence analyses of patterns of a tree structure are very important for analyzing mRNA structures. In the rest of this section and in the next section, we will deal with suffix tree-based data structures that are useful for tree pattern matching, *e.g.* we can efficiently find frequent patterns in a set of trees.

6.2.2 Algorithm Outline and Preliminaries

Our approach to constructing the suffix tree of a CS-tree is based on Farach's suffix tree construction algorithm [50]. Farach's algorithm has three steps. First, it constructs a tree called an odd tree recursively. Next, it constructs another tree called an even tree by using the odd tree. Finally it constructs the suffix tree by merging these two trees. Note that the odd tree is a trie of suffixes $S[2i-1]\dots S[n]\$$, and the even tree is a trie of suffixes $S[2i]\dots S[n]\$$. This algorithm achieves an $O(n)$ computation time for integer alphabets.

We later also define the odd and even trees for the suffix tree of a CS-tree, and our algorithm also has three following similar steps. First we build the odd tree or the even tree recursively, then we construct the even or odd tree by using the odd or even tree, respectively, and finally we merge them to construct the suffix tree.

In our algorithm, we use the following theorem by Dietz and several others [3, 26, 47]:

Theorem 10 *In any tree with n nodes, for any node v in the tree and any integer $d > 0$ that is smaller than the depth of v, we can find the ancestor of v whose depth is d in constant time after $O(n)$ preprocessing.*

Let us now define several notations. Let $\{S_1, \dots, S_k\}$ be the strings represented by a given CS-tree. Let n_i be the length of S_i and let $S_i = S_i[n_i]\dots S_i[1]$. Note that the indices are arranged in reverse order. The above

theorem 10 indicates that, for any i and j, we can access $S_i[j]$ in constant time after $O(n)$ preprocessing. Let $S_i(m)$ be S_i's suffix of length m, *i.e.*, $S_i[m] \dots S_i[1]$. Let $lcp(S, S')$ and $lcs(S, S')$ be the lengths of the longest common prefix and suffix of strings S and S', respectively. Let $parent_U(v)$ be the parent node of v in the CS-tree U if v is not the root node t; otherwise, let it be t: *i.e.*, $parent_U(v_{i,j}) = v_{i,\max(0,j-1)}$, where $v_{i,j}$ denotes the ancestor of v_i whose depth is j. Let $label(e)$ be the label given to edge e in the CS-tree. Let T_U be the suffix tree of the CS-tree U.

6.2.3 Building a Half of the Suffix Tree Recursively

All nodes in the CS-tree $U = (V, E)$ have either odd or even label length. Let V_{odd} and V_{even} be the nodes with odd label lengths and those with even label lengths, respectively. If $|V_{odd}| \geq |V_{even}|$, let $V_{small} = V_{even}$ and $V_{large} = V_{odd}$; otherwise, let $V_{small} = V_{odd}$ and $V_{large} = V_{even}$. We can obtain $|V_{odd}|$ and $|V_{even}|$ in $O(n)$ time by the ordinary depth-first search on the CS-tree. Therefore, we can determine in a linear time which node set is V_{small}. In this subsection, we will recursively construct the compacted trie T_{small} of all the labels of nodes in V_{small}. Note that the technique for constructing T_{small} is very similar to that for constructing the odd tree in Farach's algorithm.

Consider a new CS-tree $U' = (V_{small}, E_{small})$, where $E_{small} = \{(v, parent_{U'} = parent_U(parent_U(v)))|v \in V_{small}, v \neq t\}$ and the edge labels are determined as follows. Radix sort the label pairs $pair(v) = (label((v, parent_U(v))), label((parent_U(v), parent_U(parent_U(v)))))$ for all $v \in V_{small} \backslash t$ and remove duplicates, where $label(e)$ denotes the label of an edge e in the original CS-tree U. Let $rank(v)$ be the rank of $pair(v)$ in the sorted list, which belongs to an integer alphabet $[1, n/2]$ because the size of the new tree U' is not larger than half of that of the original CS-tree U. Let $orig\_pair(i)$ be a label pair $pair(v)$ such that $rank(v) = i$. Let the label of an edge $(v, parent_{U'}(v)) \in E_{small}$ be $rank(v)$. Notice that all of these procedures can be performed in a linear time.

We then construct the suffix tree $T_{U'}$ of U' by using our algorithm recursively. After that, we construct T_{small} from $T_{U'}$ as follows. We can consider a tree T' whose edge labels of $T_{U'}$ are modified to the original labels in U: for example, if the label of an edge in $T_{U'}$ is ijk, the label of the corresponding edge in T' is $orig\_pair(i), orig\_pair(j), orig\_pair(k)$. Notice that this modification can be performed by making only a minor modification of the edge label representation and that it takes only linear time.

We can construct T_{small} from T' very easily. T' contains all the labels of nodes in V_{small}, but is not the compacted trie: the first characters of labels of outgoing edges from the same node may be the same. But the second character is different, and the edges are sorted lexicographically. Thus we can change T' to T_{small} by making only a minor adjustment: we merge such edges and make a node, and if all the first characters of all the labels of edges are the same, we delete the original node.

In this way, we can construct T_{small} in a $T(n/2) + O(n)$ time, where $T(n)$ is the time our algorithm takes to build the suffix tree of a CS-tree of size n.

6.2.4 Building the Other Half of the Tree

In this section, we show how to construct the compacted trie T_{large} of all the labels of nodes in V_{large} from T_{small} in a linear time. The technique is a slightly modified form of the second step of Farach's algorithm,

which constructs the even tree from the odd tree.

If we are given an lexicographic traverse of the leaves of the compacted trie (which is called lex-ordering in [50]), and the length of the longest common prefix of adjacent leaves, we can reconstruct the trie [50, 51]. We will obtain these two parts of T_{large} from T_{small}, and construct T_{large} in the same way. This method can obtain the label length from the leaf or root for each node of the compacted trie. We can obtain that node from its specified depth and its some descendant leaf in constant time according to Theorem 10. Hence the total time required by this procedure is $O(n)$.

Any leaf in T_{large}, except for those with labels of only one character, has a label consisting of a single character followed by the label of some corresponding leaf in T_{small}. We can obtain the lex-ordering of the labels of leaves in T_{small} by an in-order traverse of T_{small} which takes only a linear time. Thus we can obtain the lex-ordering of the labels of leaves $(S_i(m))$ in T_{large} by using the radix sorting technique, because we have $S_i[m]$ and the lexicographically sorted list of $S_i(m-1)$.

The longest common prefix length of adjacent leaves of T_{large} can also be obtained easily by using T_{small}. Let $S_i(m)$ and $S_j(n)$ be the labels of two adjacent leaves in T_{large}. If $S_i[m] \neq S_j[n]$, the longest common prefix length is 0. Otherwise, it is $1 + lcp(S_i(m-1), S_j(n-1))$ which can be obtained in a constant time from T_{small} after linear-time preprocessing on T_{small} (see Theorem 4). In this way, we can construct T_{large} from T_{small} in $O(n)$.

6.2.5 Merging the Trees

Now we have two compacted tries T_{odd} and T_{even}. In this subsection, we merge T_{odd} and T_{even} to construct the target suffix tree T_U. We call the compacted trie of odd/even-length suffixes of strings the generalized odd/even tree of the strings. The odd/even tree of a CS-tree is also the generalized odd/even tree of the strings represented by the CS-tree. Farach's algorithm merges the odd and even trees in a time linear to the sum of the sizes of odd and even trees. It can be applied also to our problem of merging generalized odd and even trees and we can also achieve $O(n)$ time in our case. The outline of the algorithm is as follows.

First, we merge the even and odd trees as following by treating one of two edge labels as a prefix of the other label if the first characters of labels of two edges are the same. Let edges $e_1 = (v, v_1)$ and $e_2 = (v, v_2)$ be the edges which start from the same node v and have the same first character. Let l_1 and l_2 be the label lengths of e_1 and e_2, respectively. Without loss of generality, we let $l_1 \geq l_2$. Then we construct a internal node v_1' between v and v_1 if $l_1 > l_2$, otherwise let v_1' be v_1. In case that $l_1 > l_2$, let the label of edge (v, v_1') be the first l_2 characters of the label of original edge (v, v_1) and let the label of edge (v_1', v_1) be the last $l_1 - l_2$ characters of the label of original edge (v, v_1). Then we merge two edges (v, v_1') and e_2. Note that this merging requires only constant time because we can find the node of the CS-tree which corresponds to the first character of new edge (v_1', v_1) in a constant time. We merge recursively all over the two trees by the normal coupled depth first search. Thus the total computing time required for the merging is also $O(n)$.

Next, we unmerge the edges with different labels because we have merged edges too far. Farach showed that we can unmerge correctly in a linear time [50] for the problem of strings, which is also the case for our problem.

Consider a node u in the merged tree M, and let $v' \in T_{odd}$ and $w' \in T_{even}$ be some nodes that become

u's descendants, v and w respectively, in M. Let $L(u)$ be the length of the label of u, and $\hat{L}(u)$ be the longest common prefix length of the labels of v and w. We can see that u is merged too far if $L(u) > \hat{L}(u)$. $L(u)$ can be consulted in T_{odd} or T_{even}, and $\hat{L}(u)$ can be computed for all the nodes in $O(n)$ time by the Farach's technique which uses a data structure called d-links. We call u a border node if $\hat{L}(u) < L(u)$ and $\hat{L}(p) = L(p)$ where p is the parent of u. All the border nodes can be found in a linear time. We can correct M by only unmerging the border nodes. For each unmerged edge, we must find the node of the CS-tree that corresponds to the first character of its label, which requires only a constant time according to Theorem 10. Thus the total computing time for unmerging is $O(n)$.

Hence the step of our algorithm for merging trees takes a total of $O(n)$ time. Thus we obtain an equation $T(n) = T(n/2) + O(n)$, where $T(n)$ is the time needed to construct the suffix tree of a CS-tree of size n. Therefore, our algorithm achieves the optimal $T(n) = O(n)$ computing time for general CS-trees with integer alphabets.

6.3 The BSuffix Tree

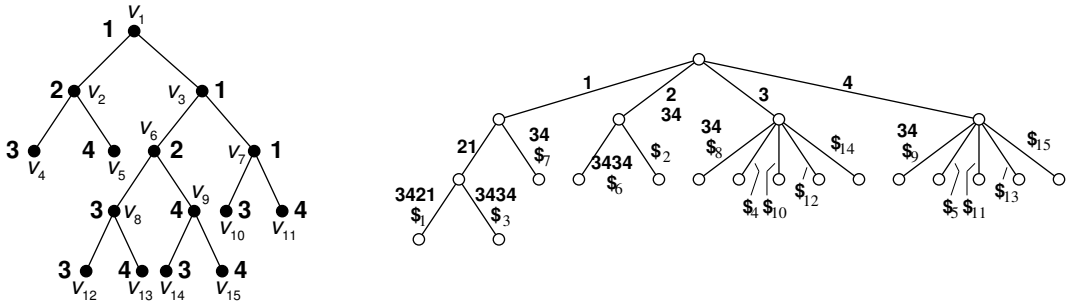
In this section, we propose a new data structure, the Bsuffix tree, which enables efficient queries of completely balanced binary trees from any binary forest (including a single tree). It can also be used for querying completely balanced k-ary subtrees from any k-ary forest (k need not be constant in this case), but we will deal with binary trees at first. The Bsuffix tree is a data structure for matching of nodes, but it can be also used for matching of edges (see subsection 6.3.3).

6.3.1 Definition of the BSuffix Tree

Consider a completely balanced binary tree P of height h. Let $p_1, p_2, \ldots, p_{2^h-1}$ be the nodes of P in breadth-first order, and let $c_i \in \{1, \ldots, n\}$ be the alphabet given for node p_i. Note that $p_{\lfloor i/2 \rfloor}$ is the parent of p_i in this order. We call $c_1 c_2 \cdots c_{2^h-1}$ the label of P. We call substring $c_{2^i} \cdots c_{2^{i+1}-1}$ of this label a Bcharacter. Furthermore, we call a string of Bcharacters a Bstring. For Bstring $b_1 b_2 \ldots b_n$, we call $b_1 b_2 \ldots b_m (m < n)$ a Bprefix of the Bstring. Note that $c_1 c_2 \cdots c_{2^h-1}$ is a Bstring of length h. For two Bcharacters b_1 and b_2, we let $b_1 > b_2$ if b_1 is lexicographically larger than b_2 in the normal string representation. Note that Bcharacter $b = c_{2^i} \cdots c_{2^{i+1}-1}$ can be represented by node $p_{2^i} \in P$ and integer i.

Consider a binary forest U of size n whose nodes are labeled with a character of an integer alphabet $\{1, \ldots, n\}$. Let $v_1, v_2, \ldots, v_n$ be the concatenated list of the breadth-first-ordered node lists of all the binary trees in forest U, and let $a_i \in \{1, \ldots, n\}$ be the label of node v_i. Let L_i be the label of the largest completely balanced binary subtree of U whose root is node v_i. We call L_i followed by $\$_i \notin \{1, \ldots, n\}$ ($\$_i \neq \$_j$) the Blabel of node v_i. If the roots of two completely balanced binary subtrees P_1 and P_2 of U are the same node and P_1 includes P_2, the label of P_2 is a Bprefix of the label of P_1. The Bsuffix tree of U is the compacted trie T of the Blabels of all the nodes in U in the Bstring sense, *i.e.*, the outgoing edges from some node in the suffix tree have a label of different Bcharacter. Figure 6.4 shows an example of a Bsuffix tree. By using T, we can very easily query any completely balanced binary subtree of U.

Edge labels of T can be represented by the first node in U and the depths of the first and the last nodes



(1) Binary tree U (2) Bsuffix tree for U

Figure 6.4: An example of the Bsuffix tree.

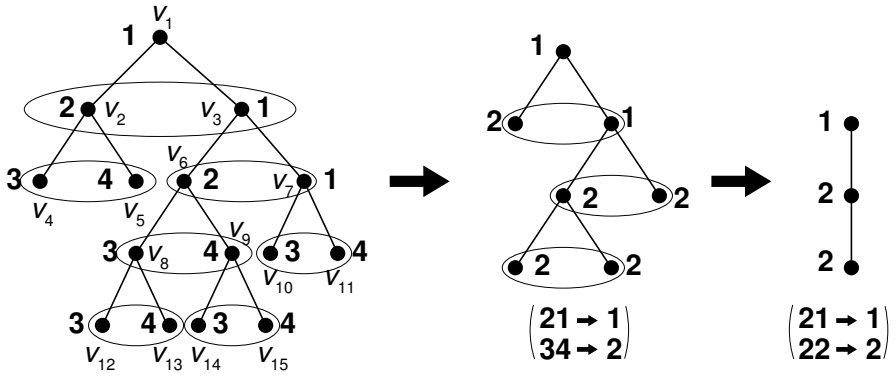


Figure 6.5: Recursive construction of new binary trees in computing Bsuffix tree.

in the corresponding subtree pattern. Therefore T can be stored in $O(n)$ space. Note that we can access any member of the edge label of T in a constant time if we have both the breadth-first list and the depth-first list of the nodes of each tree in forest U.

In a Bsuffix tree, to enable fast access to a node's outgoing edge whose first Bcharacter of its label is given, two simple preprocessing can be considered. One simple method is constructing a hash table for it, which enables linear time query in average. The data structure for it can be built in linear time. The other method is constructing a prefix tree to represent all of the first Bcharacters of edge labels, which enables deterministic $O(m \log |\Sigma|)$ query time for a query of size m, where $|\Sigma|$ denotes the size of the alphabet. This data structure can also be built in linear time.

6.3.2 Construction of the BSuffix Tree

In this subsection, we describe the $O(n)$ algorithm for constructing the Bsuffix tree T of U.

If forest U consists of only nodes with less than two children, it is obvious that we can construct the Bsuffix tree of U in $O(n)$ time. Otherwise, we first construct a new binary forest U' as follows: For every node

v_i with two children v_j, v_{j+1}, construct a node of U' (let it be w_i). If v_j and/or v_{j+1} have two children, let w_i be the parent of w_j and/or w_{j+1} in forest U'. Radix sort the label pairs (a_j, a_{j+1}) and remove duplicates. Let the label a'_j of w_j be the rank of the label pair (a_j, a_{j+1}) in the sorted list. Notice that the number of nodes in U' is not larger than $n/2$. We construct the Bsuffix tree T' of U' by using our algorithm recursively. Figure 6.5 shows an example of this recursive construction of new binary forests (trees in this case). Next, we construct T from T'.

If we are given the lexicographically sorted list of the Blabels of all the nodes in U and the length (*i.e.*, number of Bcharacters) of the longest common Bprefix of adjacent Blabels in this list, we can construct Bsuffix tree T in a linear time. We obtain these two pieces of information from T'.

Notice that the in-order traverse of leaves of T' corresponds to a lexicographically sorted list of all the first-character-deleted Blabels of nodes that have two children in U. Thus we can obtain the lexicographically sorted list of all the node Blabels of U by radix sorting the concatenated list of the in-order traverse of leaves of T' and the Blabels of nodes with no or only one child.

The longest common Bprefix length l of adjacent Blabels can also be obtained from T'. If the first characters of two adjacent Blabels are different, $l = 0$. Otherwise, if one of the adjacent Blabels consists of only one character, the depth is $l = 1$. Otherwise, we compute the depth as follows. Let v_i and v_j be the adjacent nodes. Notice that we can obtain the longest common Bprefix length l' of Blabels of w_i and w_j in U' in a constant time (see Theorem 4). Then it is clear that $l = l' + 1$.

In this way we can construct T from T' in a linear time. We obtain $T(n) = T(n/2) + O(n)$, where $T(n)$ denotes the time taken to compute the Bsuffix tree of a binary tree of size n. Therefore we conclude that our algorithm runs in $O(n)$ time.

6.3.3 Discussions on the Bsuffix Tree

Bsuffix trees are very similar to normal suffix trees. It enables efficient query for a completely balanced binary tree pattern. It can also be used for finding (largest) common completely balanced binary subtrees of two binary trees in linear time. We can also enumerate frequent patterns of completely balanced binary trees in linear time by using this data structure.

The data structure and our algorithm assume that the labels are given to nodes, but they can very easily be modified to deal with edge-matching problems as follows: Let the label of any node except for the root be the label of the incoming edge from its parent. Then T' in the above algorithm can be used as the compacted trie for edge matching.

Bsuffix trees can also be used for querying completely balanced k-ary trees from any k-ary forest U. First, if a node has less than k children, remove the edges between it and its children. Otherwise, we reconstruct each node that has k children as a completely balanced binary tree of depth $\lceil \log_2 k \rceil$ and move each child to its leaf. For each inside node and leaf to which no node was mapped, give as its label a new character that is not in use but is same for all such nodes. Notice that the size of the reconstructed forest is at most twice as that of the original one. Then construct the Bsuffix tree for this reconstructed binary tree. It can obviously used for querying completely balanced k-ary trees.

6.4 Summary

We have proposed a new data structure called the structural suffix tree, or s-suffix tree for short. We also proposed an on-line $O(n(\log|\Sigma| + \log|\Pi|))$ algorithm for constructing it, where Σ is an alphabet of fixed symbols and Π is an alphabet of parameters. This data structure enables an efficient search for frequent patterns of structures of RNA sequences or single-stranded DNA sequences. It also enables a common structure pattern to be efficiently found in more than one sequence. We also showed the practicality of our data structure and our algorithm by reporting computational experiments for finding structural patterns from RNA sequences of HIV and DNA sequences of *E. coli* using the s-suffix tree.

We also have described an optimal $O(n)$ algorithm for constructing the suffix tree of a common suffix tree (CS-tree). In addition, we proposed a new data structure called a Bsuffix tree, that enables efficient query for completely balanced subtrees.

Several tasks remain for the future. Two RNA sequences can have the same structure even if they do not have the same s-encoded string patterns. Furthermore, it is difficult to apply our algorithm to the problem of proteins, where the combinations are far more complicated. Thus we should strive to create more general data structures and algorithms for structural pattern matching of biological sequences. Recently, Farach [50] introduced a deterministic linear-time suffix tree construction algorithm for strings of an integer alphabet $\{1, \ldots, n\}$. It is an open problem whether or not such a linear time algorithm exists for constructing s-suffix trees or p-suffix trees. The existence of more useful suffix trees that allow querying more general and flexible patterns than paths or completely balanced trees remains as another open question.

Chapter 7

Concluding Remarks

In this thesis, we have shown that efficient pattern matching or indexing techniques are very important in many applications of molecular biology by demonstrating various useful pattern matching-based algorithms.

First, we dealt with the multiple alignment problem. We proposed two flexible and efficient approaches for providing alternative solutions to the computationally optimal solution of the alignment problem: One is an efficient method for enumerating suboptimal solutions, and the other is parametric analysis of the problem. For the first approach, we discussed what kind of suboptimal alignment is unnecessary to enumerate and proposed an efficient algorithm that enumerates only the necessary alignments based on Eppstein's algorithm. For the other approach, we proposed several techniques for searching the parametric space of the problem to find desired solutions. For both approaches, this thesis performed experiments on various groups of actual protein sequences and examined the efficiency of these algorithms and property of sequence groups.

Next, we dealt with the problem of cDNA clustering. We proposed efficient techniques that enables accurate querying and clustering for cDNA sequences, considering splicing mechanisms. Our algorithm is based on all-pairs comparison by a variant of Mott's spliced alignment algorithm, but it takes too much time if we apply it naively. Thus we proposed several techniques for reducing the total computation time. According to the experiments, we achieved about 4,000 to 20,000-fold speedup against a naive algorithm. Moreover, heuristic version of our algorithm is much faster though the accuracy is almost the same as our exact algorithm.

We next dealt with prokaryotic gene finding problem, one of the most important data mining problems in computational molecular biology. Our algorithm is based on large-scale pattern matching using a large pattern database called the Bio-Dictionary. We developed a very efficient pattern indexing algorithm for the Bio-Dictionary. We then proposed a scoring scheme that uses matching patterns discovered from the Bio-Dictionary for ranking gene candidates to identify genes. We demonstrated our algorithm through a series of carefully designed experiments over many organisms to show the very high accuracy of our approach.

Finally, we proposed several useful data structures and algorithms for RNA structural analyses. In this work, we dealt with three kinds of generalization of suffix trees, two of which we proposed. We first proposed a generalization of a parameterized suffix tree (p-suffix tree for short) and an on-line algorithm for building it. It is also the first on-line algorithm for building the p-suffix tree, though the time complexity is same as the best-known algorithm that is not on-line. We can find efficiently a common structural pattern of given

RNA sequences with this data structure. We next proposed a new algorithm for constructing the suffix tree of a common suffix tree, which is useful for data mining from a set of tree structures. Considering that mRNA secondary structures can be described with trees, we can use this algorithm for the analyses of mRNA structures. Our algorithm improves the previous best-known time bound. In addition, we proposed a new data structure called the Bsuffix tree that is also useful for data mining from a set of trees, and proposed an optimal construction algorithm for it.

Through all these researches, we have shown that sequence comparison and indexing techniques play a very important role in the problems of computational molecular biology. Several tasks remain for the future. For many of the problems that we dealt with, we should develop much more flexible methods as we proposed in Chapter 3. The gene finding technique developed in Chapter 5 should be extended for analyzing eukaryotic genomes, for which the algorithms of Chapter 4 must be very useful for collecting training sets for learning the splicing mechanism. It is said that the RNA structures play some role in the splicing mechanism, and it is also very interesting if the algorithms in Chapter 6 could be used to improve a gene finding system for eukaryotic genomes.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Design and Analysis of Algorithms,* Addison Wesley Publishing Co., Reading, Mass., 1974.

[2] V. R. Akmaev, S. T. Kelley and G. D. Stormo. A phylogenetic approach to RNA structure prediction. *Proc. 7th International Conference on Intelligent Systems for Molecular Biology*, pp. 10–17, 1999.

[3] S. Alstrup and J. Holm. Improved Algorithms for Finding Level Ancestors in Dynamic Trees, *Proc. 27th International Colloquium on Automata, Languages, and Programming (ICALP 2000)*, LNCS 1853, pp. 73-84, 2000.

[4] S. F. Altschul. Amid acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.,* Vol. 219, pp. 555–565, 1991.

[5] S. F. Altschul, R. J. Carroll and D. J. Lipman. Weights for data related by a tree. *J. Mol. Biol.,* Vol. 207, pp. 647–653, 1989.

[6] S. F. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology,* Vol. 48, pp. 603–616, 1986.

[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, Vol. 215, pp. 403–410, 1990.

[8] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucl. Acids Res.* Vol. 25. pp. 3389–3402, 1997.

[9] A. Apostolico and Z. Galil, eds., *Pattern Matching Algorithms*, Oxford University Press, New York, 1997.

[10] S. Araki, M. Goshima, S. Mori, H. Nakashima, S. Tomita, Y. Akiyama and M. Kanehisa. Application of parallelized DP and A* algorithm to multiple sequence alignment. *Proc. Genome Informatics Workshop IV*, pp. 94–102, 1993.

[11] T. K. Attwood, D. R. Flower, A. P. Lewis, J. E. Mabey, S. R. Morgan, P. Scordis, J. N. Selley and W. Wright. PRINTS prepares for the new millennium. *Nucl. Acids Res.*, Vol. 27, No. 1, pp. 220–225, 1999.

[12] S. Audic and J-M Claverie. Self-identification of protein-coding regions in microbial genomes. *Proc. Natl. Acad. Sci. USA*, Vol. 95, pp. 10026–10031, 1998.

[13] J. H. Badger and G. J. Olsen. CIRITICA: coding region identification tool invoking comparative analysis. *Molecular Biology and Evolution*, Vol. 16, pp. 512–524, 1999.

[14] V. Bafna and D. H. Huson. The conserved exon method for gene finding. *Proc. International Conference on Intelligent Systems on Molecular Biology (ISMB'00)*, pp. 3–12. 2000.

[15] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucl. Acids Res.*, Vol. 28, No. 1, pp. 45–48. 2000.

[16] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, Interface Foundation of North America, pp. 49–57, 1992.

[17] B. S. Baker. Parameterized pattern matching by Boyer-Moore-type algorithms. *Proc. 6th Annual ACM-SIAM Symp. Discrete Algorithms,* pp. 541–550, 1995.

[18] B. S. Baker. Parameterized pattern matching: algorithms and applications. *J. Comp. Syst. Sci.,* Vol. 52, No. 1, pp. 28–42, 1996.

[19] B. S. Baker. Parameterized duplication in strings: algorithms and application to software maintenance. *SIAM J. Comput.,* Vol. 26, No. 5, pp. 1343–1362, 1997.

[20] B. S. Baker. Parameterized diff. *Proc. 10th ACM-SIAM Symp. Discrete Algorithms,* pp. 854–855, 1999.

[21] A. Barr and E. A. Feigenbaum. *Handbook of artificial intelligence*, William Kaufman, Inc., Los Altos, Calif, 1981.

[22] A. Bateman, E. Birney, R. Durbin, S. R. Eddy, K. L. Howe and E. L. Sonnhammer. The Pfam protein families database. *Nucl. Acids Res.*, Vol. 28, pp. 263–266, 2000.

[23] BDGF Website. *http://cbcsrv.watson.ibm.com/Tgi.html.*

[24] R. Beeza-Yates and G. Gonnet. A new approach to text searching. *Comm. ACM*, Vol. 35, pp. 74–82, 1992.

[25] M. A. Bender and M. Farach. The LCA problem revisited. *Proc. LATIN American Symposium*, LNCS 1776, 2000.

[26] O. Berkman and U. Vishkin. Finding Level-Ancestors in Trees. *J. Comp. Sys. Sci.*, Vol. 48, pp. 214-230, 1994.

[27] A. Bird. CpG islands as gene markers in the vertebrate nucleus. *Trends in Genetics*, Vol. 3, pp. 342–347, 1987.

[28] J. A. Blake, J. T. Eppig, J. E. Richardson, C. J. Bult and J. A. Kadin. The mouse genome database (MGD): integration nexus for the laboratory mouse. *Nucleic Acids Res.*, Vol. 29, No. 1, pp. 91–94, 2001.

[29] A. Blanck and D. Oesterhelt. The halo-opsin gene. II. sequence, primary structure of halorhodopsin and comparison with bacteriorhodopsin. *EMBO J.,* Vol. 6, pp. 265–273, 1987.

[30] M. S. Boguski and G. D. Schuler. Establishment of a transcript map. *Nat. Genet.*, Vol. 10, pp. 369–371, 1995.

[31] M. Borodovsky and J. McIninch. GeneMark: parallel gene recognition for both DNA strands. *Computers & Chemistry*, Vol. 17, No. 19, pp. 123–133, 1993.

[32] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, Vol. 191, pp. 131–144, 1998.

[33] C. Burge and S. Karlin. Finding the genes in genomic DNA. *Current Opinion in Structural Biology*, Vol. 8, pp. 346–354, 1998.

[34] J. Burke, D. Davison and W. Hide. d2_Cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Res.*, Vol. 9, pp. 1135–1142. 1999.

[35] M. Burset and R. Guigo. Evaluation of gene structure prediction programs. *Genomics*, Vol. 34, pp. 353–367, 1996.

[36] M. Cariaso, P. Folta, M. Wagner, T. Kuczmarski and G. Lennon. IMAGEne I: clustering and ranking of I.M.A.G.E. cDNA clones corresponding to known genes. *Bioinformatics*, Vol. 15, No. 12, pp. 965–973, 1999.

[37] K. M. Chao. Computing all suboptimal alignments in linear space. *Proc. 5th Symposium on Combinatorial Pattern Matching,* LNCS 807, pp. 31–42, 1994.

[38] L. Chen, M. Sato, H. Inoko and M. Kimura. Molecular cloning and analysis of novel cDNAs specifically expressed in adult mouse testes. *Biochem. Biophys. Res. Commun.*, Vol. 240, No. 2, pp. 261–268, 1998.

[39] M. T. Chen and J. Seiferas. Efficient and Elegant Subword Tree Construction. A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words, Chapter 12,* NATO ASI Series F: Computer and System Sciences, pp. 97–107, 1985.

[40] J. M. Claverie. Computational methods for the identification of genes in vertebrate genomic sequences. *Human Molecular Genetics*, Vol. 6, No. 10, pp. 1735–1744, 1997.

[41] J. M. Claverie. Computational methods for exon detection. *Molecular Biotechnology*, Vol. 10, pp. 27–48, 1998.

[42] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *Proc. 10th ACM-SIAM Symp. Discrete Algorithms,* pp. 235–244, 1999.

[43] R. Cole, R. Hariharan and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time. *Proc. 4th Symposium on Discrete Algorithms (SODA '99)*, pp. 245–254, 1999.

[44] M. O. Dayhoff, R. M. Schwartz and B. C. Orcutt. *Atlas of Protein Sequence and Structure* (M. O. Dayhoff ed.), Vol.5, suppl. 3, pp. 345–352, National Biomedical Research Foundation, Washington D. C., 1978.

[45] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, Vol. 32, No. 3, pp. 505–536, 1985.

[46] A. L. Delcher, D. Harmon, S. Kasif, O. White and S. L. Salzberg. Improved microbial gene identification with GLIMMER. *Nucl. Acid. Res.*, Vol. 27, No. 23, pp. 4636–4641, 1999.

[47] P. Dietz. Finding level-ancestors in dynamic trees, *Proc. 2nd Workshop on Algorithms and Data Structures,* LNCS 1097, pp. 32-40, 1991.

[48] E. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, Vol. 1, pp. 395–412, 1959.

[49] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, Vol. 28, No. 2, pp. 652–673, 1998.

[50] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symp. Foundations of Computer Science,* pp. 137–143, 1997.

[51] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP '96)*, pp. 550–561, 1996.

[52] J. Felsenstein, S. Sawyer and R. Kochin. An efficient method for matching nucleic acid sequences. *Nucl. Acids Res.*, Vol. 10, pp. 133–139, 1982.

[53] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proc. 41st IEEE Symp. on Foundations of Computer Science*, pp. 390–398, 2000.

[54] J. W. Fickett. The gene identification problem: An overview for developers. *Computers Chem.*, Vol. 20, No. 1, pp. 103–118, 1996.

[55] J. W. Fickett and A. G. Hatzigeorgiou. Eukaryotic promoter recognition. *Genome Research*, Vol. 7, pp. 861–878, 1997.

[56] M. Fischer and M. Paterson. String-matching and other products. R. M. Karp eds., *Complexity of Computation*, SIAM-AMS Proc., pp. 113–125, 1974.

[57] R. D. Fleischmann, M. D. Adams, O. White, R. A. Clayton, E. F. Kirkness, A. R. Kerlavage, C. J. Bult, J. F. Tomb, B. A. Dougherty, J. M. Merrick, et al. Whole-genome random sequencing and assembly of Haemophilus Influenzae. *Science*, Vol. 269, pp. 496–512, 1995.

[58] A. Floratos, I. Rigoutsos, L. Parida and Y. Gao. Sequence homology detection through large scale pattern discovery. *Proc. 3rd ACM International Conference on Computational Molecular Biology (RECOMB'99)*, pp. 164–173, 1999.

[59] L. Florea, G. Hartzell, Z. Zhang, G. M. Rubin and W. Miller. A computer program for aligning a cDNA Sequence with a Genomic DNA Sequence. *Genome Res.*, Vol. 8, pp. 967–974, 1998.

[60] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation,* Vol. 104, pp. 197–214, 1993.

[61] D. Frishman, A. Mironov and M. Gelfand. Starts of bacterial genes: estimating the reliability of computer predictions. *Gene*, Vol. 234, pp. 257–265, 1999.

[62] D. Frishman, A. Mironov, H. W. Mewes and M. Gelfand. Combining diverse evidence for gene recognition in completely sequenced bacterial genomes. *Nucl. Acids Res.*, Vol. 26, No. 12, pp. 2941–2947, 1998.

[63] M. S. Gelfand, A. A. Mironov and P. A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Natl. Acad. Sci. USA*, Vol. 93, pp. 9061–9066, 1996.

[64] D. Gelperin. On the optimality of A*. *Artif. Intell.*, Vol. 8, No. 1, pp. 69–76, 1977.

[65] R. Giancarlo. The suffix tree of a square matrix, with applications. *Proc. 4th Symposium on Discrete Algorithms (SODA '93)*, pp. 402–411, 1993.

[66] W. Gish and D. J. States. Identification of protein coding regions by database similarity search. *Nat. Genet.*, Vol. 3, pp. 266–272, 1993.

[67] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, Vol. 162, pp. 705–708, 1982.

[68] O. Gotoh. Optimal alignment between groups of sequences and its application to multiple sequence alignment. *Comput. Applic. Biosci.,* Vol. 9, pp. 361–370, 1993.

[69] O. Gotoh. A weighting system and algorithm for aligning many phylogenetically related sequences. *Comput. Applic. Biosci.,* Vol. 11, pp. 543–551, 1995.

[70] J. Gracy, L. Chiche and J. Sallantin, Improved alignment of weakly homologous protein sequences using structural information. *Protein Engineering*, Vol. 6, pp. 821–829, 1993.

[71] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *Proc. 32nd ACM Symposium on Theory of Computing*, pp. 397–406, 2000.

[72] S. K. Gupta, J. D. Kececioglu and A. A. Schaffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Computational Biology*, Vol. 2, No. 3, pp. 459–472, 1995.

[73] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology,* Cambridge University Press, 1997.

[74] D. Gusfield, K. Bakasubramanian and D. Naor. Parametric optimization of sequence alignment. *Proc. 3rd ACM-SIAM Annual Symposium on Discrete Algorithms,* pp. 432–439, 1992.

[75] S. S. Hannenhalli, W. S. Hayes, A. G. Hatzigeorgiou and J. W. Fickett. Bacterial start site prediction. *Nucl. Acids Res.*, Vol. 27, No. 17, pp. 3577–3582, 1999.

[76] D. Harel and R. R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing,* Vol. 13, pp. 338–355, 1984.

[77] P. A. Hargrave. *Current Opinion on Structural Biology*, Vol. 1, pp. 575–581, 1991.

[78] P. E. Hart, N. J. Nillson and B. Rafael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Sys. Sci. and Cyb.*, Vol. 4, pp. 100–107.

[79] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, Vol. 18, No. 6, pp. 341–343, 1975.

[80] K. Hofmann, P. Bucher, L. Falquet and A. Bairoch. The PROSITE database, its status in 1999. *Nucl. Acids Res.*, Vol. 27, pp. 215–219, 1999.

[81] M. Hsu. A study on the shortest-path algorithm for route navigation. *A Master's Thesis, Department of Information Science, University of Tokyo,* 1994.

[82] X. Huang. On global sequence alignment. *Comput. Appl. Biosci.*, Vol. 10, pp. 227–235, 1994.

[83] X. Huang, P. A. Pevzner and W. Miller. Parametric recomputing in alignment graphs. *Proc. 5th Annual Symposium on Combinatorial Pattern Matching,* LNCS 807, pp. 87–101, 1994.

[84] X. Huang and J. Zhang. Methods for comparing a DNA sequence with a protein sequence. *Comput. Applic. Biosci.*, Vol. 12, pp. 497–506, 1996.

[85] T. Ikeda. Applications of the A* Algorithm to better routes finding and multiple sequence alignment. *A Master's Thesis, Department of Information Science, University of Tokyo,* 1995.

[86] T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Temmoku and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. *Proc. IEEE International Conference on Vehicle Navigation and Information System*, pp. 90–99, 1994.

[87] T. Ikeda and H. Imai. Fast A* algorithms for multiple sequence alignment. *Proc. Genome Informatics Workshop V*, pp. 90–99, 1994.

[88] H. Imai and T. Ikeda. k-group multiple alignment based on A* search. *Proc. Genome Informatics Workshop VI,* pp. 9–18, 1995.

[89] Z. Kan, E. C. Rouchka, W. R. Gish and D. J. States. Gene structure prediction and alternative splicing analysis using genomically aligned ESTs. *Genome Res.*, Vol. 11, No. 5, pp. 889–900, 2001.

[90] J. Kawai, A. Shinagawa, K. Shibata, M. Yoshino, M. Itoh, Y. Ishii, T. Arakawa, A. Hara, Y. Fukunishi, H. Konno, et al. Functional annotation of a full-length mouse cDNA collection. *Nature*, Vol. 409, pp. 685–690, 2001.

[91] M. A. Kehoe, V. Kapur, A. M. Whatmore and J. M. Musser. Horizontal gene transfer among group A streptococci: implications for pathogenesis and epidemiology. *Trends Microbiol.*, Vol. 4, No. 11, pp. 436–43, 1996.

[92] D. K. Kim and K. Park. Linear time construction of 2-D suffix trees. *Proc. 26th International Colloquium on Automata, Languages, and Programming (ICALP '99)*, pp. 463–472, 1999.

[93] H. Konno, Y. Fukunishi, K. Shibata, M. Itoh, P. Carninci, Y. Sugahara and Y. Hayashizaki. Computer-based methods for the mouse full-length cDNA encyclopedia: real-time sequence clustering for construction of a nonredundant cDNA library. *Genome Res.*, Vol. 11, No. 2, pp. 281–289, 2001.

[94] S. R. Kosaraju. Efficient tree pattern matching. *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS '89)*, pp. 178–183, 1989.

[95] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. *Proc. 36th IEEE Symp. Foundations of Computer Science,* pp. 631–637, 1995.

[96] M. Krawczak, J. Reiss and D. N. Cooper. The mutational spectrum of single base-pair substitutions in mRNA splice junctions of human genes: causes and consequences. *Hum. Genet.*, Vol. 90, pp. 41–54, 1992.

[97] H. P. Lenhof, K. Reinert and M. Vingron. A polyhedral approach to RNA sequence structure alignment. *Proc. 2nd Annual International Conference on Computational Molecular Biology (RECOMB '98)*, pp. 153–162, 1998.

[98] A. V. Lukashin and M. Borodovsky. GeneMark.hmm: new solutions for gene identification. *Nucl. Acids Res.*, Vol. 26, No. 4, pp. 1107–1115, 1998.

[99] R. B. Lyngso, M. Zuker and C. N. S. Pedersen. Internal loops in RNA secondary structure prediction. *Proc. 3nd Annual International Conference on Computational Molecular Biology (RECOMB '99)*, pp. 260–267, 1999.

[100] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, Vol. 22, No. 5, pp. 935–948, 1993.

[101] A. Marchler-Bauer, A. R. Panchenko, B. A. Shoemaker, P. A. Thiessen, L. Y. Geer and S. H. Bryant. CDD: a database of conserved domain alignments with links to domain three-dimensional structure. *Nucl. Acids Res.* Vol. 30, pp. 281–283, 2002.

[102] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM,* Vol. 23, pp. 262–272, 1976.

[103] G. D. Means, D. Y. Toy, P. R. Baum and J. M. Derry. A transcript map of a 2-Mb BAC contig in the proximal portion of the mouse X chromosome and regional mapping of the scurfy mutation. *Genomics*, Vol. 65, No. 3, pp. 213–223, 2000.

[104] R. T. Miller, A. G. Christoffels, C. Gopalakrishnan, J. Burke, A. A. Ptitsyn, T. R. Broveak and W. A. Hide. A comprehensive approach to clustering to expressed human gene sequence: The sequence tag alignment and consensus knowledge base. *Genome Res.*, Vol. 9, pp. 1143–1155, 1999.

[105] A. A. Mironov, M. A. Roytberg, P. A. Pevzner and M. S. Gelfand. Performance-guarantee gene predictions via spliced alignment. *Genomics*, Vol. 51, No. 3, pp. 332–339, 1998.

[106] B. Modrek and C. Lee. A genomic view of alternative splicing. *Nat. Genet.*, Vol. 30, No. 1, pp. 13–19, 2001.

[107] R. Mott. EST_GENOME: A program to align spliced DNA sequences to unspliced genomic DNA. *Comput. Applic. Biosci.*, Vol. 13, No. 4, pp. 477–478, 1997.

[108] D. Naor and D. Brutlag. On suboptimal alignments of biological sequences. *Proc. 4th Symposium on Combinatorial Pattern Matching,* LNCS 684, pp. 179–196, 1993.

[109] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pp. 163–185, 1999.

[110] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings.* Cambridge University Press, 2002.

[111] K. M. Nielsen, A. M. Bones, K. Smalla and J. D. van Elsas. Horizontal gene transfer from transgenic plants to terrestrial bacteria-a rare event? *FEMS Microbiol. Rev.*, Vol. 22, No. 2, pp. 79–103, 1998.

[112] N. J. Nilsson. *Problem-solving methods in artificial intelligence*, McGraw-Hill, New York, 1971.

[113] N. J. Nilsson. *Principles of artificial intelligence*, Tioga, Palo Alto, Calif., 1980.

[114] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, Vol. 85, No. 8, pp. 2444–2448, 1988.

[115] P. A. Pevzner. *Computational Molecular Biology*, MIT Press, 2000.

[116] J. Quackenbush, F. Liang, I. Holt, G. Pertea and J. Upton. The TIGR gene indices: reconstruction and representation of expressed gene sequences. *Nucleic Acids Res.*, Vol. 28, No. 1, pp. 141–145, 2000.

[117] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: the Teiresias algorithm. *Bioinformatics*, Vol. 14, No. 1, pp. 55–67, 1998.

[118] I. Rigoutsos and A. Floratos. Motif discovery without alignment or enumeration. *Proc. 2nd ACM International Conference on Computational Molecular Biology (RECOMB'98).* pp. 221–227, 1998.

[119] I. Rigoutsos, A. Floratos, C. Ouzounis, Y. Gao and L. Parida. Dictionary building via unsupervised hierarchical motif discovery. *J. Proteins: Structure, Function and Genetics*, Vol. 37, No. 2, pp. 264–277, 1999.

[120] I. Rigoutsos, A. Floratos, L. Parida, Y. Gao and D. Platt. The emergence of pattern discovery techniques in computational biology. *J. Metabolic Engineering*, Vol. 2, No. 3, pp. 159–177, 2000.

[121] I. Rigoutsos, Y. Gao, A. Floratos and L. Parida. Building dictionaries of 1D and 3D motifs by mining the unaligned 1D sequences of 17 archaeal and bacterial genomes. *Proc. International Conference on Intelligent Systems on Molecular Biology (ISMB'99)*, pp. 223–233, 1999.

[122] K. Robinson, W. Gilbert and G. Church. Large-scale bacterial gene discovery by similarity search. *Nat. Genet.*, Vol. 7, pp. 205–214, 1994.

[123] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. *Proc. 11th Annual International Symposium on Algorithms and Computation (ISAAC'00)*, LNCS 1969, pp. 410–421, 2000.

[124] K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Proc. 12th Genome Informatics (GIW 2001)*, pp. 175–183, 2001.

[125] S. L. Salzberg, A. L. Delcher, S. Kasif and O. White. Microbial gene identification using interpolated Markov models. *Nucl. Acid. Res.*, Vol. 26, No. 2, pp. 544–548, 1998.

[126] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, CSLI Publications, 1999.

[127] M. A. Saqi, P. A. Bates and M. J. E. Sternberg. Towards an automatic method of predicting protein structure by homology: an evaluation of suboptimal sequence alignments. *Protein Engineering,* Vol. 5, pp. 305–311, 1992.

[128] M. A. Saqi and M. J. Sternberg. A simple method to generate non-trivial alternate alignments of protein sequences. *J. Mol. Biol.,* Vol. 219, pp. 727–732, 1991.

[129] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*, PWS Pub. Co., Boston, 1997.

[130] G. Shibayama and H. Imai. Finding K-best alignment of multiple sequences. *Proc. Genome Informatics Workshop IV,* pp. 120–129, 1993.

[131] T. Shibuya. Computing the $n \times m$ Shortest Paths Efficiently. *Proc. 1st International Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, LNCS 1619, pp. 210–225, 1999.

[132] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. *Proc. 10th Annual International Symposium on Algorithms and Computation (ISAAC'99)*, LNCS 1741, pp. 225–236, 1999.

[133] T. Shibuya. New approaches for analyzing recombinations of biological sequences. *Technical Report of IEICE, Vol. 99, No. 30, COMP99-1*, pp. 1–8, 1999.

[134] T. Shibuya. Computing the $n \times m$ shortest paths efficiently. *the ACM Journal of Experimental Algorithmics*, ISSN 1084-6654, Vol. 5, No. 9, 2000.

[135] T. Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT'00)*, LNCS 1851, pp. 393–406, 2000.

[136] T. Shibuya, T. Ikeda, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku. Finding a realistic detour by AI search techniques. *Proc. 2nd Intelligent Transportation Systems*, Vol. 4, pp. 2037–2044, 1995.

[137] T. Shibuya and H. Imai. Parametric alignment of multiple biological sequences. *Proc. Genome Informatics 1996*, pp. 41–50, 1996.

[138] T. Shibuya and H. Imai. Suboptimal alignments of multiple biological sequences. *Intelligent Systems on Molecular Biology 96, Book of Abstracts,* p. 76, 1996, poster session.

[139] T. Shibuya and H. Imai. Enumerating suboptimal alignments of multiple biological sequences efficiently. *Proc. the Pacific Symposium on Biocomputing '97*, pp. 409–420, 1997.

[140] T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. *Proc. 1st Annual International Conference on Computational Molecular Biology (RECOMB '97)*, pp. 267–276, 1997.

[141] T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. *J. Computational Biology*, Vol. 4, No. 3, Mary Ann Liebert, Inc., pp. 385–413, 1997.

[142] T. Shibuya, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku. Detour queries in geographical databases for navigation and related algorithm animations. *Proc. International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'96)*, Vol. 2, pp. 333–340, December 1996.

[143] T. Shibuya, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku. Finding useful detours in geographical databases. *J. IEICE Tras. Information and Systems*, Vol. E82–D, No. 1, pp. 282–290, January 1999.

[144] T. Shibuya and I. Rigoutsos. Dictionary-driven prokaryotic gene finding. *Nucleic Acids Research*, Vol. 30, pp. 2710-2725, 2002.

[145] T. Shibuya, C. Schönbach, H. Kashima and A. Konagaya. Accurate cDNA clustering algorithm based on spliced sequence alignment. Submitted for publication.

[146] Y. Shirai and J. Tsuji. *Artificial Intelligence,* Iwanami Course: Information Science. Vol. 22, Iwanami, Tokyo, 1982, in Japanese.

[147] A. M. Shmatkov, A. A. Melikyan, F. L. Chernousko and M. Borodovsky. Finding prokaryotic genes by the 'frame-by-frame' algorithm: targeting gene starts and overlapping genes. *Bioinformatics*, Vol. 15, No. 11, pp. 874–886, 1999.

[148] A. F. A. Smit and P. Green. *http://ftp.genome.washington.edu/RM/RepeatMasker.html.*

[149] T. E. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, Vol. 147, pp. 195–197, 1981.

[150] J. L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM J. Appl. Math.,* Vol. 49, pp. 1552–1566, 1989.

[151] S-H. Sze and P. A. Pevzner. Las Vegas algorithms for gene recognition: suboptimal and error-tolerant spliced alignment. *J. Comp. Biol.*, Vol. 4, No. 3, pp. 297–309, 1997.

[152] J. D. Thompson, D. G. Higgins and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucl. Acids Res.*, Vol. 22, pp. 4673–4680, 1994.

[153] D. H. Turner, N. Sugimoto and S. M. Freier. RNA structure prediction. *Ann. Rev. Biophys. Chem.*, 17, pp. 167–192, 1988.

[154] T. Ueda, H. Sasaki, Y. Kuwahara, M. Nezu, T. Shibuya, H. Sakamoto, K. Yanagihara, K. Mafune, M. Makuuchi and M. Terada. Deletion of the carboxyl-terminal exons of K-sam/FPGR2 by short homology-mediated recombination, generating preferential expression of specific mRNAs. *Cancer Research*, Vol. 59, No. 24, pp. 6080–6086, Dec 15, 1999.

[155] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica,* Vol. 14, pp. 249–60, 1995.

[156] J. Usuka, W. Zhu and V. Brendel. Optimal spliced alignment of homologous cDNA to a genomic DNA template. *Bioinformatics*, Vol. 16, No. 3, pp. 203–211, 2000.

[157] M. Vingron and M. S. Waterman. Sequence alignment and penalty choices: review of concepts, case studies and implications. *J. Mol. Biol.,* Vol. 235, pp. 1–12, 1994.

[158] Z. Wang and K. Zhang. Finding common RNA secondary structures from RNA sequences. *Proc. 4th Symposium on Combinatorial Pattern Matching,* LNCS 1645, pp. 258–269, 1999.

[159] M. S. Waterman. Parametric and ensemble sequence alignment algorithms. *Bulletin of Mathematical Biology,* Vol. 56, pp. 743–767, 1994.

[160] M. S. Waterman. Introduction to computational biology: maps, sequences and genomes. Chapman & Hall, 1995.

[161] M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J. Mol. Biol.,* Vol. 197, pp. 723–728, 1987.

[162] M. S. Waterman, M. Eggert and E. Lander. Parametric sequence comparisons. *Proc. Natural Academy of Science, USA,* Vol. 89, pp. 6090–6093, 1992.

[163] P. Weiner. Linear pattern matching algorithms. *Proc. 14th Symposium on Switching and Automata Theory,* pp. 1–11, 1973.

[164] Z. Zhang, W. R. Pearson and W. Miller. Aligning a DNA sequence with a protein sequence. *Proc. 1st Annual International Conference on Research in Computational Molecular Biology (RECOMB '97)*, pp. 337–343, 1997.

[165] R. Zimmer and T. Lengauer. Fast and numerically stable parametric alignment of biosequences. *Proc. 1st Annual International Conference on Computational Molecular Biology (RECOMB'97),* pp. 344–353, 1997.

[166] M. Zuker. Suboptimal sequence alignment in molecular biology. Alignment with error analysis. *J. Mol. Biol.,* Vol. 221, pp. 403–420, 1991.