BRICS

Basic Research in Computer Science

BRICS DS-00-4 C. N. S. Pedersen: Algorithms in Computational Biology

Algorithms in Computational Biology

Christian N. S. Pedersen

BRICS Dissertation Series DS-00-4

ISSN 1396-7002 March 2000

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/00/4/`

Algorithms in Computational Biology

Christian Nørgaard Storm Pedersen

Ph.D. Dissertation

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS ·

Department of Computer Science
University of Aarhus
Denmark

Algorithms in Computational Biology

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Christian Nørgaard Storm Pedersen
August 31, 1999

Abstract

In this thesis we are concerned with constructing algorithms that address problems of biological relevance. This activity is part of a broader interdisciplinary area called computational biology, or bioinformatics, that focuses on utilizing the capacities of computers to gain knowledge from biological data. The majority of problems in computational biology relate to molecular or evolutionary biology, and focus on analyzing and comparing the genetic material of organisms. One deciding factor in shaping the area of computational biology is that DNA, RNA and proteins that are responsible for storing and utilizing the genetic material in an organism, can be described as strings over finite alphabets. The string representation of biomolecules allows for a wide range of algorithmic techniques concerned with strings to be applied for analyzing and comparing biological data. We contribute to the field of computational biology by constructing and analyzing algorithms that address problems of relevance to biological sequence analysis and structure prediction.

The genetic material of organisms evolves by discrete mutations, most prominently substitutions, insertions and deletions of nucleotides. Since the genetic material is stored in DNA sequences and reflected in RNA and protein sequences, it makes sense to compare two or more biological sequences to look for similarities and differences that can be used to infer the relatedness of the sequences. In the thesis we consider the problem of comparing two sequences of coding DNA when the relationship between DNA and proteins is taken into account. We do this by using a model that penalizes an event on the DNA by the change it induces on the encoded protein. We analyze the model in detail, and construct an alignment algorithm that improves on the existing best alignment algorithm in the model by reducing its running time by a quadratic factor. This makes the running time of our alignment algorithm equal to the running time of alignment algorithms based on much simpler models.

If a family of related biological sequences is available, it is natural to derive a compact characterization of the sequence family. Among other things, such a characterization can be used to search for unknown members of the sequence family. One widely used way to describe the characteristics of a sequence family is to construct a hidden Markov model that generates members of the sequence family with high probability and non-members with low probability. In the thesis we consider the general problem of comparing hidden Markov models. We define novel measures between hidden Markov models, and show how to compute them efficiently using dynamic programming. Since hidden Markov models are widely used to characterize biological sequence families, our measures and methods for comparing hidden Markov models immediately apply to comparison of entire biological sequence families.

Besides comparing sequences and sequence families, we also consider problems of finding regularities in a single sequence. Looking for regularities in a single biological sequence can be used to reconstruct part of the evolutionary history of the sequence or to identify the sequence among other sequences. In the thesis we focus on general string problems motivated by biological applications because biological sequences are strings. We construct an algorithm that finds all maximal pairs of equal substrings in a string, where each pair of equal substrings adheres to restrictions in the number of characters between the occurrences of the two substrings in the string. This is a generalization of finding tandem repeats, and the running time of the algorithm is comparable to the running time of existing algorithms for finding tandem repeats. The algorithm is based on a general technique that combines a traversal of a suffix tree with efficient merging of search trees. We use the same general technique to construct an algorithm that finds all maximal quasiperiodic substrings in a string. A quasiperiodic substring is a substring that can be described as concatenations and superpositions of a shorter substring. Our algorithm for finding maximal quasiperiodic substrings has a running time that is a logarithmic factor better than the running time of the existing best algorithm for the problem.

Analyzing and comparing the string representations of biomolecules can reveal a lot of useful information about the biomolecules, although the three-dimensional structures of biomolecules often reveal additional information that is not immediately visible from their string representations. Unfortunately, it is difficult and time-consuming to determine the three-dimensional structure of a biomolecule experimentally, so computational methods for structure prediction are in demand. Constructing such methods is also difficult, and often results in the formulation of intractable computational problems. In the thesis we construct an algorithm that improves on the widely used mfold algorithm for RNA secondary structure prediction by allowing a less restrictive model of structure formation without an increase in the running time. We also analyze the protein folding problem in the two-dimensional hydrophobic-hydrophilic lattice model. Our analysis shows that several complicated folding algorithms do not produce better foldings in the worst case, in terms of free energy, than an existing much simpler folding algorithm.

Acknowledgments

The last eight years have passed away quickly. I have learned a lot of interesting things from a lot of interesting people, and I have had many opportunities for traveling to interesting places to attend conferences and workshops. There are many who deserve to be thanked for their part in making my period of study a pleasant time. Here I can only mention a few of them.

Thanks to Rune Bang Lyngsø for being my office mate for many years and for being my co-worker on several projects that range from mandatory assignments in undergraduate courses, to research papers included in this thesis. I have known Rune since we ended up in the same artillery battery almost ten years ago and even though he plans to spend his future far away from Denmark I hope that we can continue to collaborate on projects. Thanks to Gerth Stølting Brodal for always being willing to discuss computer science. Working with Gerth on two of the papers included in this thesis was very inspiring. Besides a lot of good discussions on topics related to our papers, we also spent much time discussing anything from the practical aspects of installing and running Linux on notebooks to computer science in general. Thanks to Ole Caprani for many interesting discussions on conducting and teaching computer science, for letting me be a teaching assistant in his computer architecture course for many years, and for pointing me in the direction of computational biology. Thanks to Jotun Hein for keeping me interested in computational biology by being an always enthusiastic source of inspiration. Also thanks to Lars Michael Kristensen and the other Ph.D. students, and to everybody else who are responsible for the good atmosphere at BRICS and DAIMI.

Thanks to Dan Gusfield for hosting me at UC Davis for seven months and for a lot of inspiring discussions while I was there. Also thanks to Fred Roberts for hosting me at DIMACS for two months. I enjoyed visiting both places and I met a lot of interesting people. Most notably Jens Stoye who was my office mate for seven month at UC Davis. We spent a lot of time discussing many aspects of string matching and computational biology. An outcome of these discussions is included as a paper in this thesis. Jens also had a car which we used for sight seeing trips. Among those a nice skiing trip to Lake Tahoe.

Last, but not least, I would like to thank my advisor Sven Skyum for skillfully guiding me through the four years of the Ph.D. program. His willingness to letting me roam through the various areas of computational biology while making sure that I never completely lost track of the objective has certainly been an important part of the freedom I have enjoyed during my Ph.D. program.

Christian Nørgaard Storm Pedersen,
Århus, August 31, 1999.

My thesis defense was held on March 9, 2000. The thesis evaluation committee was Erik Meineche Schmidt (Department of Computer Science, University of Aarhus), Anders Krogh (Center of Biological Sequence Analysis, Technical University of Denmark), and Alberto Apostolico (Department of Computer Science, University of Padova and Purdue University).

I would like to thank all three members of the committee for attending the defense and for their nice comments about the work presented in the thesis. This final version of the thesis has been subject to minor typographical changes and corrections. Furthermore, it has been updated with current information about the publication status of the included papers.

Christian Nørgaard Storm Pedersen,
Århus, Summer 2000.

Contents

Chapter 1

Introduction

We'll start the war from right here.
— Theodore Roosevelt, Jr., Utah Beach, June 6, 1944.

An algorithm is a description of how to solve a specific problem such that the intended recipient of the description can follow it in a mechanical fashion to solve the problem addressed by the algorithm. With the advent of automated computing devices such as modern computers, an algorithm has in most contexts become synonymous with a description that can be turned into a computer program that instructs a computer how to solve the problem addressed by the algorithm. The ability of modern computers to perform billions of simple calculations per second and to store billions of bits of information, makes it possible by using the proper computer programs to address a wide range of problems that would otherwise remain out of reach. Such possibilities have spawned several interdisciplinary activities where the objective is to utilize the capacities of computers to gain knowledge from huge amounts of data. An important part of such activities is to construct good algorithms that can serve as basis for the computer programs that are needed to utilize the capacities of computers.

1.1 Computational Biology

The work presented in this thesis is concerned with constructing algorithms that address problems with biological relevance. Such work is part of an interdisciplinary area called *computational biology* which is concerned with utilizing the capacities of computers to address problems of biological interest. Computational biology spans several classical areas such as biology, chemistry, physics, statistics and computer science, and the activities in the area are numerous. From a computational point of view the activities are ranging from algorithmic theory focusing on problems with biological relevance, via construction of computational tools for specific biological problems, to experimental work where a laboratory with test tubes and microscopes is substituted with a fast computer and a hard disk full of computational tools written to analyze huge amounts of biological data to prove or disprove a certain hypothesis.

The area of computational biology is also referred to as *bioinformatics*. The two names are used interchangeably, but there seems to be a consensus forming where computational biology is used to refer to activities which mainly focus on constructing algorithms that address problems with biological relevance, while bioinformatics is used to refer to activities which mainly focus on constructing and using computational tools to analyze available biological data. It should be emphasized that this distinction between computational biology and bioinformatics only serves to expose the main focus of the work. This can be illustrated by the work presented in Chapter 6. There we focus on constructing an efficient algorithm for comparing hidden Markov models, but we also implement the constructed algorithm and perform experiments on biological data in order to validate the biological relevance of the algorithm. The work thus contains aspects of both computational biology and bioinformatics.

The work of constructing algorithms that address problems with biological relevance, that is, the work of constructing algorithms in computational biology, consists of two interacting steps. The first step is to pose a biological interesting question and to construct a *model* of the biological reality that makes it possible to formulate the posed question as a *computational problem*. The second step is to construct an *algorithm* that solves the formulated computational problem. The first step requires knowledge of the biological reality, while the second step requires knowledge of algorithmic theory. The quality of the constructed algorithm is traditionally measured by standard algorithmic methodology in terms of the resources, most prominently time and space, it requires to solve the problem. However, since the problem solved by the algorithm originates from a question with biological relevance, its quality should also be judged by the biological relevance of the answers it produces.

The quality of an algorithm that solves a problem with biological relevance is thus a combination of its running time and space assumption and the biological relevance of the answers it produces. These two aspects of the quality of an algorithm both depend on the modeling of the biological reality that led to the formulation of the computational problem that is addressed by the algorithm. Constructing a good algorithm that address a problem with biological relevance is therefore an interdisciplinary activity that involves interchanging between modeling the biological reality and constructing the algorithm, until a reasonable balance between the running time and space assumption of the algorithm, and the biological relevance of the answers it produces, is achieved. The degree of interchanging between modeling and constructing of course depends on how closely related the problem addressed by the algorithm is to a specific biological application, and therefore how relevant it is to judge the algorithm by the biological relevance of the answers it produces.

The details of a specific model and algorithm of course depend on the questions being asked. Most questions in computational biology are related to molecular or evolutionary biology and focus on analyzing and comparing the composition of the key biomolecules DNA, RNA and proteins, that together constitute the fundamental building blocks of organisms. The success of ongoing efforts to develop and use techniques for getting data about the composition of these biomolecules, most prominently DNA sequencing methods for extracting the genetic material from DNA molecules, e.g. [39, 192, 202], has resulted in a flood of available biological data to compare and analyze.

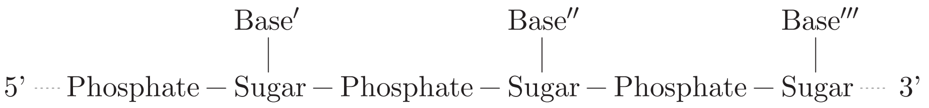


Figure 1.1: An abstract illustration of a segment of a DNA or RNA molecule. It shows that the molecule consists of a backbone of sugars linked together by phosphates with an amine base side chain attached to each sugar. The two ends of the backbone are conventionally called the 5’ end and the 3’ end.

1.2 Biological Sequences

The genetic material of an organism is the blueprint of the molecules it needs for the complex task of living. Questions about how the genetic material is stored and used by an organism have been studied intensively. This has revealed that the biomolecules DNA, RNA and proteins are the important players of the game, and thus important components to model in any method for comparing and analyzing the genetic material of organisms.

The DNA (deoxyribonucleic acid) molecule was discovered in 1869 while studying the chemistry of white blood cells. The very similar RNA (ribonucleic acid) molecule was discovered a few years later. DNA and RNA are chain-like molecules, called polymers, that consist of *nucleotides* linked together by phosphate ester bonds. A nucleotide consists of a phosphoric acid, a pentose sugar and an amine base. In DNA the pentose sugar is 2-deoxyribose and the amine base is either adenine, guanine, cytosine, or thymine. In RNA the pentose sugar is ribose instead of 2-deoxyribose and the amine base thymine is exchanged with the very similar amine base uracil. As illustrated in Figure 1.1 a DNA or RNA molecule is a uniform backbone of sugars linked together by the phosphates with side chains of amine bases attached to each sugar. This implies that a DNA or RNA molecule can be specified uniquely by listing the sequence of amine base side chains starting from one end of the sequence of nucleotides. The two ends of a nucleotide sequence are conventionally denoted the 5’ end and the 3’ end. These names refer to the orientation of the sugars along the backbone. It is common to start the listing of the amine base side chains from the 5’ end of the sequence. Since there is only four possible amine base side chains, the listing can be described as a string over a four letter alphabet.

Proteins are polymers that consists of *amino acids* linked together by peptide bonds. An amino acid consists of a central carbon atom, an amino group, a carboxyl group and a side chain. The side chain determines the type of the amino acid. As illustrated in Figure 1.2 chains of amino acids are formed by peptide bonds between the nitrogen atom in the amino group of one amino acid and the carbon atom in the carboxyl group of another amino acid. A protein thus consists of a backbone of the common structure shared between all amino acids with the different side-chains attached to the central carbon atoms. Even though there is an infinite number of different types of amino acids, only twenty of these types are encountered in proteins. Similar to DNA

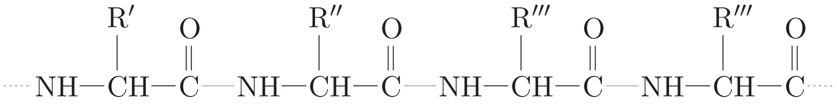


Figure 1.2: An abstract illustration of a segment of a protein. It shows that the molecule consists of a backbone of elements shared between the amino acids with a variable side chain attached to the central carbon atom in each amino acid. The peptide bonds linking the amino acids are indicated by gray lines.

and RNA molecules, it is thus possible to uniquely specify a protein by listing the sequence of side chains. Since there is only twenty possible side chains, the listing can be described as a string over a twenty letter alphabet.

The chemical structure of DNA, RNA and protein molecules that makes it possible to specify them uniquely by listing the sequence of side chains, also called the sequence of residues, is the reason why these biomolecules are often referred to as *biological sequences*. Referring to a biomolecule as a biological sequence signals that the emphasis is only on the sequence of residues and not on other aspects of the biomolecule, e.g. its three-dimensional structure. The correspondence between biological sequences and strings over finite alphabets has many modeling advantages, most prominently its simplicity. For example, a DNA sequence corresponds to a string over the alphabet $\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$, where each character represent one of the four possible nucleotides. Similarly, an RNA sequence corresponds to a string over the alphabet $\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{U}\}$. The relevance of modeling biomolecules as strings over finite alphabets follows from the way the genetic material of an organism is stored and used.

Probably one of the most amazing discoveries of this century is that the entire genetic material of an organism, called its *genome*, is (with few exceptions) stored in two *complementary* DNA sequences that wound around each other in a helix. Two DNA sequences are complementary if the one is the other read backwards with the *complementary bases* adenine/thymine and guanine/cytosine interchanged, e.g. ATTCGC and GCGAAT are complementary because ATTCGC with A and T interchanged and G and C interchanged becomes TAAGCG, which is GCGAAT read backwards. Two complementary bases can form strong interactions, called base pairings, by hydrogen bonds. Hence, two complementary DNA sequences placed against each other such that the head (the 5' end) of the one sequence is placed opposite the tail (the 3' end) of the other sequence is glued together by base pairings between opposition complementary bases. The result is a double stranded DNA molecule with the famous double helix structure described by Watson and Crick in [201]. Despite the complex three-dimensional structure of this molecule, the genetic material it stores only depends on the sequence of nucleotides and can thus be described without loss of information as a string over the alphabet $\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$.

The genome of an organism contains the templates of all the molecules

	2nd				
1st	U	C	A	G	3rd
U	Phe	Ser	Tyr	Cys	U
	Phe	Ser	Tyr	Cys	C
	Leu	Ser	TC	TC	A
	Leu	Ser	TC	Trp	G
C	Leu	Pro	His	Arg	U
	Leu	Pro	His	Arg	C
	Leu	Pro	Gln	Arg	A
	Leu	Pro	Gln	Arg	G
A	Ile	Thr	Asn	Ser	U
	Ile	Thr	Asn	Ser	C
	Ile	Thr	Lys	Arg	A
	Met	Thr	Lys	Arg	G
G	Val	Ala	Asp	Gly	U
	Val	Ala	Asp	Gly	C
	Val	Ala	Glu	Gly	A
	Val	Ala	Glu	Gly	G

Figure 1.3: The genetic code that describes how the 64 possible triplets of nucleotides are translated to amino acids. The table is read such that the triplet AUG encodes the amino acid Met. The three triplets UAA, UAG and UGA are termination codons that signal the end of a translation of triplets.

necessary for the organism to live. A region of the genome that encodes a single molecule is called a *gene*. A *chromosome* is a larger region of the genome that contains several genes. When a particular molecule is needed by the organism, the corresponding gene is *transcribed* to an RNA sequence. The transcribed RNA sequence is complementary to the complementary DNA sequence of the gene, and thus – except for thymine being replaced by uracil – identical to the gene. Sometimes this RNA sequence is the molecule needed by the organism, but most often it is only intended as an intermediate template for a protein.

In eukaryotes (which are higher order organisms such as humans) a gene usually consists of coding parts, called exons, and non-coding parts, called introns. By removing the introns and concatenating the exons, the intermediate template is turned into a sequence of *messenger* RNA that encodes the protein. The messenger RNA is *translated* to a protein by reading it three nucleotides at a time. Each triplet of nucleotides, called a *codon*, uniquely describes an amino acid which is added to the sequence of amino acids being generated.

The correspondence between codons and amino acids are given by the almost universal genetic code shown in Figure 1.3. For example, the RNA sequence UUC CUC is translated to the amino acid sequence Phe Leu. Finding the genes in a genome are of immense interest. It is difficult and not made any easier by the fact that every nucleotide in the genome can be part of up to six different genes. This follows because a nucleotide can end up in any of the three positions in a codon depending on where the transcription starts, combined with the fact that the genome can be transcribed in both directions.

1.3 Outline of Thesis

The rest of this thesis is divided into two parts. The first part of the thesis is a partial overview of the field of computational biology with focus on the results in biological sequence analysis and structure prediction that are described in the papers presented in the second part of the thesis.

The first part consists of three chapters. In Chapter 2 we consider problems of comparing biological sequences and biological sequence families. We focus on methods for comparing two biological sequences in order to determine their evolutionary relatedness, and on methods for comparing entire biological sequence families. This involves the more abstract problem of comparing two hidden Markov models. In Chapter 3 we consider problems of finding regularities in strings. We focus on methods for finding tandem repeats, maximal pairs and maximal quasiperiodic substrings. The methods we present are general string algorithms that can be applied to biological sequence analysis because biological sequences are strings. In Chapter 4 we consider problems in structure prediction concerned with predicting elements of the full three-dimensional structure of a biomolecule from its description as a biological sequence. We focus on methods for predicting the secondary structure of RNA sequences, and on methods for predicting the tertiary structure of proteins.

The second part consists of Chapters 5 through 10. Each of these six chapters contains a reprint of a paper that present research done during my Ph.D. program. Each chapter is self-contained and begins with a short description of the publication status of the results presented in the chapter.

Part I

Overview

Chapter 2

Comparison of Sequences

This was their finest hour.
— Winston S. Churchill, House of Commons, June 18, 1940.

Sequences of characters are among the primary carriers of information in our society. Textual sources such as books, newspapers and magazines document almost every corner of society and provide information for future generations. Reading and comparing the information contained in fragments of old textual sources help historians to reconstruct the history of objects and events.

For a biologist interested in the history of life, fragments of biological sequences that describe the genetic material of organisms serve much the same purpose as fragments of old texts to a historian. Just as new texts are printed everyday and old texts disappear, the genetic material evolves by small discrete changes, called mutations or evolutionary events, that over the course of evolution result in a multitude of different organisms. Mutations that result in organisms that are unfit to survive in the real world most likely result in a branch of evolution that quickly wither away. As the genetic material is stored in DNA sequences and reflected in RNA and protein sequences, it makes sense to compare two or more biological sequences that are believed to have evolved from the same ancestral sequence in order to look for similarities and differences that can help to infer knowledge about the relatedness of the sequences and perhaps to reconstruct part of their common evolutionary history.

For a historian who has to dig through thousands of pages of text in order to establish the circumstances of a historical event the amount of information available can seem staggering, but usually it is nothing compared to the amount of information that face a biologist in terms of known biological sequences. Currently (in July 1999) GenBank, which is a database of known DNA sequences, contains approximately 2.975.000.000 characters distributed in 4.028.000 sequences. This corresponds to a book of 743.750 pages each containing 50 lines of 80 characters. The amount of available data is staggering and grows fast, e.g. in [7] it is referred that plans are that the complete human genome consisting of approximately 3.500.000.000 characters will be sequenced and available for analysis by the end of year 2001. (On June 26, 2000, Celera Genomics, www.celera.com, announced the first assembly of the complete human genome consisting of 3.12 billion base pairs.) To dig through such an amount of data one cannot expect to get far by manual methods. Computational methods to analyze the data are needed. In this Chapter we focus on methods that can be applied to compare biological sequences. In Section 2.1 we focus on the comparison of two sequences in order to determine their evolutionary relatedness. This relates to the work in our paper *Comparison of Coding DNA* presented in Chapter 5. In Section 2.2 we focus on the comparison of more sequences with an emphasis on comparing families of sequences. This relates to the work in our paper *Measures on Hidden Markov Models* presented in Chapter 6.

2.1 Comparison of Two Sequences

When comparing two objects a direct approach is to look for similarities in their appearance. For example, to compare people by their eye colors or the shape of their noses. A more indirect approach is to look for similarities in their history. For example, to compare people by their genealogies. Which approach should be taken depends on the objects and the purpose of the comparison. When comparing two biological sequences both approaches are applicable. The direct approach makes sense because similarities between biological sequences indicate common functionality or three dimensional structure. The indirect approach makes sense because differences between biological sequences can be explained by evolution of the genetic material. Because similarity and evolutionary relatedness of biological sequences are highly correlated it is difficult to draw a clear line between the two approaches.

We focus on the problem of comparing two biological sequences in order to determine their relatedness based on the evolution that has occurred between them. The evolution of a biological sequence is commonly explained as a series of evolutionary events that have transformed an ancestral sequence into the sequence. Two sequences are said to be *homologous* if they have evolved from a common ancestral sequence. The evolution between two homologous sequences, called their evolutionary history, can be explained as the evolutionary events that have occurred in the evolution from the common ancestor to the two sequences. The answer to the question of the evolutionary relatedness of two sequences should be based on their evolutionary history. Most often we can only guess on the evolutionary history of two homologous sequences because no information about the common ancestor or the occurred events is available. To make an educated guess we can use an *evolutionary model* that models the evolution of sequences in such a way that the evolutionary history according to the evolutionary model can be inferred computationally. In the following sections we first describe a widely used way of modeling evolution and formalizing evolutionary relatedness, and then review methods for computing the evolutionary relatedness of two sequences based on this formalization.

2.1.1 Evolutionary Models

An *evolutionary model* is an abstraction of evolution in nature; biological sequences are abstracted as strings over a finite alphabet Σ, evolutionary events

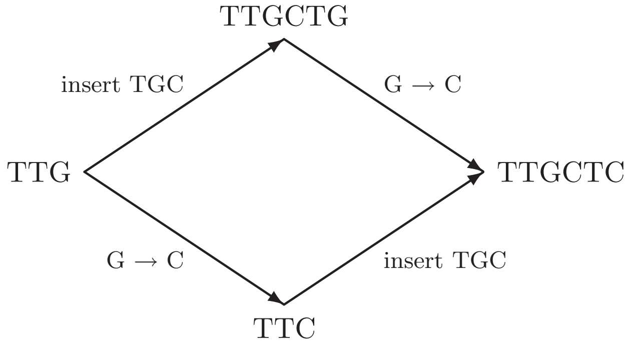


Figure 2.1: Two possible evolutions from TTG to TTGCTC.

are limited to events from a set $\mathcal{E}$ of allowed events, and evolution is quantified by a score function that assigns a score to every possible way of evolving one string into another by a sequence of allowed events. For example, DNA sequences are usually abstracted as strings over the alphabet $\{A, G, C, T\}$ and the evolutionary events acting on them limited to substitutions, insertions and deletions of characters. Figure 2.1 shows two possible evolutionary paths from TTG to TTGCTC that both involves one substitution and one insertion.

To predict evolution we need a guideline. A biological reasonable guideline is the *parsimony principle* which says that evolution in nature follows the path of least resistance. Hence, if we construct the score function of an evolutionary model such that the cost $cost(x \xrightarrow{E} y)$ of evolving string x into string y by a sequence E of events is a measure of the believed resistance of the corresponding evolutionary path in nature, then the parsimony principle tells us that the most likely evolutionary path chosen by nature to evolve one string into another is a sequence of events of minimum cost. This leads to the definition of the *parsimony cost* of evolving string x into string y as the minimum cost of a sequence of events that evolves x into y, that is

$$evol(x, y) = \min\{cost(x \xrightarrow{E} y) \mid E \in \mathcal{E}^*\}\,. \tag{2.1}$$

Two homologous sequences a and b are not necessarily related by a direct evolutionary path but rather via a common ancestor c that has evolved into a and b respectively. The total parsimony cost $evol(c, a) + evol(c, b)$ of evolving the common ancestor c into the sequences a and b is thus a measure of the evolutionary relatedness of a and b. An evident problem of this measure is that the common ancestor c is usually unknown. The parsimony principle yields a solution saying that since nature is cheap, a good guess on a common ancestor is a sequence that minimizes the total parsimony cost of evolving into a and b. This leads to the definition of the *evolutionary distance* between two homologous sequences a and b as the minimum total parsimony cost of evolving a common ancestor into a and b, that is

$$dist(a, b) = \min\{evol(c, a) + evol(c, b) \mid c \in \Sigma^*\}\,. \tag{2.2}$$

The evolutionary distance is a widely used way of formalizing the evolutionary relatedness of biological sequences, e.g. [171, 173, 102, 78, 19]. Most often it is formulated under two reasonable assumptions that simplify its computation by eliminating the need to minimize over all possible common ancestors.

The first assumption is that the score function is additive. Since we have already assumed that evolution can be explained as discrete events it seems reasonable to define the cost of a sequence of events as the sum of the costs of each event. Let $cost(x \xrightarrow{e} y)$ be a function that assigns costs to transforming x into y by a single event e. Let E be a sequence of events $e_1, e_2, \ldots, e_k$ that transforms one string $x^{(0)}$ into another string $x^{(k)}$ as $x^{(0)} \xrightarrow{e_1} x^{(1)} \xrightarrow{e_2} \cdots \xrightarrow{e_k} x^{(k)}$. The cost of evolving string $x^{(0)}$ into string $x^{(k)}$ by the sequence of events E is

$$cost(x^{(0)} \xrightarrow{E} x^{(k)}) = \sum_{i=1}^{k} cost(x^{(i-1)} \xrightarrow{e_i} x^{(i)})\,. \tag{2.3}$$

The second assumption is that events are reversible, that is, for any event e that transforms x into y, there is an event e' that transforms y into x such that $cost(x \xrightarrow{e} y) = cost(y \xrightarrow{e'} x)$, then instead of considering the possible evolutions from c to a and from c to b, we can reverse the direction and consider the possible evolutions from a to c and from c to b. Combined with the assumption of an additive score function this simplifies Equation 2.2 to

$$\begin{aligned} dist(a,b) &= \min\{evol(a,c) + evol(c,b) \mid c \in \Sigma^*\} \\ &= \min\{cost(a \xrightarrow{E} c) + cost(c \xrightarrow{E'} b) \mid c \in \Sigma^* \text{ and } E, E' \in \mathcal{E}^*\} \qquad (2.4) \\ &= \min\{cost(a \xrightarrow{E} b) \mid E \in \mathcal{E}^*\} \end{aligned}$$

The hardness of computing the evolutionary distance between two strings a and b and its relevance as a measure of the evolutionary relatedness of the corresponding biological sequences depends entirely on the parameters of the underlying evolutionary model. The allowed events should be chosen to reflect the events that are believed to have been important in the evolution of the biological sequences in nature and the score function should be chosen to reflect the believed frequency of these events.

In nature evolutionary events affect DNA sequences directly and are reflected in the encoded RNA and protein sequences. The most frequent evolutionary events are *substitution* of a nucleotide with another nucleotide, and *insertion* or *deletion* of a small block of consecutive nucleotides. Less frequent evolutionary events are events that change larger segments of a DNA sequence such as *inversion* that replace a segment with the reversed segment, *transposition* that moves a segment, and *translocation* that exchanges segments between the ends of two chromosomes, and *duplication* that copies a segment.

When considering the most frequent events substitutions, insertions and deletions, the problem of computing the evolutionary distance is usually formulated as an *alignment problem.* We return to this approach in the next section. When considering the less frequent events, e.g. inversion, the problem of computing the evolutionary distance is usually called *genome rearrangement*

$$\begin{bmatrix} \mathrm{T} & - & - & - & \mathrm{T} & \mathrm{G} \\ \mathrm{T} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{C} \end{bmatrix}$$

Figure 2.2: An alignment of TTG and TTGCTC.

to indicate that these events rearrange larger parts of the genome. Most work in this area has been done on computing the so called inversion (or reversal) distance between a sequence of genes. The general idea is to abstract the genome as an integer sequence where each integer represent an encoded gene. Given two integer sequences (the one can without loss of generality be assumed to be the sorted sequence) that describe the order of the same genes in two genomes, the problem is to determine the minimum number of inversions that translate the one sequence into the other. For example, 3241 can be transformed into 1234 by two inversions as $32\underline{41} \rightarrow \underline{321}4 \rightarrow 1234$. The problem of computing the inversion distance between two integer sequences has been shown NP complete in [35]. Variations of the problem have been studied, e.g. [63, 27, 187], and several approximation algorithms have been formulated, e.g. [103, 18].

The modeling of sequence evolution and the derived measure of evolutionary distance as presented in this section is of course not the only approach to formalize the evolutionary relatedness of sequences, but it is probably the most widely used due to its connection to the alignment problem addressed in the next section. Another approach is to model evolution as a stochastic process and to define the score of a sequence of events as the likelihood of that sequence of events as the outcome of the stochastic process. The evolution between two sequences is then predicted as the most likely sequence of events under assumption of the model. One such model is presented by Thorne, Kishino and Felsenstein in [182]. An advantage of the stochastic approach is that it provides a statistical framework that allows us to talk about the likelihood of different evolutionary paths. This makes it more natural to model that evolution in nature *most likely*, but not necessarily, follows the path of least resistance.

2.1.2 Pairwise Alignment

An *alignment* of two strings is a way to communicate a comparison of the two strings. Formally, an alignment of two strings a and b over an alphabet Σ is a $2 \times \ell$ matrix where the entries are either characters from the alphabet or the blank character "–" such that concatenating the non-blank characters in the first and second row yields the strings a and b respectively. Two non-blank characters in the same column of an alignment are said to be aligned, or matched, by the alignment. A maximal block of columns in an alignment where one row consists of only blank characters is called a gap in the alignment. Figure 2.2 shows an alignment of TTG and TTGCTC with three matches and one gap. Columns of two blank characters are normally not allowed in an alignment because they have no meaning in most interpretations of an alignment.

An *alignment score function* is a function that assigns a score to each possible alignment that describes its quality with respect to some criteria. Depending on the score function, we say that an alignment of a and b with either minimum or maximum score has *optimal score* and is an *optimal alignment* of a and b. The *alignment problem* for a given alignment score function is to compute an optimal alignment of two strings a and b.

The alignment notation has been used to compare biological sequences to such an extend that the word "alignment" in many contexts is synonymous with the phrase "comparison of two biological sequences". Responsible for this success is that the alignment notation is useful both as a way to emphasize similarities between strings, and as a way to explain differences between strings in terms of substitutions, insertions and deletions of characters. The success is also due to the fact that efficient methods to solve the alignment problem for biological reasonable score functions have been known since the late 1960's. We review these methods in the next section.

It should come as no surprise that there are many opinions on how to construct a biological reasonable alignment. Much work has been done to construct alignment score functions that attempt to capture how an "expert" would align two sequences. In the rest of this section we review how to formulate an alignment score function that connects the alignment problem with the problem of computing the evolutionary distance cf. Equation 2.4. We take a slightly more general approach than what is common in the literature, e.g. [171, 173], because we want to emphasize the connection between the alignment problem using a classical score function and the alignment problem using the more complicated score function presented in Chapter 5 by explaining how both score functions are instances of the same general score function.

The idea is to view an alignment of two strings a and b as describing a set of substitutions, insertions and deletions of characters that explain the difference between a and b. The aligned characters describe substitutions, and the gaps describe insertions and deletions. For example, the alignment in Figure 2.2 explains the difference between TTG and TTGCTC by a substitution and an insertion that can transform TTG into TTGCTC as shown in Figure 2.1. We define the score of an alignment of a and b as the cheapest way of transforming a into b by a sequence of the events described by the alignment, that is, the score of an alignment $\mathcal{A}$ of a and b that describes events $e_1, e_2, \ldots, e_k$ is

$$\mathit{score}(\mathcal{A}) = \min\{\mathit{cost}(a \overset{E}{\to} b) \mid E = \pi(e_1, e_2, \ldots, e_k)\}\,, \tag{2.5}$$

and the score of an optimal alignment of a and b is

$$\mathit{align}(a, b) = \min\{\mathit{score}(\mathcal{A}) \mid \mathcal{A} \text{ is an alignment of } a \text{ and } b\}\,. \tag{2.6}$$

The function $\mathit{cost}(x \overset{E}{\to} y)$ that assigns costs to transforming x into y by a sequence E of events is defined cf. Equation 2.3, that is, defined in terms of a function $\mathit{cost}(x \overset{e}{\to} y)$ that assigns costs to transforming x into y by a single event e. Since an event is either a substitution, insertion or deletion, this cost function is commonly specified by two functions; the *substitution cost* that assigns costs to transforming x into y by a single substitution, and the *gap cost* that assigns costs to transforming x into y by a single insertion or deletion.

The optimal alignment score of a and b defined by Equation 2.6 is almost the same as the evolutionary distance between a and b defined by Equation 2.4 when allowed evolutionary events are limited to substitutions, insertions and deletions. However, there is a subtle difference because different sequences of events are considered. When computing the evolutionary distance cf. Equation 2.4 we minimize over all possible sequences of substitutions, insertions, and deletions that can transform a into b. When computing the optimal alignment score cf. Equation 2.6 we only minimize over sequences of substitutions, insertions, and deletions that can be expressed as an alignment. This excludes sequences of events where several events act on the same position in the string, e.g. a substitution of a character followed by a deletion of the character. Excluding such sequences of events can be justified by the parsimony principle by saying that nature is unlikely to use two events if one is enough.

For most biological reasonable choices of score function, i.e. substitution and gap cost, the difference in definition between optimal alignment score and evolutionary distance is thus irrelevant because the score function ensures that the cheapest sequence of events which yields the evolutionary distance is not a sequence excluded when computing the optimal alignment score. For example, Wagner and Fisher in [196], and Sellers in [173], show that if the substitution cost is a metric that only depends on the characters being substituted, and the gap cost is a sub-additive function that only depends on the length of the insertion or deletion, then the cheapest sequence of events that transform one string into another can always be expressed as an alignment. This can be generalized to most reasonable score functions including the one presented in Chapter 5. We will not delve by this issue but concentrate on the algorithmic aspects of computing an optimal alignment using the score function in Equation 2.6 for various choices of substitution and gap cost. The structure of these functions decides the complexity of computing an optimal alignment.

Pairwise Alignment using a Classical Score Function

If the substitution cost is given by a function $d : \Sigma \times \Sigma \to \mathbb{R}$ such that $d(\sigma, \sigma')$ is the cost of changing character σ to character σ', and the gap cost is given by a function $g : \mathbb{N} \to \mathbb{R}$ such that $g(k)$ is the cost of insertion or deletion of k characters, then we say that the score function is a classical score function.

A classical score function implies that the score of an alignment cf. Equation 2.5 is the sum of the costs of each event described by the alignment. This eliminates the need to minimize over all possible orders of the events described by an alignment when computing the score of the alignment, e.g. the score of the alignment in Figure 2.2 is $d(T, T)+g(3)+d(T, T)+d(G, C)$. We say that the classical score of an alignment does not depend on the order in which the events described by the alignment take place. This is a significant simplification that makes it possible to compute an optimal alignment of two strings efficiently.

Let $D(i, j)$ denote the score of an optimal alignment of the prefixes $a[1 \mathinner{..} i]$ and $b[1 \mathinner{..} j]$ of the two strings a and b of lengths $n \geq m$. The score of an optimal alignment of a and b is $D(n, m)$. Since the score of an alignment does not depend on the order of the events, we can choose to compute it as the sum of the cost of the *rightmost event* and the costs of the *remaining events*. The rightmost event in the alignment in Figure 2.2 is the substitution of G with C described by the rightmost column and the remaining alignment are the events described by the other five columns. We can thus compute the score $D(i, j)$ of an optimal alignment of $a[1 \mathinner{..} i]$ and $b[1 \mathinner{..} j]$ by minimizing the sum of the cost of the rightmost event and the optimal cost of the remaining events over all possible rightmost events. The rightmost event is either a substitution, an insertion of length k, or a deletion of length k. This leads to the following recurrence for computing $D(i, j)$, where $D(0, 0) = 0$ is the terminating condition.

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + d(a[i], b[j]) & \text{if } i > 0,\ j > 0 \\ \min\limits_{0<k\leq i} \{D(i-k, j) + g(k)\} & \text{if } i > 0,\ j \geq 0 \\ \min\limits_{0<k\leq j} \{D(i, j-k) + g(k)\} & \text{if } i \geq 0,\ j > 0 \end{cases} \tag{2.7}$$

By using dynamic programming, i.e. storing the score $D(i, j)$ in a table entry when computed for the first time, this recurrence gives an algorithm that in time $O(n^3)$ computes the optimal score $D(n, m)$ of an alignment of a and b. By using the table storing the scores $D(i, j)$ for $0 \leq i \leq n$ and $0 \leq j \leq m$ that was built during the computation of the optimal score $D(n, m)$, we can compute an optimal alignment of a and b, and not only its score, by backtracking the steps of the computation of $D(n, m)$ to successively decide what was chosen as the rightmost event. For each step in the backtracking we have to decide among $O(n)$ possible rightmost events, so backtracking takes time $O(n^2)$.

The ideas of this dynamic programming method to compute the optimal score of an alignment of two strings were presented by Needleman and Wunsch in [149]. Their motivation for developing the method was not to compute the evolutionary distance, but to detect similarities between amino acid sequences. In their presentation of the method they want to maximize a similarity instead of minimizing a cost. The original method by Needleman and Wunsch is cleanly explained by Waterman, Smith and Byers in [200].

In many cases the structure of the gap cost allows for a more efficient computation than the one just outlined. If the gap cost is linear, i.e. $g(k) = \alpha k$ for $\alpha > 0$, then the cost of a gap of length k is equal to the cost of k gaps of length one. In this case the score of an alignment can be computed by considering the columns of the alignment independently. Using the above terminology, the rightmost event is either a substitution, an insertion of length one, or a deletion of length one. This leads to the following simplification of the above recurrence where we avoid to consider all possible gap lengths.

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + d(a[i], b[j]) & \text{if } i > 0,\ j > 0 \\ D(i-1, j) + \alpha & \text{if } i > 0,\ j \geq 0 \\ D(i, j-1) + \alpha & \text{if } i \geq 0,\ j > 0 \end{cases} \tag{2.8}$$

By using dynamic programming this simplified recurrence gives an algorithm that in time $O(n^2)$ computes an optimal alignment of two strings a and b of lengths at most n using linear gap cost. The score of an optimal alignment of two strings using linear gap cost is often referred to as the weighted edit distance cf. [196], or the weighted Levenshtein distance cf. [117], between the two strings. (If the gap cost only counts the number of inserted or deleted characters, i.e. $g(k) = k$, and the substitution cost only depends on the equality of the characters, i.e. $d(x, y) = 0$ if $x = y$ and $d(x, y) = 1$ if $x \neq y$, then the prefix "weighted" is removed.)

Distance measures between strings similar to the weighted edit distance, and dynamic programming methods based on recurrences similar to Equation 2.8, have been presented independently by several authors in areas such as speech processing, molecular biology, and computer science. Kruskal in [112, pp. 23–29] gives a good overview of the history and the various discoveries of methods to compute measures similar to the weighted edit distance. These methods are the founding algorithms of computational biology and one feels tempted to describe the period of their discovery by the quote beginning this chapter. Sankoff in [171] formulates a method motivated by comparison of biological sequences. He also describes a variation of the method that makes it possible to specify a bound on the maximum number of insertions and deletions allowed. Wagner and Fisher in [196] formulate a method motivated by automatic spelling correction. They also note that the method for a particular choice of substitution and gap cost can be used to compute the longest common subsequence of two strings. Sellers in [173] considers the mathematical properties of the weighted edit distance and shows that if the substitution cost is a metric on characters, then the weighted edit distance is a metric on strings.

Biologists tend to believe that longer gaps (insertions or deletions) are more common than shorter gaps, e.g. [55, 65, 23]. To model this belief the gap cost should penalize shorter gaps and favor longer gaps. A commonly used way to do this is to use an affine gap cost function, i.e. a function of the form $g(k) = \alpha k + \beta$ for $\alpha, \beta > 0$. Gotoh in [67], and others in [61, 3], show how to compute an optimal alignment of two strings of lengths at most n using affine gap cost in time $O(n^2)$. A more general way is to use a concave gap cost function, i.e. a function g where $g(k+1) - g(k) \leq g(k) - g(k-1)$ as proposed by Waterman in [198]. Both Miller and Myers in [140], and Eppstein, Galil and Giancarlo in [52], show how to compute an optimal alignment of two strings of lengths at most n using concave gap cost in time $O(n^2 \log n)$. Choosing a biological reasonable gap cost is difficult. Biologists want to use a gap cost function that results in the best alignment according to an "experts opinion". Many empirical studies have been done, e.g. Benner *et al.* in [23] propose a concave gap cost function $g(k) = 35.03 - 6.88 \log_{10} d + 17.02 \log_{10} k$ for aligning proteins (where the parameter d is chosen to indicate how much the proteins are believed to have diverged during evolution).

From an algorithmic perspective it is interesting to note that the alignment problem for certain non-trivial choices of substitution and gap cost can be solved more efficient than using the simple quadratic time dynamic programming method. Hunt and Szymanski in [93] show how to compute the

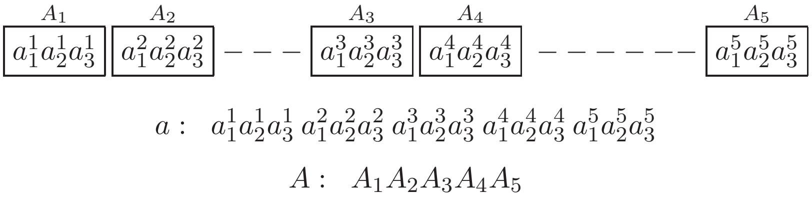


Figure 2.3: If the coding DNA sequence a of a gene encodes a protein A then each amino acid in A is encoded by a triplet of consecutive nucleotides in a. A triplet of nucleotides that encodes an amino acid is called a codon. Because of introns (non-coding parts of the genome) the codons in the coding DNA sequence of the gene are not necessarily consecutive in the genome.

longest common subsequence of two strings a and b of lengths at most n in time $O(r \log n)$ where $r = |\{(i,j) \mid a[i] = b[j]\}|$. In the worst case the parameter r is $O(n^2)$ but if a and b are strings over a large alphabet then it can be expected to be much smaller. In the worst case the method is thus slower than the simple dynamic programming method but if the alphabet size is large it can be expected to perform better. Masek and Paterson in [133] show how to compute the edit distance between two strings a and b of lengths at most n in time $O(n^2/\log^2 n)$. They use a general technique to speed up dynamic programming methods introduced in [13] that is commonly known as the "Four Russians" technique. This technique is also used by Myers in [146, 147] to formulate efficient methods for regular expression pattern matching.

So far we have only been concerned with the time it takes to compute an optimal alignment but space consumption is also important. The dynamic programming methods presented above to compute an optimal alignment of two strings of lengths at most n uses space $O(n^2)$ to store the table of scores $D(i,j)$ used during backtracking. If we avoid backtracking, and only want to compute the score of an optimal alignment, then the space consumption can be reduced to $O(n)$ by observing that the table of scores can be computed row by row. The observation follows because the content of a row only depends on the content of the previous row. It is however not obvious how to compute an optimal alignment, and not only its score, in space $O(n)$.

Hirschberg in [86] presents how to compute the longest common subsequence of two strings of lengths at most n in time $O(n^2)$ and space $O(n)$. The method he uses to compute the longest common subsequence is a dynamic programming method based on a recurrence similar to Equation 2.8. The space-saving technique can thus be used to compute an optimal alignment of two strings of lengths at most n using linear gap cost in time $O(n^2)$ and space $O(n)$. The technique is generalized by Myers and Miller in [148] to compute an optimal alignment of two strings of lengths at most n using affine gap cost in time $O(n^2)$ and space $O(n)$. The technique has become a "common trick" to reduce the space consumption of dynamic programming methods based on recurrences similar to Equation 2.8. A different formulation of the space-saving technique is presented by Durbin *et al.* in [46, Section 2.6]. This formulation makes it easier to adapt the technique to more complicated dynamic programming alignment methods, e.g. the alignment method based on the DNA/Protein score function presented in Chapter 5 and reviewed in next section.

The applications and variations of the alignment methods reviewed in this section are too many to mention. One very popular application is to use alignment methods to search among known sequences stored in a database for sequences that are similar, or closely related, to a new sequence. Heuristic alignment methods such as BLAST [4, 5] and FASTA [118, 119] are probably some of the most used programs in biological sequence analysis. These heuristics achieve a significant speedup compared to the exact dynamic programming method reviewed in this section. This speedup is vital for a biologist who want to search large sequence databases such as GenBank several times a day.

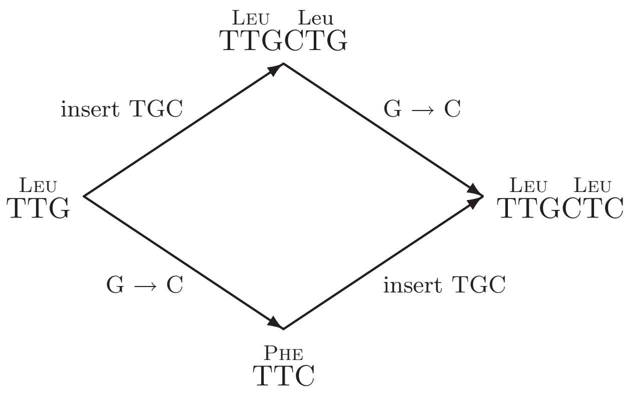
Pairwise Alignment using the DNA/Protein Score Function

In Section 1.2 we explained how proteins are encoded by genes. Figure 2.3 illustrates that each triplet of nucleotides, called a codon, in the coding DNA sequence of a gene encodes an amino acid of the encoded protein cf. the genetic code. The redundancy of the genetic code makes it possible for very different looking DNA sequences to encode the same protein. For example, the two DNA sequences TTG TCT CGC and CTT AGC AGG both encode the same amino acids Leu Ser Arg. This shows that many mutations can occur in a DNA sequence with little or no effect on the encoded protein and implies that proteins evolve slower than the underlying coding DNA sequences.

If we want to compare two DNA sequences that both encode a protein it is difficult to decide whether to compare the DNA sequences or the encoded proteins. If we chose to compare the DNA sequences we risk missing similarities that are only visible in the slower evolving proteins, on the other hand, if we chose to compare the proteins there is no way of telling how alike the underlying codons of two amino acids are and we restrict ourselves to insertions and deletions of amino acids instead of nucleotides. It would be desirable to consider the DNA sequences and the encoded proteins simultaneously.

Hein in [83] presents an algorithm that aligns coding DNA sequences using a score function that models that an event on a coding DNA sequence also influences the encoded protein. We refer to this score function as the DNA/Protein score function. Hein shows how to compute an alignment of two strings of lengths at most n with minimum DNA/Protein score in time $O(n^4)$. In Chapter 5 we examine the DNA/Protein score function in details and present an improved algorithm that computes an alignment of two strings of lengths at most n with minimum DNA/Protein score in time $O(n^2)$. The alignment problem using score functions derived from the DNA/Protein score function is considered by Arvestad in [14] and Hua, Jiang and Wu in [89].

The DNA/Protein score function is a hierarchical score function that penalizes a substitution, insertion or deletion of a nucleotide on the *DNA level* and on the *protein level*. Let a be a DNA sequence that encodes a protein A as illustrated in Figure 2.3. An event that transforms a to a' affects one or



$$c_d(\mathrm{G},\mathrm{C}) + g_d(3) + dist_p(\textsc{Leu}, \textsc{Leu Leu}) + dist_p(\textsc{Leu Leu}, \textsc{Leu Leu})$$

$$c_d(\mathrm{G},\mathrm{C}) + g_d(3) + dist_p(\textsc{Leu}, \textsc{Phe}) + dist_p(\textsc{Phe Leu}, \textsc{Leu Leu})$$

Figure 2.4: The alignment of TTG and TTGCTC in Figure 2.2 describes the substitution $G \to C$ and the insertion of TGC. These two events can occur in two different orders with different DNA/Protein score. The DNA/Protein score of the alignment thus depends on the order in which the events take place.

more codons and therefore also transforms the encoded protein from A to A'. Because of the redundancy in the genetic code it is possible that A and A' are equal. The cost of an event is the sum of its *DNA level cost* and its *protein level cost*. The DNA level cost should reflect the difference between a and a'. This is done by a classical score function that specifies the DNA level cost of substituting a nucleotide x with y as the DNA level substitution cost $c_d(x, y)$, and the DNA level cost of inserting or deleting k nucleotides as the DNA level gap cost $g_d(k)$. The protein level cost should reflect the difference between A and A'. This is done by defining the protein level cost of an event that changes A to A' as the distance between A and A' given by the score of an optimal alignment of A and A' when using a classical score function with substitution cost c_p and gap cost g_p. We use $dist_p(A, A')$ to denote this distance and say that c_p is the protein level substitution cost and that g_p is the protein level gap cost.

The DNA/Protein score of an alignment of two strings is defined cf. Equation 2.6 as the cheapest way of transforming the one string into the other string by a sequence of the events described by the alignment, where the cost of each event is given by the DNA/Protein score function. Figure 2.4 illustrates that in contrast to the classical score of alignment, the DNA/Protein score of an alignment *depends on the order in which the event take place*. The reason is that the protein level cost $dist_p(A, A')$ of an event that changes A to A' can depend on all the characters in A and A', and therefore can depend on all the events that have occurred before it. An important step towards formulating an efficient alignment algorithm using the DNA/Protein score function is to reduce this dependency. Two reasonable assumptions make this possible.

The first assumption is to restrict insertions and deletions to lengths that are divisible by three. The reason for this assumption is that an insertion or deletion of length not divisible by three changes the reading frame and causes a frame shift. Figure 5.1 on page 72 illustrates that a frame shift in a sequence of coding DNA has the power to change the entire suffix of the encoded protein. The assumption to disregard frame shifts can be justified by their rareness in nature that is due to their dramatic effects. The absence of frame shifts implies that an event on a sequence of coding DNA only changes the encoded protein locally, that is, if an event affects only nucleotides in codons that encode a segment X of A, then it changes $A = UXV$ to $A' = UX'V$. The second assumption is restrictions on the protein level substitution and gap cost such that the protein level cost of an event only depends on the amino acids that are encoded by codons affected by the event, that is, such that the protein level cost of an event that changes $A = UXV$ to $A' = UX'V$ only depends on X and X'. The details are stated in Section 5.2.3 and Lemma 5.1 on page 74.

These two assumptions make it possible to compute the protein level cost of an event as illustrated in Figure 5.3 on page 73. In Section 5.3 we explain how this simplification of the protein level cost makes it possible to decompose an alignment into *codon alignments* and compute the DNA/Protein score of the alignment by summing the DNA/Protein score of each codon alignment. A codon alignment is a minimal part of the alignment which aligns an integer number of codons. Figure 2.5 shows an alignment decomposed into codon alignments. Figure 5.5 on page 76 shows another example of an alignment decomposed into codon alignments. The DNA/Protein score of each codon alignment can be computed independently by minimizing over all possible orders of the events described by the codon alignment. Hence, if each codon alignment describes at most some fixed number events independent of the total number of events in the alignment, then the DNA/Protein score of the alignment can be computed in time proportional to the number of codon alignments in the decomposition. This would be a significantly speedup compared to considering all possible orders of the events described by the entire alignment, and an important step towards an efficient alignment algorithm.

Unfortunately, as explained in Section 5.3, a codon alignment in the decomposition of an alignment can describe as many events as the alignment itself. The way to circumvent this problem is to consider only alignments that can be decomposed into (or built of) codon alignments that describe at most some fixed number of events. In the appendix after Section 5.7 we show that if the combined gap cost $g(k) = g_d(3k) + g_p(k)$ is affine, i.e. $g(k) = \alpha k + \beta$, and obeys that $\alpha \geq 2\beta \geq 0$, then an alignment with minimum DNA/Protein score can always be decomposed into codon alignments that describe at most five events. The fifteen different types of codon alignments that describe at most five events are shown in Figure 5.7 on page 77 and Figure 5.14 on page 89.

If we adhere to this assumption on the combined gap cost, the alignment problem using the DNA/Protein score function is thus reduced to computing an alignment of minimum DNA/Protein score that can be decomposed into codon alignments that describe at most five events. The fifteen types of codon alignments that describe at most five events are thus the building blocks of the

$$\left[\boxed{\begin{array}{cccccc} \mathrm{C} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{C} \\ \mathrm{T} & \mathrm{T} & - & - & - & \mathrm{G} \end{array}}\boxed{\begin{array}{cccccc} \mathrm{T} & - & - & - & \mathrm{T} & \mathrm{G} \\ \mathrm{T} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{C} \end{array}}\right]$$

Figure 2.5: Splitting an alignment into codon alignments.

alignments we have to consider in order to find an optimal alignment. When using a classical score function we can compute the score of an alignment as the sum of the cost of the rightmost event and the costs of the remaining events. Similarly, when using the DNA/Protein score function we can compute the score of an alignment as the sum of the cost of the *rightmost codon alignment* and the costs of the *remaining codon alignments*, where the cost of a codon alignment is its DNA/Protein score. This observation suggests a simple algorithm for computing an optimal alignment using the DNA/Protein score function presented by Hein in [83] and summarized in Section 5.4.

The general idea of the algorithm is similar to Equation 2.8. We construct a table D where entry (i, j) holds the score of an optimal alignment of $a[1 \mathinner{..} 3i]$ and $b[1 \mathinner{..} 3j]$. To compute entry (i, j) we minimize over all possible rightmost codon alignments of an alignment of $a[1 \mathinner{..} 3i]$ and $b[1 \mathinner{..} 3j]$, the sum of the cost of the rightmost codon alignment and the optimal cost of the remaining alignment. The cost of the rightmost codon alignment can be computed in constant time as it describes at most five events. The optimal cost of the remaining alignment is $D(i', j')$, where i' and j' depend on the choice of rightmost codon alignment. By using dynamic programming this implies that we can compute entry (i, j) in time proportional to the number of possible rightmost codon alignments in an alignment of $a[1 \mathinner{..} 3i]$ and $b[1 \mathinner{..} 3j]$. This number is bounded by $O(i^2 + j^2)$. In total this gives an algorithm that computes an alignment of two strings of lengths at most n with minimum DNA/Protein score in time $O(n^4)$.

In Section 5.5 we describe how to construct an alignment that computes an alignment of two strings of lengths at most n with minimum DNA/Protein score in time $O(n^2)$. The general idea of the algorithm is similar to the above algorithm. The problem is to avoid having to minimize over all possible rightmost codon alignments. This problem is solved by a lot of bookkeeping in arrays that, so to say, keep track of all possible future situations in such a way that we can pick the best rightmost codon alignment in constant time when the future becomes the present. The idea of keeping track of future situations is vaguely inspired by Gotoh [67] who uses three arrays to keep track of future situations when computing an optimal alignment with affine gap cost. Our bookkeeping is albeit more complicated. By being careful we "only" have to keep approximately 400 arrays. This roughly implies that the constant factor of the $O(n^2)$ running time of our method is about 400 times bigger than the constant factor of the $O(n^2)$ running time of an alignment method based on Equation 2.8. The algorithm as described in Section 5.5 only considers codon alignments of types 1–11. Extending the algorithm to consider also codon alignments of types 12–15 is not difficult and is done in a recent implementation of the algorithm available at www.daimi.au.dk/∼cstorm/combat.

Using this implementation we have performed some preliminary experiments to compare the DNA/Protein score function to simpler score functions that ignore the protein level in order to determine the effects of taking the protein level into account. These experiments indicate that aligning using the DNA/Protein score function is better than aligning using a score function that ignores the protein level when there are few changes on the protein compared to the changes on the underlying DNA. This is not surprising when taking into account how the DNA/Protein score function is designed. We choose not to describe the experiments, or the results, in further details because they are preliminary.

2.2 Comparison of More Sequences

A sequence family is a set of homologous sequences. Members of a sequence family diverge during evolution and share similarities, but similarities that span the entire family might be weak compared to similarities that span only few members of the family. When comparing any two members of the family the faint similarities that span the entire family are thus likely to be shadowed by the stronger similarities between the particular two members. To detect similarities that span an entire sequence family it is therefore advisable to use other methods than just pairwise comparisons of the members.

Comparison of several sequences is a difficult problem that involves many modeling choices. The comparison of several sequences is typical communicated using a multiple alignment that express how the sequences relate by substitutions, insertions, and deletions. In this section we focus on methods to compute multiple alignments and ways to extract a compact characterization of a sequence family based on a comparison of its members. Such a characterization can be used to search for unknown members of the family, and for comparison against the characterizations of other families. This relates to the work presented in Chapter 6.

2.2.1 Multiple Alignment

A *multiple alignment* of a set of strings $S_1, S_2, \dots S_k$ over an alphabet Σ is a natural generalization of a pairwise alignment. A multiple alignment is a $k \times \ell$ matrix $\mathcal{A} = (a_{ij})$, where the entries a_{ij}, $1 \leq i \leq k$ and $1 \leq j \leq \ell$, are either symbols from the alphabet or the blank symbol "–", such that the concatenation of the non-blank characters in row i yields S_i. Figure 2.6 shows a multiple alignment of five strings. Computing a good multiple alignment of a set of strings is a difficult and much researched problem. Firstly, it involves choosing a score function that assigns a score to each possible multiple alignment describing its quality with respect to some criteria. Secondly, it involves constructing a method to compute a multiple alignment with optimal score.

The *sum-of-pairs* score function introduced by Carillo and Lipman in [36] defines the score of a multiple alignment of k strings as the sum of the scores of the $k(k-1)/2$ pairwise alignments induced by the multiple alignment. It

$$\begin{bmatrix} \mathrm{A} & \mathrm{A} & \mathrm{G} & \mathrm{A} & \mathrm{A} & - & \mathrm{A} \\ \mathrm{A} & \mathrm{T} & - & \mathrm{A} & \mathrm{A} & \mathrm{T} & \mathrm{G} \\ \mathrm{C} & \mathrm{T} & \mathrm{G} & - & \mathrm{G} & - & \mathrm{G} \\ \mathrm{C} & \mathrm{C} & - & \mathrm{A} & \mathrm{G} & \mathrm{T} & \mathrm{T} \\ \mathrm{C} & \mathrm{C} & \mathrm{G} & - & \mathrm{G} & - & - \end{bmatrix}$$

Figure 2.6: A multiple alignment of five strings.

is difficult to give a reasonable biological justification of the sum-of-pairs score function but nonetheless it has been widely used, e.g. [16, 68, 145]. If a classical score function with linear gap cost is used to compute the score of the induced pairwise alignments, then an optimal sum-of-pairs multiple alignment of k strings of lengths at most n can be computed in time $O(2^k \cdot n^k)$ and space $O(n^k)$ by a generalization of the dynamic programming method for computing an optimal pairwise alignment. Despite the simplicity of this multiple alignment method, its steep running time and space consumption makes it impractical even for modestly sized sets of relatively short strings.

Wang and Jiang in [197] show that the problem of computing a multiple alignment with optimal sum-of-pairs score is NP hard. However, the need for good multiple alignments has motivated several heuristics and approximation algorithms for computing a multiple alignment with a good sum-of-pairs score. For example, Feng and Doolittle in [54] present a heuristic based on combining good pairwise alignments. Combining good pairwise alignments is also the general idea of the approximation algorithm presented by Bafna *et al.* in [17] which in polynomial time computes a multiple alignment of k strings with a sum-of-pairs score that for any fixed $l < k$ is at most a factor $2 - l/k$ from the optimal score. The approximation algorithm is a generalization of ideas presented by Gusfield in [73] and Pevzner in [163].

Many score functions other than sum-of-pairs, and corresponding methods for computing an optimal multiple alignment, have been proposed in the literature. For example, to construct a multiple alignment of biological sequences it seems natural to take the evolutionary relationships between the sequences into account. Hein in [82] presents a heuristic which simultaneously attempts to infer and use the evolutionary relationships between members of a sequence family to guide the construction of a multiple alignment of the members of the sequence family. Krogh *et al.* in [111] present a popular and successful heuristic for computing multiple alignments, which use profile hidden Markov models to describe the relationships between members of a sequence family. We return to profile hidden Markov models in Section 2.2.2.

A multiple alignment of a set of strings is useful for many purposes. The relationships between strings expressed by a multiple alignment is used to guide many methods that attempt to infer knowledge such as evolutionary history, or common three-dimensional structure, from a set of biological sequences. On the other hand, knowledge about the evolutionary history, or the common three-dimensional structure, of a set of biological sequences can also be used to produce a good multiple alignment. As mentioned above, the method by Hein in [82] attempts to incorporate the correspondence between evolutionary history and multiple alignments into a single method for constructing a multiple alignment while reconstructing the evolutionary history.

In the rest of this section we will not focus on any specific application of multiple alignments, but instead focus on the problem of deriving a compact characterization of a set of strings from a multiple alignment of its members. If the set of strings is a biological sequence family, such a compact characterization has at least two interesting applications. Firstly, it can be used to search a sequence database for unknown members of the family. Secondly, it can be used to compare sequence families by comparing their characterizations rather than comparing the individual members of the families.

The *consensus string* of a set of strings $S_1, S_2, \ldots, S_k$ is a string that attempts to capture the essence of the entire set of strings. There is no consensus on defining a consensus string, but if a multiple alignment of the set of strings is available it seems natural to use the relationships expressed by the multiple alignment to construct the consensus string. Most often this is done by extracting the dominant character from each column in the multiple alignment. In the simplest case the dominant character is chosen as the most frequent occurring character, where ties are broken arbitrarily. Because blanks are not part of the alphabet of the strings, it is common to ignore columns where the dominant character is a blank. This implies that each position in the consensus string corresponds to a column in the multiple alignment, but that columns in the multiple alignment where the dominant character is blank do not correspond to positions in the consensus string. Using this definition a possible consensus string of the multiple alignment in Figure 2.6 is the string CTGAGG, where the sixth column does not correspond to a position in the consensus string because the most frequent character in this column is a blank.

If the aligned set of strings is a biological sequence family, then the extracted consensus string is usually referred to as the *consensus sequence* of the family. It is natural to interpret the consensus sequence of a family as a possible ancestral sequence from which each sequence in the family has evolved by substitutions, insertions, and deletions of characters. Each row in the multiple alignment of the family then describes how the corresponding sequence has evolved from the consensus sequence; a character in a column that corresponds to a position in the consensus sequence has evolved from that position in the consensus sequence by a substitution; a blank in a column that corresponds to a position in the consensus sequence indicates that the character in that position in the consensus sequence has been deleted; a character in a column that does not correspond to a position in the consensus sequence has been inserted.

The consensus sequence is a very compact characterization of a multiple aligned sequence family. The simplicity of the consensus sequence characterization is attractive because it makes it possible to compare sequence families by their consensus sequences using any method for sequence comparison. However, the consensus sequence characterization of a multiple aligned sequence family is probably to coarse-grained because it abstracts away all information

	1	2	3	4	5	6	7
A	0.4	0.2		0.6	0.4		0.2
C	0.6	0.4					
G			0.6		0.6		0.4
T		0.4				0.4	0.2
-			0.4	0.4		0.6	0.2

Figure 2.7: The profile of a multiple alignment in Figure 2.6. The entries of the profile that are not filled are zero. An entry for a character in a column of the profile is the frequency with which that character appears in the corresponding column in the multiple alignment.

in the multiple alignment except for the dominant character in each column. The *profile* of a multiple alignment as introduced by Gribskov *et al.* in [71, 70] is a more fine-grained method to characterize a set of strings from a multiple alignment of its members that attempts to remedy this problem.

A profile of a multiple alignment describes for each column the frequency with which each character in the alphabet (and the blank character) appears in the column. Figure 2.7 shows the profile of the multiple alignment in Figure 2.6. Gribskov *et al.* in [71, 70] show how to compare a profile and a string in order to determine how likely it is that the string is a member of the set of strings characterized by the profile. The general idea of the method is similar to alignment of two strings. The profile is viewed as a "string" where each column is a "character". The objective is to compute an optimal alignment of the string and the profile where the score reflects how well the string fits the profile. This is done by using a position dependent scoring scheme that defines the cost of matching a character from the string against a column in the profile as the sum of the costs of matching the character to each character in alphabet weighted with the frequency with which the character appears in the column of the profile. For example, the cost of matching character G to the second column of the profile in Figure 2.7 is $0.2 \cdot d(A, G) + 0.4 \cdot d(C, G) + 0.4 \cdot d(T, G)$, where $d(x, y)$ is the cost of matching character x with character y. An optimal alignment of a string of length n and a profile of m columns can be computed in time $O(|\Sigma|nm)$, where $|\Sigma|$ is the size of the alphabet of the string and profile. Gotoh in [69] shows how to compare two profiles which makes it possible to compare sequence families by their profile characterizations. The general idea of the method is once again similar to alignment of two strings.

2.2.2 Hidden Markov Models

One of the most popular and successful way to characterize sequence families is to use profile hidden Markov models, which are are simple types of hidden Markov models. A *hidden Markov model* M over an alphabet Σ describes a probability distribution P_M over the set of finite strings $S \in \Sigma^*$, that is, $P_M(S)$ is the probability of the string $S \in \Sigma^*$ under the model M. A hidden Markov model M can be used to characterize a family of strings by saying that a string S

is a member of the family if the probability $P_M(S)$ is significant.

Similar to a Markov model, a hidden Markov model consists of a set of states connected by transitions. Each state has a local probability distribution, the *state transition probabilities*, over the transitions from that state. We use $P_q(q')$ to denote the probability of a transition from state q to q'. The transition structure of a hidden Markov model is can be illustrated as a directed graph with a node for each state, and an edge between two nodes if the corresponding state transition probability is non-zero. Unlike a Markov model, a state in a hidden Markov model can generate a character according to a local probability distribution, the *symbol emission probabilities*, over the characters in the alphabet. We use $P_q(\sigma)$ to denote the probability of generating character $\sigma \in \Sigma$ in state q. A state that does not have symbol emission probabilities is a *silent state*.

It is convenient to think of a hidden Markov model as a generative model in which a *run* generates a string $S \in \Sigma^*$ with probability $P_M(S)$. A run starts in a special start-state, and continues from state to state according to the state transition probabilities, until a special end-state is reached. Each time a non-silent state is entered, a character is generated according to the symbol emission probabilities of that state. A run thus follows a Markovian sequence of states and generates a sequence of characters. The name "*hidden* Markov model" is because the Markovian sequences of states, the path, is hidden while only the generated sequence of characters, the string, is observable.

The basic theory of hidden Markov models was developed and applied to problems in speech recognition in the late 1960's and early 1970's. Rabiner in [166] gives a very good overview of the theory of hidden Markov models and its applications to problems in speech recognition. Hidden Markov models were first applied to problems in computational biology in the late 1980's and early 1990's. Since then they have found many applications, e.g. modeling of DNA [38], protein secondary structure prediction [15], gene prediction [110], and recognition of transmembrane proteins [175]. Probably the most popular application, introduced by Krogh *et al.* in [111], is to use *profile hidden Markov models* to characterize a sequence family by modeling how the sequences relate by substitutions, insertions and deletions to the consensus sequence of the family. The prefix "profile" is because profile hidden Markov models address the same problem as profiles of multiple alignments.

A profile hidden Markov model is characterized by its simple transition structure. Figure 2.8 shows the transition structure of a small profile hidden Markov model. The transition structure consists of repeated elements of *match*, *insert*, and silent *delete* states. The number of repeated elements is the *length* of the model. Each element of a match, insert and delete state models a position in the consensus sequence of the sequence family, and describes how members of the family deviate from the consensus sequence at that position. The match state models that the generated character has evolved from the position in the consensus sequence. The insert state models that the generated character has been inserted between the two neighboring positions in the consensus sequence. The self-loop on the insert state models that several consecutive characters can be inserted between two positions in the consensus sequence. The delete state models that the position has been deleted from the consensus sequence.

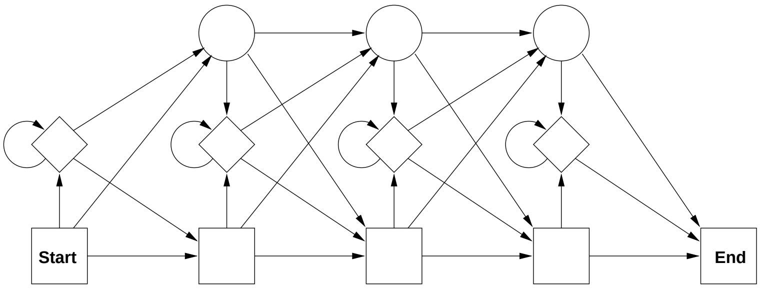


Figure 2.8: The transition structure of a profile hidden Markov model. The squares are the match-states, the diamonds are the insert-states and the circles are the silent delete-states. This figure is copied from the paper in Chapter 6.

The parameters of a profile hidden Markov model M (the length and the transition and emission probabilities) should be chosen to reflect the characteristics of the modeled sequence family (the length of the consensus sequence and how each member relates to the consensus sequence), such that the probability, $P_M(S)$, that it generates a string S can be used to distinguish between members and non-members of the family. The parameters can be chosen in consistence with an existing multiple alignment of members of the sequence family by setting the length of the model to the length of the consensus sequence of the multiple alignment, and by setting the transition and emission probabilities according to the frequency with which each character occurs in each column of the multiple alignment. This approach is similar to constructing a standard profile of a multiple alignment as discussed in the previous section. More interestingly, the parameters can also be estimated from an unaligned set of members of the sequence family. The estimation is done by setting the length of the model to the believed length of the consensus sequence, and by successively adjusting the transition and emission probabilities to maximize the probability of the model having generated the known members of the family.

Adjusting the parameters of a hidden Markov model M to maximize the probability, $P_M(S)$, that it generates a given string S is a fundamental and difficult problem. No exact method exists to decide, in general, the parameters that maximize $P_M(S)$. However, iterative methods that successively refine the parameters of the model, such that $P_M(S)$ is guaranteed to converge to a local maximum, are available. Refining the parameters of a model M to maximize $P_M(S)$ for a given string S is called *training* the model M with respect to S. Many training methods use the forward algorithm (which is described below) to compute for any pair of state q and index i, the probability of being in state q having generated prefix $S[1\,..\,i]$ of the string S. This set of probabilities is subsequently used to adjust the parameters of the model. Training methods are addressed in details by Rabiner in [166], and by Krogh *et al.* in [111].

Fundamental Algorithms for Hidden Markov Models

Many applications of hidden Markov models in speech recognition and computational biology are variations of two fundamental problems and their solutions. The first problem is to determine the probability, $P_M(S)$, that a model, M, generates a given string S. The second problem is to determine the most likely path in a model M that generates a given string S. Because the ideas behind the algorithms solving these problems are fundamental to many applications of hidden Markov models, including the algorithms for comparing hidden Markov models we present in Chapter 6, we present the algorithms in further details below. In the presentation we assume that the transition structure of a model contains no cycles of silent states. If the transition structure contains cycles of silent states, the presentation of the algorithms is more technical, but the asymptotic running times of the algorithms are unaffected by the presence of cycles of silent states. The full details are described in [166, 46].

The first problem, i.e. computing the probability $P_M(S)$ that model M generates string S, is solved by the *forward algorithm*. The general idea of the forward algorithm is to build a table, A, indexed by states from M and indices from S, such that entry $A(q,i)$ holds the probability of being in state q in M having generated the prefix $S[1..i]$ of S. The entry indexed by the end-state and the length of S then holds the desired probability $P_M(S)$ of being in the end-state having generate S. To explain the algorithm we call state q' a *predecessor* of state q in M, if the transition probability from q' to q, $P_{q'}(q)$, is non-zero. The probability of being in state q having generated $S[1..i]$ is then the sum over all predecessors q' of q of the probability of coming to state q via predecessor q' having generated $S[1..i]$. There are two cases. If q is a non-silent state, the last character of $S[1..i]$ is generated in q. In this case $A(q,i)$ is the sum over all predecessors q' of q of terms $A(q', i-1) \cdot P_{q'}(q) \cdot P_q(S[i])$. If q is a silent state, no character is generated in q. In this case $A(q,i)$ is the sum over all predecessors q' of q of terms $A(q',i) \cdot P_{q'}(q)$. In summary we get:

$$A(q,i) = \begin{cases} \displaystyle\sum_{q' \to q} A(q', i-1) \cdot P_{q'}(q) \cdot P_q(S[i]) & \text{if } q \text{ is non-silent} \\ \displaystyle\sum_{q' \to q} A(q', i) \cdot P_{q'}(q) & \text{if } q \text{ is silent} \end{cases} \tag{2.9}$$

By using dynamic programming this recurrence yields an algorithm for computing $P_M(S)$ with running time and space consumption $O(mn)$, where m is the number of transitions in M, and n is the length of S.

The second problem, i.e. computing the probability of the most likely path in model M that generates string S, and the path itself, is solved by the *Viterbi algorithm*. The only difference between the Viterbi algorithm and the forward algorithm is that entry $A(q,i)$ holds the probability of the most likely path to state q that generates $S[1..i]$. This probability is the maximum, instead of the sum, over all predecessors q' of q of the probability of coming to state q via predecessor q' having generated $S[1..i]$. The entry indexed by the end-state and the length of S holds the probability of the most likely path in M

that generates S. The most likely path can be obtained by backtracking the performed maximization steps. The running time and space consumption of the Viterbi algorithm, including backtracking, is the same as the running time and space consumption of the forward algorithm.

The forward and the Viterbi algorithms are both useful when applied to profile hidden Markov models. The probability that a model generates a given string, which is computed by the forward algorithm, and the probability of the most likely path that generates a given string, which is computed by the Viterbi algorithm, are both plausible measures of how likely it is that a string is a member of the sequence family modeled by the profile hidden Markov model. Both measures can be used to search a sequence database for new members of the family, or inversely, to search a database of profile hidden Markov models (families) for the model (family) that most likely describe a new sequence.

Another application of the Viterbi algorithm is to construct a multiple alignment. The idea is to interpret the most likely path in a profile hidden Markov model M that generates the string S as an alignment of S against the consensus sequence of sequence family modeled by M. The interpretation is as follows; if the most like path passes the kth match state, such that $S[i]$ is most likely generated by the kth match state, then $S[i]$ should be matched against the kth character in the consensus sequence; if the most like path passes the kth insert state, such that $S[i]$ is most likely generated by the kth insert state, then $S[i]$ should be inserted between the kth and $(k+1)$st character in the consensus sequence; finally, if the most likely path passes the kth delete state, then the kth character in the consensus sequence has been deleted from S. The most likely path thus explains how to construct S by matching, inserting and deleting characters from the consensus sequence, i.e. describes an alignment of S against the consensus sequence. Alignments of several sequences against the consensus sequence can be combined to a multiple alignment of the sequences.

Constructing multiple alignments using the Viterbi algorithm, combined with training methods to construct profile hidden Markov models from unaligned set of sequences, is one of the most popular and successful heuristics for multiple sequence alignment. It was also the primary application of profile hidden Markov models by Krogh *et al.* in [111]. Other applications of profile hidden Markov models are described by Durbin *et al.* in [46], and Eddy in [48, 49]. Software packages, e.g. SAM [91] and HMMER [47], that implement methods using profile hidden Markov models are widely used, and libraries, e.g. Pfam (available at http://pfam.wustl.edu), that store multiple alignments and profile hidden Markov models characterizations of sequences families are available and growing. In July 1999, the Pfam library contained multiple alignments, and profile hidden Markov models characterizations, of 1488 protein families. (In June 2000, it contained 2290 protein families characterization.)

Algorithms for Comparing Hidden Markov Models

The availability of profile hidden Markov models characterizations, e.g. the Pfam library, motivates the application of comparing entire protein families by comparing their profile hidden Markov model characterizations. This application motivates the more general problem of comparing hidden Markov models. A general method for comparing any systems that can be described by hidden Markov models seems desirable, but the problem of comparing hidden Markov models has not been studied in the literature. Perhaps because the motivation for comparing hidden Markov models has not been apparent until the above stated application of comparing entire biological sequence families by their profile hidden Markov model characterizations.

In Chapter 6 we present measures and methods for comparing hidden Markov models. The proposed measures and methods are not limited to profile hidden Markov models, and are thus applicable to applications beyond comparison of sequence families characterized by profile hidden Markov models.

We define the *co-emission probability*, $A(M_1, M_2)$, of two hidden Markov models, M_1 and M_2, that generate strings over the alphabet Σ, as the probability that the two models independently generate the same string, that is

$$A(M_1, M_2) = \sum_{S \in \Sigma^*} P_{M_1}(S) P_{M_2}(S). \tag{2.10}$$

The co-emission probability is the building block of our measures. The complexity of computing the co-emission probability depends on the transition structure of the two models M_1 and M_2. If the two models are profile hidden Markov models, we can compute the co-emission probability using a dynamic programming algorithm very similar to the forward algorithm. The idea is to build a table, A, indexed by states from the two hidden Markov models, such that entry $A(q, q')$, where q is a state in M_1, and in q' is a state in M_2, holds the probability of being in state q in M_1, and state q' in M_2, having independently generated identical strings on the path to q in M_1, and on the path to q' in M_2. The entry indexed by the two end-states then holds the probability of being in the end-state in both models having generated identical strings, that is, the co-emission probability, $A(M_1, M_2)$, of the two models.

The details of computing $A(q, q')$ for a pair of states, (q, q'), in two profile hidden Markov models, are described in Section 6.3. The general idea is to compute $A(q, q')$ by summing the probabilities of the possible ways of reaching state q in M_1, and state q' in M_2, having generated the same strings. For a pair of states, (g, g'), we say that it is a predecessor pair of (q, q'), if there is a transition from state g to state q in M_1, and a transition from state g' to state q' in M_2. The probability, to be stored in $A(q, q')$, of being in state q in M_1, and in state q' in M_2, having generated the same strings, is the sum over every possible predecessor pair (g, g') of (q, q') of the probability of reaching (q, q') via (g, g') having generated the same strings. If we define

$$p = \sum_{\sigma \in \Sigma} P_q(\sigma) \cdot P_{q'}(\sigma) \tag{2.11}$$

as the probability of generating the same character in state q in M_1, and in state q' in M_2, we can compute $A(q, q')$ by the following recurrence:

$$
A(q,q') = \begin{cases} \displaystyle\sum_{\substack{g \to q \\ g' \to q'}} p \cdot A(g,g') \cdot P_{g'}(q') \cdot P_g(q) & \text{if } q,\, q' \text{ are non-silent} \\ \displaystyle\sum_{\substack{g \to q \\ g' \to q'}} A(g,g') \cdot P_{g'}(q') \cdot P_g(q) & \text{if } q,\, q' \text{ are silent} \\ \displaystyle\sum_{g \to q} A(g,q') \cdot P_g(q) & \begin{array}{c}\text{if } q \text{ is silent, and} \\ q' \text{ are non-silent}\end{array} \end{cases} \tag{2.12}
$$

If every predecessor pair (g,g') of (q,q') is different from (q,q'), the above recurrence shows how to compute $A(q,q')$ recursively. Unfortunately, if both q and q' are insert states, the self-loops on insert states imply that (q,q') is a predecessor pair of (q,q'), which seems to imply that we need to know $A(q,q')$ in order to compute $A(q,q')$. The trick to circumvent this dilemma is to consider how the path to state q in M_1, and the path to state q' in M_2, loops in the self-loop. More precisely, let $A_k(q,q')$ denote the probability of being in insert states q in M_1, and in insert state q' in M_2, having generated the same string, under the additional assumption that one path, say the path to q, has looped exactly k times in the self-loop, ant that the other path, say the path to q', has looped at least k times in the self-loop. The probability $A(q,q')$ of being in insert states q in M_1, and in insert state q' in M_2, having generated the same strings is then the infinite sum over $A_k(q,q')$ for all $k \geq 0$.

It turns out that this infinite sum can be computed efficiently. The first step is to observe that $A_0(q,q')$ can be computed without considering (q,q') as a possible predecessor pair of (q,q'). The reason is that we know by definition that one path, say the path to q, does not loop in the self-loop, so the predecessor of q cannot be q itself. The details are in Equation 6.5 on page 98. The second step is to observe that $A_k(q,q') = rA_{k-1}(q,q') = r^k A_0(q,q')$, where r is the probability of independently choosing the self-loops and generating the same character in state q and q', cf. Equation 6.7 on page 99. The second observation implies that the infinite sum over $A_k(q,q')$, for all $k > 0$, is a geometric series that can be computed as $A(q,q') = \sum_{k=0}^{\infty} r^k A_0(q,q') = A_0(q,q')/(1-r)$.

The running time of the described algorithm for computing the co-emission probability, $A(M_1, M_2)$, of two profile hidden Markov models, M_1 and M_2, is bounded by the time it takes to compute all entries in the table A. Since an entry, $A(q,q')$, can be computed in time proportional to the number of predecessor pairs of (q,q'), the running time is $O(m_1 m_2)$, where m_i is the number of transitions (edges in the transition structure) in M_i.

The above algorithm for computing the co-emission probability for profile hidden Markov models can also be used to compute the co-emission probability for slightly more general hidden Markov models. The only property of the transition structures of the models required by the algorithm is that the states can be numbered such that a transition from state i to state j implies that $i \leq j$. Hidden Markov models with this property are called left-right models cf. [166]. The transition structure of left-right models is, except for self-loops, a directed acyclic graph. In Section 6.5.1 we describe how to extend the algorithm further to handle models, where each state is allowed to be on a single cycle in the transition structure. The extension is quite technical but the running time of the extended algorithm remains $O(m_1 m_2)$. In Section 6.5.2 we describe how to approximate the co-emission probability for general hidden Markov models. The approximation is an iterative process that is guaranteed to converge exponentially fast to the co-emission probability of the models. More precisely, in k rounds we can find an upper and lower bound on $A(M_1, M_2)$ that differ by at most a factor c^k from $A(M_1, M_2)$, where $c < 1$ is a constant that depends on M_1 and M_2. The running time of each round is $O(m_1 m_2)$.

More generally, it is possible to compute the exact co-emission probability for any pair of hidden Markov models, M_1 and M_2, in polynomial time. This is not described in Chapter 6, only hinted at by Equation 6.22 on page 107. The idea is to consider $A(q, q')$ as a variable contributed by state q in M_1 and state q' in M_2, which value is the probability of being in state q in M_1, and in state q' in M_2, having independently generated the same string. Two models M_1 and M_2 in total contribute $n_1 n_2$ variables, where n_i is the number of states in M_i. Each variable can be described in terms of other variables by a linear equation cf. the recurrence in Equation 2.12. The result is a set of $n_1 n_2$ linear equations with $n_1 n_2$ unknowns, where the co-emission probability of M_1 and M_2 is the value of the variable $A(q, q')$, which corresponds to the pair of end-states. The value of this variable can be computed by solving the set of linear equations. The algorithm described above for computing the co-emission probability of two profile hidden Markov models, and more generally, two left-right models, can thus be seen as an efficient way of solving the set of linear equations when the structure of the equations has special properties.

The co-emission probability has a nice mathematically interpretation as an inner product in the infinite dimensional space spanned by all finite strings over a finite alphabet. Consider a hidden Markov model, M, generating strings over a finite alphabet Σ. The probability distribution over the set of finite strings $S \in \Sigma^*$ described by the hidden Markov model can be seen as a vector in the infinite dimensional space spanned by all finite strings over the alphabet Σ, where the coordinate corresponding to a string S in the probability, $P_M(S)$, of model M generating S. In this interpretation of hidden Markov models, the co-emission probability $A(M_1, M_2)$ of two hidden Markov models, M_1 and M_2, over the same alphabet, is simply the inner product, $\langle M_1, M_2 \rangle = |M_1||M_2| \cos v$, of the models, where v is the angle between the models, and $|M_i| = \sqrt{\langle M_i, M_i \rangle}$ is the length of M_i. This formulation of the co-emission probability implies that the co-emission probability $A(M_1, M_2)$ is not itself a good measure of the similarity of the models M_1 and M_2. To see why, consider $A(M_1, M_1) = |M_1||M_1|$ and $A(M_1, M_2) = |M_1||M_2| \cos v$, and observe that if $|M_2| \cos v > |M_1|$, then $A(M_1, M_2) > A(M_1, M_1)$. It is thus perfectly possible, as describe in Proposition 6.1 on page 100, that the model having the largest co-emission probability with a specific model is not the model itself.

To circumvent the problems of the co-emission probability as a similarity measure, we define in Section 6.4 four measures using the co-emission probability as the building block. The four measures are summarized in Figure 2.9. The

$$
\begin{aligned}
D_{\text{angle}}(M_1, M_2) &= \arccos\left(A(M_1, M_2) \Big/ \sqrt{A(M_1, M_1)A(M_2, M_2)}\right) \\
D_{\text{diff}}(M_1, M_2) &= \sqrt{A(M_1, M_1) + A(M_2, M_2) - 2A(M_1, M_2)} \\
S_1(M_1, M_2) &= \cos\left(D_{\text{angle}}(M_1, M_2)\right) \\
S_2(M_1, M_2) &= 2A(M_1, M_2) / (A(M_1, M_1) + A(M_2, M_2))
\end{aligned}
$$

Figure 2.9: The four measures between hidden Markov models M_1 and M_2 defined in Chapter 6 using the co-emission probability as the building block.

first two measures are metrics, where $D_{\text{angle}}(M_1, M_2)$ is the angle between the two models, i.e. $\arccos\left(\langle M_1, M_2\rangle / (|M_1||M_2|)\right)$, and $D_{\text{diff}}(M_1, M_2)$ is the Euclidean norm of the difference between the two models, i.e. $|M_1 - M_2|$. The last two measures, $S_1(M_1, M_2)$ and $S_2(M_1, M_2)$, are similarity measures that fulfill some useful properties stated and explained on page 102. All four measures can be computed within the time it takes to compute the co-emission probabilities $A(M_1, M_1)$, $A(M_2, M_2)$ and $A(M_1, M_2)$.

To evaluate the four measures in practice, we have implemented the algorithm for computing the co-emission probability for left-right models, such that we can compute each of measures efficiently for this type of hidden Markov models. In Section 6.6 we describe an experiment, where we compared fifteen hidden Markov models for three classes of signal peptides. The fifteen models are constructed such that they group into three groups of five models each. The models in each group describe similar properties, and should therefore be more similar, or closer, to each other than to the models in the other groups. To test this hypothesis, we performed all pairwise comparisons between the fifteen models using each of the four measures. The results are shown in Figure 6.4 and 6.5 on page 110 and 111, which show that all measures capture that models within the same group are more alike than models from different groups.

An experiment we plan to perform is to use our measures to evaluate the profile hidden Markov model training methods included in the software packages SAM and HMMER. (Recall that training a (profile) hidden Markov model is to estimate its parameters to maximize the probability of a set of strings being generated by the model.) The general idea of the experiment is to take a model, M, from the Pfam library of models, and use this model to generate a set of strings, which are used to train a model, M', using the traning methods included in SAM and HMMER. Finally, the models M and M' are compared to see how similar, or close, the trained model is to real model. Performing the experiment for different models and training methods, hopefully improves the knowledge of how to train a model. For example, how many strings are needed to make the trained model, M', sufficiently similar to the real model, M, and how do training methods perform if M and M' do not contain the same number of repeated elements, that is, if one model is longer than the other.

Chapter 3

Regularities in Sequences

Never was so much owed by so many to so few.
— Winston S. Churchill, House of Commons, August 20, 1940.

Regularities in experimentally obtained data often reveal important knowledge about the underlying physical system. The physical system could be the quotations on the stock market, the weekly lotto numbers, or biological sequences. Regularities in a biological sequence can be used to identify the sequence among other sequences such as explained below, or to infer information about the evolution of the sequence such as explained in [24].

The genomes of eukaryotes, i.e. higher order organisms such as humans, contain many regularities. Tandem repeats, or tandem arrays, which are consecutive occurrences of the same string, are the most frequent. For example, the six nucleotides TTAGGG appear at the end of every human chromosome in tandem arrays that contain between one and two thousand copies [144]. A number of diseases, such as Fragile X syndrome, Huntington's disease and Kennedy's disease, are all related to tandem repeated regions of the genome. These diseases are caused by an increasing numbers of tandem repeats of a three base long DNA sequence, which somehow interfere with the normal transcription of particular proteins and thereby cause the disease.

Other tandem repeated regions of a genome, the so called *variable number of tandem repeat* (VNTR) regions, are tandem arrays in which the number of repeated DNA sequences varies highly between each individual. If the repeated DNA sequence in a VTNR region is short (between three and five bases), the region is often referred to as a *short tandem repeat* (STR) region. VNTR and STR regions occur frequently and regularly in many genomes, including the human genome, and are very useful as *genetic fingerprints* because two genomes can be distinguished with very high probability by only comparing the number of repeated strings in a few VNTR or STR regions.

Genetic fingerprinting has many applications, most prominently as a tool in forensics or as evidence is criminal or paternity cases. For example, the Danish Department of Justice decided in 1997 that the quality of paternity testing should be such that the probability of an erroneous identification is at most 0.0001. In [142] it is reported that this quality can be achieved by looking at only 9–12 STR regions (and in some cases up to eight other VNTR regions) in the genomes of the child and the potential father. The testing is done by taking blood samples from the child and the potential father. The blood samples are processed in a laboratory to produce data which are examined to “count” the number of repeats in the examined regions. The paternity is decided by comparing the difference between the counts with the expected difference between two random individuals. Genetic fingerprinting is a fascinating combination of molecular, computational and statistical methods. The applications of genetic fingerprinting are numerous and important, so it might turn out that the quote of this chapter also applies to small repeated segments of DNA.

In this chapter we concentrate on the computational aspects of finding certain well defined regularities in strings over a finite alphabet. In Section 3.1 we review tools and methods that combine to a general technique for finding regularities in strings that we apply in the following sections. In Section 3.2.1 we review methods for finding tandem repeats. In Section 3.2.2 we consider the more general problem of finding pairs of identical substrings where the number of characters between the substrings are bounded. This relates to the work in our paper *Finding Maximal Pairs with Bounded Gap* presented in Chapter 7. In Section 3.2.3 we review methods for the problem of finding quasiperiodic substrings which are substrings that can be constructed by concatenations and superpositions of a shorter string. This relates to the work in our paper *Finding Maximal Quasiperiodicities in Strings* presented in Chapter 8.

The methods we present in this chapter can all be applied to biological sequence analysis because biological sequences are strings. There are however two typical objections against this application. The first objection is that most repetitive structures in biological sequences are not exact repetitive structures, but rather repetitions of nearly identical strings. The second objection is that simpler brute force methods are sufficient to find the repetitive structures of current interest in biological sequences. To a large extend these two objections reflect that the methods we present in this chapter are not developed specifically towards biological sequence analysis but rather as general string algorithms.

3.1 Tools and Techniques

In this section we present three useful tools for detecting regularities in strings; suffix trees, height-balanced trees, and sums for analyzing the running time of algorithms. When combined, these three tools imply a general technique for finding regularities in a string S of length n in time $O(n \log n)$ plus the time it takes to report the detected regularities. In short, the general technique is a traversal of the suffix tree, where we at each node compute and use a height-balanced tree that stores the leaf-list of the node.

Throughout this chapter we will use S, α, β and γ to denote strings over some finite alphabet Σ. We will let $|S|$ denote the length of S, $S[i]$ the ith character in S for $1 \leq i \leq |S|$, and $S[i \mathinner{..} j] = S[i]S[i+1] \cdots S[j]$ a substring of S. If $i > j$ then $S[i \mathinner{..} j]$ is the empty string. We say that a string α occurs at position i in the string S if $\alpha = S[i \mathinner{..} i + |\alpha| - 1]$.

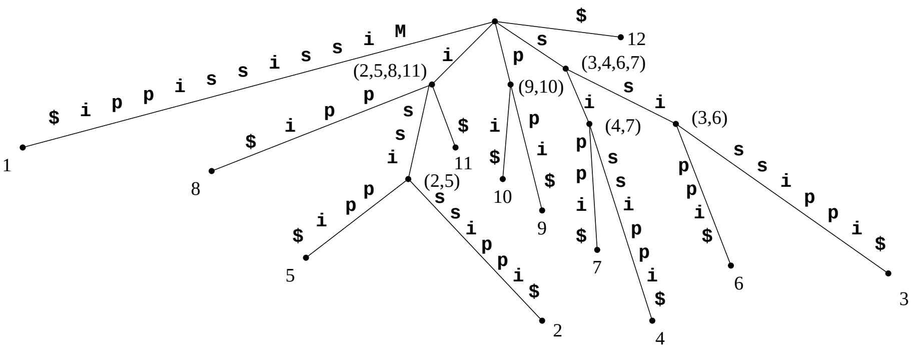


Figure 3.1: The suffix tree of the string *Mississippi* where each node, except the root, is annotated with its leaf-list.

3.1.1 Tries and Suffix Trees

A *trie* is a tree-based data structure for storing a set of strings over some finite alphabet. Every edge in a trie is labelled with a character from the alphabet. The concatenation of the characters on the path from the root to a node v is the *path-label* of v, and is denoted $L(v)$. The labelling of the edges is such that no two edges out of the same node are labelled with the same character, and such that every path-label is a prefix of one or more of the stored strings. This implies that every internal node in a trie has between one and the size of the alphabet children, and that every string stored in a trie is equal to the path-label of a single node. A *compressed trie* is a trie where chains of single-child nodes are compressed into single edges. The edges in a compressed trie are labelled with strings rather than single characters.

The concept of a trie-like structure to represent a set of strings was presented early in this century by Thue in [184]. Fredkin in [60] presented the trie structure (almost) as done above. He also introduced the name "trie" from the word "information re*trie*val". Morrison in [143] presented a tree structure for storing a set of strings based on a trie storing a binary representation of the strings in the set. Knuth describes tries and their history in [105, Section 6.3].

For string matching problems a particular useful variant of the trie is the *suffix tree*. The suffix tree $T(S)$ of a string S is the compressed trie of all suffixes of the string $S\$$, where $\$ \notin \Sigma$. The termination character "\$" ensures that no suffix of $S\$$ is a prefix of another suffix of $S\$$. This implies a one-to-one correspondence between the leaves in the suffix tree and the suffixes of $S\$$. Each leaf in the suffix tree is annotated with an index i that corresponds to its path-label $S[i\,..\,n]\$$, where $n = |S|$. The set of indices stored at the leaves in the subtree rooted at node v is the *leaf-list* of v, and is denoted $LL(v)$. Figure 3.1 shows the suffix tree of the string *Mississippi*, where each node, except the root, is annotated with its leaf-lists.

The suffix tree $T(S)$ of a string S of length n has $n+1$ leaves and at most n internal nodes. Each edge is labelled with a substring of $S\$$ which can be compactly represented by two indices into $S\$$. The suffix tree $T(S)$ can thus be stored in $O(n)$ space. The suffix tree $T(S)$ can be constructed in time $O(n^2)$ by building and compressing the trie of the strings $S[i\,..\,n]\$$ for $i = 1, 2, \ldots, n+1$ in the standard way as described in e.g. [105]. However, the $n+1$ strings, of total length $n(n+1)/2$, which are stored in $T(S)$ are all suffixes of the same string. This relationship has been exploited to construct algorithms that construct the suffix tree $T(S)$ in linear-time $O(n)$.

The first linear-time construction algorithm was presented by Weiner in [203] in 1973. The algorithm was actually presented for the construction of position trees, but was easily adapted for the construction of suffix trees. Few years later, McCreight in [137] presented another linear-time construction algorithm that excels in being more space efficient in practice than Weiner's algorithm. Several years later, Ukkonen in [189] presented an on-line linear-time construction algorithm that is much simpler than the earlier construction algorithms by Weiner and McCreight. Ukkonen however suggested that his algorithm is a heavily disguised version of McCreight's algorithm. The connection is not obvious to the untrained eye but explained in details by Giegerich and Kurtz in [64]. Recently, Farach in [53] presented a construction algorithm that is well suited for construction of suffix trees of strings over large alphabets.

The suffix tree can be used to solve complex string problems. Many applications are reported in [43, 74]. The immediate application of suffix trees is for exact string matching, i.e. to decide if a pattern P of length m occurs in a string S of length n. The classical Knuth-Morris-Pratt algorithm [106] solves this problem in time $\Theta(n+m)$. This time bound can also be achieved using suffix trees. If P occurs in S, then P is a prefix of a suffix of S and there is a path in $T(S)$ starting at the root that spells P. Constructing the suffix tree $T(S)$ takes time $O(n)$. Searching for a path starting at the root that spells P takes time $O(m)$. For example, from the suffix tree in Figure 3.1 follows that the pattern *iss* occurs in the string *Mississipi* at position 2 and 5 because the path that spells *iss* ends on the edge above the node with leaf-list $\{2, 5\}$.

3.1.2 Bounding Traversal Time

The suffix tree $T(S)$ captures many of the regularities in S because the positions in the leaf-list $LL(v)$ tell where the path-label $L(v)$ occurs in S. Many algorithms for finding regularities in a string follow the general scheme of first constructing the suffix tree of the string, then traversing it to examine the leaf-lists in order to detect the regularities. The running time of such algorithms is to a large extend determined by the time spent on examining the leaf-lists at each node. In the worst case, the total length of all leaf-lists at the internal nodes, in a suffix tree of a string of length n, is $\Theta(n^2)$. For example, the total length of all leaf-lists at the internal nodes, in the suffix tree of the string that consists of n identical characters, is $2 + 3 + 4 + \cdots + n = n(n+1)/2 - 1$.

The following two lemmas bound the time an algorithm can spend at each node in a tree with n leaves without spending more than time $O(n \log n)$ in total. The first bound, stated in Lemma 3.1, is referred to in the literature as the "smaller-half trick". It is used in the formulation and analysis of several algorithms for finding tandem repeats, e.g. [41, 11, 177]. The second bound, stated in Lemma 3.2, is hinted at in [138, Exercise 35], and used in [139, Chapter 5] for analysis of finger searches. In accordance with the common name of the first bound, we refer to the second bound as the "extended smaller-half trick". In the papers presented in Chapter 7 and 8, both bounds are used extensively. There the "extended smaller-half trick" is stated and used under the assumption that the tree summed over is a binary tree. Below we state and prove a more general formulation of the "extended smaller-half trick".

Lemma 3.1 (Smaller-half trick) *If each internal node v in a tree with n leaves supplies a term $O(\sum_{i=1}^{k-1} n_i)$, where $n_1 \leq n_2 \leq \cdots \leq n_k$ are the number of leaves in the subtrees rooted at the children of v, then the sum over all terms is $O(n \log n)$.*

Proof. Let $N = n_1 + n_2 + \cdots + n_k$ be the number of leaves in the subtree rooted by v. Since $n_i \leq n_k$ for all $i = 1, 2, \ldots, k-1$, then $n_i \leq N/2$ for all $i = 1, 2, \ldots, k-1$. This implies that a leaf is counted at most $O(\log n)$ times in the sum. The $O(n \log n)$ bound follows because there are n leaves. □

Lemma 3.2 (Extended smaller-half trick) *If each internal node v in a tree with n leaves supplies a term $O(\sum_{i=1}^{k-1} n_i \log(N/n_i))$, where $n_1 \leq n_2 \leq \cdots \leq n_k$ are the number of leaves in the subtrees rooted at the children of v and $N = \sum_{i=1}^{k} n_i$ is the number of leaves in the subtree rooted at v, then the sum over all terms is $O(n \log n)$.*

Proof. As the terms are $O(\sum_{i=1}^{k-1} n_i \log(N/n_i))$ we can find constants, a and b, such that the terms are upper bounded by $\sum_{i=1}^{k-1}(a + b\, n_i \log(N/n_i))$. We will by induction in the number of leaves in the tree prove that the sum over all terms is upper bounded by $a(n-1) + b\, n \log n = O(n \log n)$.

If the tree is a leaf then the upper bound holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $n-1$ leaves. Consider a tree with n leaves where the number of leaves in the subtrees rooted at the children of the root are $n_1 \leq n_2 \leq \cdots \leq n_k$. According to the induction hypothesis the sum over all nodes in these subtrees, i.e. the sum over all nodes in the tree except the root, is bounded by $\sum_{i=1}^{k}(a(n_i - 1) + b\, n_i \log n_i)$. The entire sum is thus bounded by

$$\begin{aligned}\sum_{i=1}^{k}(a(n_i - 1) + b\, n_i \log n_i) + \sum_{i=1}^{k-1} n_i \log(n/n_i)&\\ &= a(n-1) + b\, n_k \log n_k + \sum_{i=1}^{k-1} b\, n_i \log n\\ &< a(n-1) + \sum_{i=1}^{k} b\, n_i \log n\\ &= a(n-1) + b\, n \log n\end{aligned}$$

which proves the lemma. □

3.1.3 Merging Height-Balanced Trees

A *height-balanced tree* is a binary search tree where each node stores an element from a sorted list, such that for each node v, the elements in the left subtree of v are smaller than the element at v, and the elements in the right subtree of v are larger than the element at v. A height-balanced tree satisfies that for each node v, the heights of the left and right subtree of v differ by at most one. Figure 8.3 on page 150 shows a height-balanced tree with 15 elements. A height-balanced tree with n elements has height $O(\log n)$. Operations such as element insertions, element deletions, and membership queries, can all be performed in time $O(\log n)$, where updates are based on performing left and right rotations in the tree. AVL trees [1] are an example of height-balanced trees. We refer to [2, 105] for further details.

When merging two sorted lists that contain n_1 and n_2 elements, there are $\binom{n_1+n_2}{n_2}$ possible placements of the elements of one list in the combined list. For $n_1 \leq n_2$ this gives that $\lceil \log \binom{n_1+n_2}{n_2} \rceil = \Theta(n_1 \log(n_2/n_1))$ comparisons are necessary to distinguish between the possible orderings. Hwang and Lin in [94] show how to merge two sorted lists that contain n_1 and n_2 elements using less than $\lceil \log \binom{n_1+n_2}{n_2} \rceil + \min\{n_1, n_2\}$ comparisons. Brown and Tarjan in [34] first note that it seems difficult to implement the Hwang and Lin merging algorithm with a running time proportional to the number of performed comparisons, then show how to merge two height-balanced trees that store n_1 and n_2 elements, where $n_1 \leq n_2$, in time $O(n_1 \log(n_2/n_1))$. This time bound implies by the "extended smaller-half trick" that n height-balanced trees, that each stores a single element, can be merged together in an arbitrary order, until only one tree remains, in time $O(n \log n)$. Brown and Tarjan in [34] propose this as a way to perform merge sort in time $O(n \log n)$, we propose it as a way to keep track of the leaf-lists during a traversal of a suffix tree.

Let $T(S)$ be the suffix tree of a string S of length n. We define $T_B(S)$ as the binary expansion of $T(S)$, which is obtained by expanding every node v in $T(S)$ that has more than two children, $v_1, v_2, \ldots, v_k$, into a binary tree with root v, and leaves $v_1, v_2, \ldots, v_k$. All new edges in $T_B(S)$ are labelled with the empty string, such that all nodes expanded from v have the same path-label as v. We want to perform a traversal of $T_B(S)$, where we at each leaf construct a height-balanced tree that stores the index at the leaf, and at each internal node v, with children v_1 and v_2, construct a height-balanced tree that stores $LL(v)$ by the merging the height-balanced trees that store $LL(v_1)$ and $LL(v_2)$. The total time it takes to perform this traversal is clearly bounded by the time it takes to construct the height-balanced trees at each internal node. By using the Brown and Tarjan merging algorithm, we can construct the height-balanced tree at node v that stores $LL(v)$ by merging the height-balanced trees that store $LL(v_1)$ and $LL(v_2)$ in time $O(n_1 \log(n_2/n_1))$, where $n_1 = |LL(v_1)|$, $n_2 = |LL(v_2)|$, and $n_1 \leq n_2$. The "extended smaller-half trick" then implies that the total time it takes to perform the traversal is $O(n \log n)$.

Within this time bound we can also traverse $T(S)$ directly, and thus avoid to construct $T_B(S)$ explicitly. At node v we simply merge the height-balanced trees that store the leaf-lists at its children in any sequence of merge operations, until only one height-balanced tree that stores $LL(v)$ remains. However, since there is a one-to-one correspondence between the performed merge operations and the nodes in a possible binary expansion of $T(S)$, the binary expansion $T_B(S)$ is a good mental picture that might clarify the exposition an algorithm based on traversing a suffix tree while merging height-balanced trees.

Traversing the suffix tree $T(S)$, or its binary expansion $T_B(S)$, while keeping track of the leaf-lists by efficient merging of height-balanced trees, is by itself not very useful. What makes it a useful technique for solving various string problems, is the additional operations that can be performed on the height-balanced trees during the traversal without affecting the total time of $O(n \log n)$ it takes to perform the traversal. In Theorem 8.3, on page 150, we list a number of useful operations on height-balanced trees. For example, $\mathsf{MultiPred}(T, e_1, \ldots, e_k)$, that for each e_i finds $\max\{x \in T \mid x \leq e_i\}$, and $\mathsf{MultiSucc}(T, e_1, \ldots, e_k)$, that for each e_i finds $\min\{x \in T \mid x \geq e_i\}$. All the listed operations, including MultiPred and MultiSucc, take a sorted list, $(e_1, e_2, \ldots, e_k)$, of elements as input, and have running time $O(k \cdot \max\{1, \log(N/k)\})$, where N is the number of elements in the height-balanced tree, T, the operation is performed on.

Operations with running time $O(k \cdot \max\{1, \log(N/k)\})$ are very useful when combined with a traversal of $T_B(S)$, where we keep track of the leaf-lists in height-balanced trees. If we at each node v in $T_B(S)$, where $n_1 = |LL(v_1)|$, $n_2 = |LL(v_2)|$, and $n_1 \leq n_2$, choose $k \leq n_1$, and $N \leq n_1 + n_2$, then we can perform any fixed number of such operations in time $O(n_1 \log((n_1 + n_2)/n_1))$, which by the "extended smaller-half trick" is the amount of time that we can spend without affecting the total time of $O(n \log n)$ that it already takes to perform the traversal. This approach has many applications. For example, in Chapter 7 we use height-balanced trees, T_1 and T_2, that store $LL(v_1)$ and $LL(v_2)$, to find, for each element in $LL(v_1)$, its successor in $LL(v_2)$. This is done by performing $\mathsf{MultiSucc}(T_2, e_1, e_2, \ldots, e_k)$, where $e_1, e_2, \ldots, e_k$ are the n_1 elements in $LL(v_1)$ in sorted order. The sorted list of elements can be constructed in time $O(n_1)$ by traversing T_1, and the MultiSucc operation takes time $O(n_1 \log(n_2/n_1))$. In total, this is well within the time that we can afford to spend by the "extended smaller-half trick" without using more than time $O(n \log n)$ in total.

The combination of suffix trees, efficient merging of height-balanced trees, and operations on height-balanced trees, which has been presented in this section, constitutes a general technique to search for regularities in strings. To our knowledge, the general technique has not been applied to string problems, except those presented in Chapter 7 and 8, but we believe that it can be used to improve, or simplify algorithms, for various strings problems, e.g. to improve the running time of the algorithm for the string statistic problem presented by Apostolico and Preparata in [12] from $O(n \log^2 n)$ to $O(n \log n)$.

3.2 Finding Regularities

In this section we review computational methods for finding mathematically well defined regularities in strings. We focus on methods for finding tandem repeats, maximal pairs and maximal quasiperiodicities. These are all types of regularities that can be detected by the suffix tree based methods which we present in details in Chapter 7 and Chapter 8.

3.2.1 Tandem Repeats

A string w is a *tandem array* if it can be constructed by concatenations of a shorter string, i.e. if $w = \alpha^k$ for some $\alpha \in \Sigma^+$ and $k \geq 2$; otherwise the string w is *primitive*. A tandem array α^k is a called *primitive tandem array* if α is primitive. A tandem array α^k occurs at position i in string S if $\alpha^k = S[i \mathinner{..} i + k|\alpha| - 1]$. A tandem array that contains only two repetitions is called a a *tandem repeat* (or *square*), i.e. $\alpha^2 = \alpha\alpha$ is a tandem repeat.

Tandem repeats are probably the most widely studied type of repetitive structure. A tandem repeat $\alpha\alpha$ is a *primitive tandem repeat* if α is primitive. A tandem repeat $\alpha\alpha$ occurs at position i in string S if $\alpha\alpha = S[i \mathinner{..} i + |\alpha\alpha| - 1]$. Two occurences of tandem repeats $\alpha\alpha$ and $\beta\beta$ are of the same *type* if and only if $\alpha = \beta$. A string that contains no occurrences of tandem repeats is *squarefree*. In the beginning of this century Thue in [183, 184] showed that it is possible to construct arbitrary long squarefree strings over any alphabet of more than two characters. Since then several methods have been presented that in time $O(n)$ decide if a string of length n is squarefree, e.g. [132, 165, 42, 43].

To construct and evaluate methods that find occurences of tandem repeats it is useful to know how many occurrences, and types of tandem repeats, there can be in a string of length n. From the definition of a tandem repeat follows that a string that consists of n identical characters contains $\sum_{i=1}^{n} \lfloor i/2 \rfloor = \Theta(n^2)$ occurrences of tandem repeats. It is well known that a string of length n contains at most $O(n \log n)$ occurrences of primitive tandem repeats, e.g. [41, 11, 176, 177]. The detailed analysis in [176] actually shows an upper bound of $1.45(n + 1) \log n - 3.3n + 5.87$ on the number of occurrences of primitive tandem repeats in a string of length n. The highly repetitive Fibonacci strings, defined as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1}F_{n-2}$, are often analyzed when counting occurrences and types of tandem repeats. A Fibonacci string of length n contains $\Theta(n \log n)$ occurrences of primitive tandem repeats, e.g. [41, 59]. The detailed analysis in [59] actually shows that the nth Fibonacci string F_n contains $0.7962|F_n| \log |F_n| + O(|F_n|)$ occurrences of primitive tandem repeats. If we only count the number of different types of tandem repeats that can occur in a string, a recent result by Fraenkel and Simpson in [58] shows that for any position i, in a string S of length n, there are at most two types of tandem repeats, whose rightmost occurrence in S, is at position i. This implies that at most $2n$ different types of tandem repeats can occur in a string of length n. The detailed analysis in [59] shows that $0.7639|F_n| + o(1)$ types of tandem repeats occur in the nth Fibonacci string F_n.

Crochemore in [41] was the first to present a method that finds all occurrences of primitive tandem repeats, in a string S of length n, in time $O(n \log n)$ and space $O(n)$. His algorithm is based on successively refining a partitioning of the positions in S into classes, such that two positions i and j, after refinement step k, are in the same class if and only if $S[i \mathinner{..} i + k - 1] = S[j \mathinner{..} j + k - 1]$. Occurrences of primitive tandem repeats are detected during each refinement step, and each refinement step is done without looking at the positions in the largest class in the current partitioning. The "smaller-half trick", and the $O(n \log n)$ bound on the number of occurrences of primitive tandem repeats,z are used in the analysis of the running time of the algorithm.

Apostolico and Preparata in [11] present a different method for finding all occurences of primitive tandem repeats, in a string S of length n, in time $O(n \log n)$ and space $O(n)$. The general idea of their algorithm is very similar to the general technique for detecting regularities that we present in Section 3.1.3. Their algorithm is also based on a traversal of the binary expansion of the suffix tree $T(S)$ during which they at each node construct a data structure, called a leaf-tree, that stores its leaf-list by merging the leaf-trees that store the leaf-lists of its two children. Occurrences of primitive tandem repeats are detected while merging the leaf-trees. The algorithm differs from an application of the general technique because a leaf-tree is a specially constructed data structure that is less flexible than a general height-balanced tree. The key feature of a leaf-tree is that merging together n leaf-trees, that each stores a single element, by successively merging a smaller leaf-tree into a larger leaf-tree, until only one leaf-tree remains, takes time $O(n \log n)$ by the "smaller-half trick".

Since there are strings of length n that contain $\Theta(n^2)$ occurrences of tandem repeats, reporting all occurrences of tandem repeats in a string of length n takes time $\Theta(n^2)$ in the worst case. Main and Lorentz in [131] were the first to present a method that finds all occurrences of tandem repeats, in a string of length n, in time $O(n \log n + z)$, where z is the number of reported occurrences. The algorithm is a divide-and-conquer method that recursively finds all occurrences of tandem repeats fully contained in the first half of S, and all occurrence of tandem repeats fully contained in the second half of S, then finds all occurrences of tandem repeats that start in the first half of S and end in the second half S. The algorithm does not use the suffix tree. Landau and Schmidt in [113] present a similar algorithm with the same running time. However, their algorithm uses a suffix tree to solve the subproblem of finding all occurrences of tandem repeats that start in the first half of S and end in the second half of S, and it also extends to find all occurrences of k-mismatch tandem repeats in time $O(kn \log(n/k)+z)$. A k-mismatch tandem repeat is a string that becomes an exact tandem repeat after k or fewer characters are changed.

Stoye and Gusfield in [177] present a simple algorithm that finds all occurrences of tandem repeats, in a string S of length n, in time $O(n \log n + z)$. It is easy to modify such that it finds all occurrences of primitive tandem repeats in time $O(n \log n)$. The algorithm is based on a traversal of the suffix tree $T(S)$ during which all branching occurrences of tandem repeats are detected. An occurrence of a tandem repeat $\alpha\alpha$ at position i in S is a brancing occurrence if and only if $S[i + |\alpha|] \neq S[i + |\alpha\alpha|]$. It is easy to verify that there is a branching occurrence of a tandem repeat $\alpha\alpha$ at position i in S, if and only if, there is a node v in the suffix tree $T(S)$ with path-label α, where the indices i and $j = i + |\alpha|$ are in the leaf-lists of distinct children. Stoye and Gusfield show that all occurrences of tandem repeats can be deduced from the branching occurrences of tandem repeats in time proportional to their number. This follows because a non-branching occurrence of a tandem repeat $\alpha\alpha$ at position i implies an occurrence of another tandem repeat $\alpha'\alpha'$ at position $i+1$, where α is a left-rotation of α', i.e. $\alpha = wa$ and $\alpha' = aw$, for some $a \in \Sigma$ and $w \in \Sigma^*$.

Recall that a string of length n contains at most $2n$ different types of tandem repeats. A step towards finding an occurence of each type of tandem repeat that occur in a string more efficiently than finding, and comparing, the occurrences of all tandem repeats in the string, is taken by Kosaraju in [109], who presents an algorithm which, in time $O(n)$, for each position i, in a string S of length n, finds the shortest tandem repeat that occurs at position i. Recently, extending on an idea presented by Crochemore in [42], Gusfield and Stoye in [75] present an algorithm which in time $O(n)$ finds an occurrence of every type of tandem repeat that occur in a string S of length n. Their algorithm annotates the suffix tree $T(S)$ with a marker for each type of tandem repeat that occur in S, which describes the endpoint of the path from the root in $T(S)$ that spells the tandem repeat. Gusfield and Stoye show how to use the annotated suffix tree to list all occurrences of tandem repeats in S in time proportional to their number, which yields an algorithm that in time $O(n+z)$ finds all occcurrences of tandem repeats in a string of length n, where z is the number of reported occurrences. Kolpakov and Kucherov in [107] also present an algorithm that finds all occurrences of tandem repeats in time $O(n+z)$.

3.2.2 Maximal Pairs

A *pair* in a string S is the occurrence of the same substring twice. Formally, we say that $(i, j, |\alpha|)$ is a pair of α in S, if and only if, $1 \leq i < j \leq |S| - |\alpha| + 1$, and $\alpha = S[i \mathinner{..} i + |\alpha| - 1] = S[j \mathinner{..} j + |\alpha| - 1]$. That is, if the substring α occurs at positions i and j in S. The *gap* of a pair $(i, j, |\alpha|)$ is the number of characters $j - i - |\alpha|$ between the two occurrences of α in S. A pair $(i, j, |\alpha|)$ is *left-maximal* if $S[i-1] \neq S[j-1]$, i.e. if the characters to the immediate left of the two occurrences of α are different, and *right-maximal* if $S[i+|\alpha|] \neq S[j+|\alpha|]$, i.e. if the characters to the immediate right of the two occurrences of α are different. A pair $(i, j, |\alpha|)$ is *maximal* if it is both left- and right-maximal. For example, the two occurrences of *ma* in *maximal* form a maximal pair of *ma* with gap two, and the two occurrences of *si* in *Mississippi* form a right-maximal pair of *si* with gap one. From the definition follows that a string of length n contains at most $O(n^3)$ pairs and $O(n^2)$ maximal pairs. This is witnessed by the string $(aab)^{n/3}$, which contains $\Theta(n^3)$ pairs and $\Theta(n^2)$ maximal pairs.

Gusfield in [74, Chapter 7] presents a simple suffix tree based algorithm that finds all maximal pairs, in a string of length n, in time $O(n+z)$, where z is the number of reported maximal pairs. The algorithm is based on the observation illustrated in Figure 3.2, which states that $(i, j, |\alpha|)$ is a right-maximal pair in S if and only if there is a node v in the suffix tree $T(S)$ with path-label α, such that i and j are elements in the leaf-lists of distinct children.

In Chapter 7 we consider the more general problem of finding all maximal pairs in a string S of length n, that adhere to the restrictions on gaps given by two simple functions, $g_1, g_2 : \mathbb{N} \to \mathbb{N}$, that specify the upper and lower bound on allowed gaps. In Section 7.3 we present an algorithm that finds all maximal pairs $(i, j, |\alpha|)$ in S where $g_1(|\alpha|) \leq j - i - |\alpha| \leq g_2(|\alpha|)$, i.e. all maximal pairs

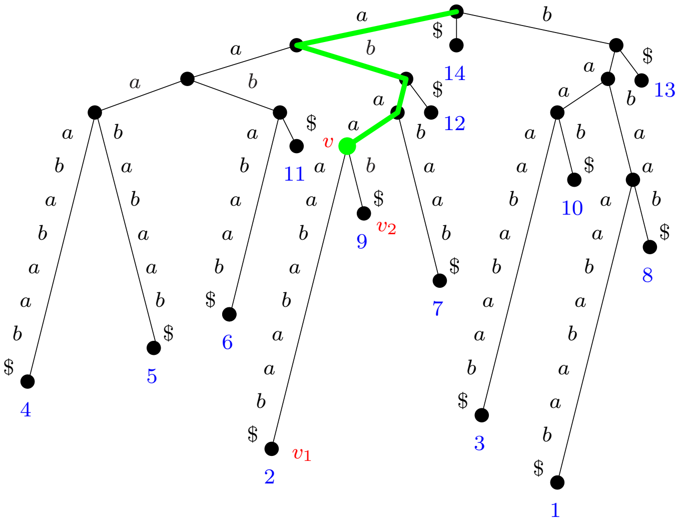


Figure 3.2: The suffix tree of the string *babaaaababaab*, where the marked path to v spells the string *abaa*. This string forms the right-maximal pairs $(2, 9, 4)$, because positions 2 and 9 are elements in leaf-lists of distinct children of v.

with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, in time $O(n \log n + z)$ and space $O(n)$, where z is the number of reported pairs. In Section 7.4 we present an algorithm that finds all maximal pairs $(i, j, |\alpha|)$ in S where $j - i - |\alpha| \geq g_1(|\alpha|)$, i.e. all maximal pairs with gap at least $g_1(|\alpha|)$, in time $O(n + z)$ and space $O(n)$, where z is the number of reported pairs.

The general idea of both algorithms follows from the connection between right-maximal pairs and suffix trees. The idea is to traverse the binary expansion of the suffix tree, $T_B(S)$, and to report the right-maximal pairs $(i, j, |\alpha|)$ at node v that are maximal and adhere to the restrictions on gaps given by g_1 and g_2, where α is the path-label of v, and i and j are elements in the leaf-lists of distinct children of v. A problem when implementing this idea is to avoid to inspect all right-maximal pairs in order to decide which ones are maximal with gaps adhering to the restrictions given by g_1 and g_2. We address this problem in two steps. Firstly, we develop algorithms that find only right-maximal pairs with gaps adhering to the restrictions given by g_1 and g_2. Secondly, we extend these algorithms to filter out right-maximal pairs that are not left-maximal. In the following we will explain the techniques we use to implement these two steps in both algorithms. Throughout the explanation we will use v to denote a node in $T_B(S)$ with path-label α and children v_1 and v_2, where $LL(v_1) = (p_1, p_2, \ldots, p_s)$ and $LL(v_2) = (q_1, q_2, \ldots, q_t)$ such that $|LL(v_1)| \leq |LL(v_2)|$, $p_i \leq p_{i+1}$ for $1 \leq i < s$, and $q_j \leq q_{j+1}$ for $1 \leq j < t$,

The algorithm for finding all maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is an application of the general technique for detecting regularities presented in Section 3.1.3. When we visit node v during the traversal of $T_B(S)$, two height-balanced trees, T_1 and T_2, that store the leaf-lists $LL(v_1)$ and $LL(v_2)$ are available. To report all right-maximal pairs $(i, j, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, where $i \in LL(v_1)$, and $j \in LL(v_2)$, we first $\mathsf{MultiSucc}(T_2, e_1, e_2, \ldots, e_s)$, where $e_i = p_i + g_1(|\alpha|) + |\alpha|$, which returns for every p_i in $LL(v_1)$ a reference to the node in T_2 that stores the minimum element q_j in $LL(v_2)$ such that the pair $(p_i, q_j, |\alpha|)$ has gap at least $g_1(|\alpha|)$. Starting at that node, we traverse T_2 in order to visit nodes that store increasing elements such that we can report the right-maximal pairs $(p_i, q_j, |\alpha|)$, $(p_i, q_{j+1}, |\alpha|)$, $(p_i, q_{j+2}, |\alpha|)$, and so on, until the gap exceeds $g_2(|\alpha|)$.

Since the sorted list of elements $(e_1, e_2, \ldots, e_s)$ can be constructed in time $O(n_1)$ by traversing T_1, and the MultiSucc operation can be performed in time $O(n_1 \log(n_2/n_2))$, where $n_1 = |LL(v_1)|$ and $n_2 = |LL(v_2)|$, the time we spend at node v, not counting the time spent on reporting pairs, is well within the time allowed by the "extended smaller-half trick". It is easy to see that reporting takes time time proportional to the number of reported pairs. The total running time of the algorithm for finding right-maximal pairs with bounded gap is thus $O(n \log + z)$, where z is the number of reported pairs.

To extend the reporting step of the above algorithm to report only maximal pairs, we must avoid to report right-maximal pairs that are not left-maximal. To ensure that the extended reporting step still takes time proportional to the number of reported pairs, we must avoid to look explicitly at all right-maximal pairs that are not left-maximal. This is achieved by maintaining an additional height-balanced tree during the traversal of $T_B(S)$, which makes it possible, in constant time, to skip blocks of consecutive elements in $LL(v_2)$ that all have the same character to their immediate left in S. This height-balanced tree, the block-start tree, is used to extend the reporting step such that whenever we are about to report a pair $(p_i, q_j, |\alpha|)$, where $S[p_i - 1] = S[q_j - 1]$, we instead skip the block of elements in $LL(v_2)$ that starts with q_j, and have $S[p_i - 1]$ as the character to their immediate left in S, and continue reporting from the index that follows this block. The details are in Section 7.3.2.

The algorithm for finding all maximal pairs with gap at least $g_1(|\alpha|)$ is also based on a traversal of $T_B(S)$. However, its running time of $O(n + z)$ implies that it cannot afford to keep track of the leaf-lists in height-balanced trees. It turns out that the lack of an upper bound on the allowed gaps makes it possible to report the proper pairs using a reporting scheme that can be implemented without using height-balanced trees. The idea of the reporting scheme is to report pairs $(i, j, |\alpha|)$ at node v, where we start with i and j being from opposite ends of the leaf-lists $LL(v_1)$ and $LL(v_2)$, and work them inwards in the leaf-lists, until the gap becomes smaller than $g_1(|\alpha|)$.

More precisely, to report all right-maximal pairs $(i, j, |\alpha|)$ with gap at least $g_1(|\alpha|)$, where $i \in LL(v_1)$ and $j \in LL(v_2)$, we report for every p_i, in increasing order, the pairs $(p_i, q_t, |\alpha|)$, $(p_i, q_{t-1}, |\alpha|)$, $(p_i, q_{t-2}, |\alpha|)$, and so on, until the gap becomes smaller than $g_1(|\alpha|)$. We stop reporting when the gap of the pair $(p_i, q_t, |\alpha|)$ is smaller than $g_1(|\alpha|)$, when this happens, all pairs $(p_{i'}, q_{t'}, |\alpha|)$, where $i' \geq i$ and $t' < t$, have an even smaller gap. If the leaf-lists, $LL(v_1)$ and $LL(v_2)$, are stored in the heap-tree data structure described in Section 7.4.1, then we can implement the reporting scheme in time proportional to the number of reported pairs. The details are in Algorithm 7.3 in Section 7.4.2. The key feature of heap-trees is that two heap-trees can be merged in amortized constant time. A traversal of $T_B(S)$, where we at each node construct a heap-tree that stores its leaf-list by merging the heap-trees that store the leaf-lists of its children, thus takes time $O(n)$ in the worst case. Since the reporting at each node takes time proportional to the number of reported pairs, this implies that the total running of the algorithm for finding right-maximal pairs with lower bounded gap time is $O(n + z)$, where z is the number of reported pairs.

To extend the reporting step to report only maximal pairs, we need to store the leaf-lists in a slightly more complicated data structure, which we call a colored heap-tree. Besides supporting the operations of an ordinary heap-tree, a colored heap-tree also makes it possible, in constant time, to skip blocks of elements in $LL(v_2)$ that have the same character to their immediate left in S. As in the previous algorithm, this operation is used during the reporting step to avoid to inspect explicitly all right-maximal pairs that are not left-maximal. The details are in Algorithm 7.4 in Section 7.4.2

Finding pairs with bounded gap is a flexible method for detecting various regularities in strings. For example, a right-maximal pair $(i, j, |\alpha|)$ with gap zero corresponds to a branching occurrence of a tandem repeat $\alpha\alpha$ at position i. As explained in Section 3.2.1, all occurrences of tandem repeats in a string can be deduced from the set of branching occurrences of tandem repeats in the string in time proportional to their number. This implies that the particular instance of our algorithm which finds all right-maximal pairs with gap zero can be used as yet another method for finding all occurrences of tandem repeats in time $O(n \log n + z)$, where z is the number of reported occurrences.

3.2.3 Maximal Quasiperiodicities

Recall that a string is primitive if it cannot be constructed by concatenations of a shorter string. The string *abaabaabaab* is primitive, but is can be constructed by two superpositions of the string *abaab*. The string *abaab* is also primitive, but unlike the string *abaabaabaab* it cannot be constructed by superpositions of a shorter string. In some sense this makes the string *abaab* more primitive than the string *abaabaabaab*. To formalize this difference between primitive stings, Ehrenfeucht, cf. [8], suggested the notation of a quasiperiodic string.

A string is *quasiperiodic* if it can be constructed by concatenations and superpositions of a shorter string; otherwise the string is *superprimitive*. We say that a quasiperiodic string is covered by a shorter string, while a superprimitive string is covered only by itself. It is perfectly possible that several different strings cover the same quasiperiodic string. For example, *abaab* and *abaabaab* both cover *abaabaabaab*. A superprimitive string α that covers a string γ is a *quasiperiod* of γ. In Lemma 8.1 on page 146 we show that every string is covered by exactly one superprimitive string. Hence, a superprimitive string α that covers a string γ is *the* quasiperiod of γ. For example, the string *abaab* is the quasiperiod of the string *abaabaabaab*. Apostolico, Farach and Iliopoulos in [10] present an algorithm that finds the quasiperiod of a string of length n in time $O(n)$. This algorithm is simplified, and turned into an on-line algorithm, by Breslauer in [28]. Moore and Smyth in [141] present an algorithm that finds all substrings that cover a string of length n in time $O(n)$.

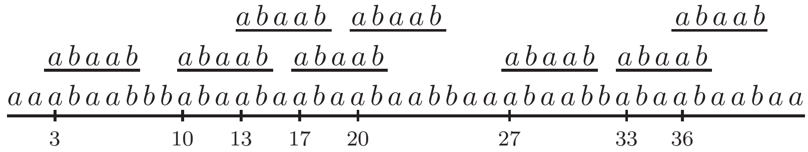


Figure 3.3: A string of length 42 in which the substring from position 10 to 24 is a maximal quasiperiodic substring. To see why, consider the suffix tree of the string, and let u be the node with path-label $L(u) = abaab$ and leaf-list $LL(u) = (3, 10, 13, 17, 20, 27, 33, 36)$. There are four maximal runs in the leaf-list $LL(u)$; $R_1 = (3)$ from 3 to 7, $R_2 = (10, 13, 17, 20)$ from 10 to 24, $R_3 = (27)$ from 27 to 31, and $R_4 = (33, 36)$ from 33 to 40. The run R_2 spans the maximal quasiperiodic substring *abaabaabaabaab*.

Before continuing with our treatment of quasiperiodic strings, we note a small discrepancy in the common classification of strings. Recall that a string is primitive if it cannot be written as α^k for any $k \geq 2$. Based on this definition, and the correspondence between quasiperiodic and superprimitive, one would expect that a periodic string is a string that can be written α^k, for some $k \geq 2$. This is the definition by Gusfield in [74, page 40]. However, most commonly a periodic string is defined as a string that can be written as $\alpha^k\alpha'$, for some $k \geq 2$, where α' is a prefix of α. This common definition of a periodic string implies that a string can be both primitive and periodic. For example, *bcabcabc* is primitive and periodic. This discrepancy is of course not a problem, but it can be confusing if one is unaware of its presence.

Apostolico and Ehrenfeucht in [9] introduced the notion of *maximal quasiperiodic substrings* of a string. A quasiperiodic substring $\gamma = S[i\,..\,j]$, with quasiperiod α, is maximal if two properties hold. Firstly, no extensions of γ are covered by α. Secondly, αa does not cover γa, where $a = S[j{+}1]$ is the character following γ in S. We identify a maximal quasiperiodic substring $\gamma = S[i\,..\,j]$ with quasiperiod α by the triple $(i, j, |\alpha|)$.

For an illustration of maximal quasiperiodic substrings, consider the string S of length 42 shown in Figure 3.3. The substring $S[13\,..\,21] = abaabaab$ is quasiperiodic, but not maximal, because its quasiperiod *abaab* covers the extension $S[10\,..\,24] = abaabaabaabaab$. On the other hand, the substring $S[33\,..\,40] = abbabaab$ is quasiperiodic, but not maximal, because its quasiperiod *abaab* extended with $S[41] = a$ covers $S[33\,..\,41] = abbabaaba$. Finally, the substring $S[10\,..\,24] = abaabaabaabaab$ is maximal quasiperiodic with quasiperiod *abaab*, and it is identified by the triple $(10, 20, 5)$.

A covering of a string γ by a shorter string α implies that γ contains an occurrence of a tandem repeat $\beta\beta$, where β is a prefix of α. For example, in Figure 3.3 the covering of $S[13\,..\,21] = abaabaab$ by *abaab* implies the tandem repeat *abaaba*. Every quasiperiodic string thus contains one or more occurrences of tandem repeats. The result of Thue in [184], which states that arbitrary long square-free strings exist, thus implies that arbitrary long strings which contain no quasiperiodic substrings also exist. Apostolico and Ehrenfeucht in [9] present an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log^2 n)$ and space $O(n \log n)$. In Chapter 8 we present an algorithm that improves this algorithm by finding all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Both algorithms are based on traversals of the suffix tree of the string during which maximal quasiperiodic substrings are detected at nodes with superprimitive path-labels. To present the general idea of the algorithms in further details, we need some additional terminology from Section 8.2.

Let S be the string in which we want to find maximal quasiperiodic substrings. For a node v in the suffix tree $T(S)$, we partition its leaf-list, $LL(v) = (i_1, i_2, \ldots, i_k)$, $i_j < i_{j+1}$ for $1 \leq j < k$, into a sequence of disjoint subsequences, $R_1, R_2, \ldots, R_r$, such that each R_ℓ is a maximal subsequence, $i_a, i_{a+1}, \ldots, i_b$, where $i_{j+1} - i_j \leq |L(v)|$, for $a \leq j < b$. Each R_ℓ is denoted a *run* at v, and represents a maximal substring of S that can be covered by $L(v)$, i.e. $L(v)$ covers $S[\min R_\ell \mathinner{..} |L(v)| - 1 + \max R_\ell]$. We say that R_ℓ is a run from $\min R_\ell$ to $|L(v)| - 1 + \max R_\ell$. A run R_ℓ at v *coalesces* at v if it contains indices from at least two children of v, i.e. if for no child w of v we have that $R_\ell \subseteq LL(w)$. For an example consider Figure 3.3, which illustrates a leaf-list $LL(v)$ that contains four runs. The run $R_2 = (10, 13, 17, 20)$ is a coalescing run because the occurence of $L(v) = abaab$ at position 20 is followed by the character b, while the occurrences of $abaab$ at the other positions in the run are followed by the character a. The run $R_4 = (33, 36)$ is not a coalescing run because the occurrences of $abaab$ at both positions in the run are followed by the same character a, i.e. both positions are contained in the leaf-list at child w of v.

A fundamental connection between the maximal quasiperiodic substrings of S and the suffix tree $T(S)$ is stated by Apostolico and Ehrenfeucht in [9], and restated in Theorem 8.2 on page 148. It says that $(i, j, |\alpha|)$ is a maximal quasiperiodic substring of S, if and only if, there is a non-leaf node v in $T(S)$ with superprimitive path-label α, such that there is a coalescing run from i to $j + |\alpha| - 1$ at v. This yields the high level structure of both our algorithm, and the algorithm by Apostolico and Ehrenfeucht; traverse the suffix tree $T(S)$, compute at each node v a data structure that keeps track of the runs in $LL(v)$, and report the coalescing runs if $L(v)$ is superprimitive.

The algorithm by Apostolico and Ehrenfeucht keeps track of the runs in $LL(v)$ in a two-level data structure, where the first level is a balanced search tree in which each leaf correspond to a run in $LL(v)$, and the second level is a set of balanced search trees, where each search tree corresponds to a leaf in the first level, and stores the positions in the run that corresponds to that leaf. Apostolico and Ehrenfeucht describe how to compute this data structure at each node in $T(S)$ in time $O(n \log^2 n)$ in total. The tricky part of their algorithm is to detect the nodes in the suffix tree that have superprimitive path-labels. They sketch how to do this by inserting additional elements into the data structure that are computed at every node during the traversal. The analysis of the running time of the algorithm is done using the "smaller-half trick", but redoing the analysis using the knowledge of the "extended smaller-half trick" does not yield a better analysis of the running time.

Our algorithm presented in Section 8.5 is based on the general technique for detecting string regularities explained in Section 3.1.3. The algorithm is split into two phases. In the first phase we traverse the suffix tree $T(S)$ in order to identify the nodes that have superprimitive path-labels. In the second phase we report the coalescing runs, i.e. the maximal quasiperiodic substrings, from the nodes that have superprimitive path-labels. In contrast to the algorithm by Apostolico, we do not keep track of the runs in $LL(v)$ explicitely, instead we compute at each node v a height-balanced tree which stores $LL(v)$. The operations on height-balanced trees listed in Theorem 8.3 on page 150 then make it possible to compute the coalescing runs in $LL(v)$ within the time permitted by the "extended smaller-half trick". The first phase, i.e. identifying the nodes that have superprimitive path-labels, is the most difficult part of the algorithm. It is implemented as two traversals of the suffix tree during which we pose and answer questions about the superprimitivity of the path-labels.

A question about the superprimitivity of path-label $L(v)$ is generated at an ancestor node w of v, and posed at an ancestor node u of w. A question about the superprimitivity of $L(v)$ posed at node u is a triple (i, j, v), where $i \in LL(v) \subset LL(u)$, and $j = i + |L(v)| - |L(u)| \in LL(u)$. The answer to a question (i, j, v) posed at node u is true, if and only if, i and j are in the same coalescing run at u. If the answer to the question (i, j, v) posed at u is true, then $L(u)$ covers $L(v)$, i.e. $L(v)$ is quasiperiodic. The idea is to pose and answer enough questions to establish the superprimitivity of the path-label of every node in $T(S)$. This is done in two traversals of $T(S)$. Questions are generated in the first traversal, and answered in the second traversal. The many details are explained in Step 1 and Step 2 in Section 8.5. To reduce the space consumption of our algorithm to $O(n)$, the two traversals must be implemented as two interleaved traversals as explained in Section 8.7.

Recently, and independent of our work, Iliopoulos and Mouchard in [95] have published work on detecting maximal quasiperiodic substrings, which includes an algorithm that finds all maximal quasiperiodic substrings in time $O(n \log n)$. Their algorithm is not based on the suffix tree, but on the partitioning technique used by Crochemore in [41] to detect primitive tandem repeats, combined with a data structure which supposedly is described by Imai and Asano in [97]. Together with the algorithm by Iliopoulos and Mouchard, our algorithm show that finding maximal quasiperiodic substrings can be done in two different ways corresponding to the algorithms by Crochemore [41] and Apostolico and Preparata [11] for finding primitive tandem repeats.

Chapter 4

Prediction of Structure

Nuts!
— Anthony C. McAucliffe, Bastogne, December 22, 1944.

The simplicity, and the amount of information preserved, when describing complex biomolecules as sequences of residues, i.e. strings over finite alphabets, is the foundation of most algorithms in computational biology, including the ones that have been presented in the previous chapters. However, in the real world the biomolecules DNA, RNA, and proteins are not one-dimensional strings, but full three-dimensional structures, e.g. the three-dimensional structure of the double stranded DNA molecule that stores the genetic material of an organism is the famous double helix described by Watson and Crick in [201].

The genetic material of an organism is, as described in Section 1.2, the blueprint for the RNA and protein molecules that participate in the biochemical processes necessary for the organism to live. The functionality of an RNA or protein molecule is the tasks it performs in the organism, that is, the biochemical processes it participates in. It is widely believed that the functionality of a biomolecule is determined primarily by its three-dimensional structure together with a few key residues. Hence, the majority of the residues in an RNA or protein sequence are, by themselves, not very important for the functionality of the biomolecule, and knowledge about the structure of a biomolecules is much more important for deciding its functionality than knowledge about its entire sequence of residues. Knowledge about the functionality of biomolecules is highly valued because it can be used to design new biomolecules with a specific and desired functionality, e.g. to cure or control a disease.

Since the structure of a biomolecule is more important for its functionality than its sequence of residues, related biomolecules that perform similar tasks in different organisms probably have very similar structures while their sequences of residues can be quite different. This implies that it can be useful to include structural information when comparing biomolecules from different organisms. For example by alignment of structures as described in [181].

The structural information about a biomolecule is usually classified in four structural levels. Each structural level can be seen as a stepping stone towards the next structural level and a better understanding of the functionality of the biomolecule. The *primary structure* describes the sequence of residues that forms the biomolecule. For an RNA molecule this is a sequence of nucleotides, and for a protein this is a sequence of amino acids. The *secondary structure* describes structural elements that are important in the formation of the full three-dimensional structure of the biomolecule. For an RNA molecule this is a set of base pairs, and for a protein this is segments of the amino acid sequence that form regular structural patterns such as helices and strands. The *tertiary structure* describes the full three-dimensional structure of the biomolecule. The *quaternary structure* describes how several biomolecules come together and interact to form larger aggregated structures.

To determine experimentally the higher structural levels of an RNA or protein molecule is a difficult and time consuming task. Methods for computational prediction of the higher structural levels of a biomolecule based on its primary structure, or other available information, are thus in demand. In the following sections we review and present such methods. Our guideline is models for structure formation based on energy minimization. In Section 4.2 we focus on RNA secondary structure prediction and the results in our paper *An Improved Algorithm for RNA Secondary Structure Prediction* presented in Chapter 9. In Section 4.3 we focus on protein tertiary structure prediction and the results in our paper *Protein Folding in the 2D HP Model* presented in Chapter 10.

4.1 Free Energy Models

To computationally predict the structure of a biomolecule it is necessary to have a precise model which abstracts the formation of structure in the real world to the desired level of detail. The three-dimensional structure of a biomolecule is not static, but vibrates around an equilibrium called the *native state*. Experiments reported by Anfinsen *et al.* in [6] show that a protein in the real world folds into, i.e. vibrates around, a unique three-dimensional structure, the *native conformation*, independent of its starting conformation. The same is true for an RNA molecule. The *structure prediction problem* is to predict the native conformation of a biomolecule in a model of structure formation.

A quantum mechanical description of the interaction between all the atoms in the biomolecule would be a possible model of structure formation, but the structure prediction problem in such a model would be difficult, if not impossible, to solve. Since biomolecules are part of the real world they are believed to obey the laws of thermodynamics which say that a biomolecule will spend most of its time in a state of least free energy. The most stable structure of a biomolecule, i.e. the native conformation, should thus be a structure of least free energy. Aspects such as kinetics, or interactions with other biomolecules, also influence the structure of a biomolecule, but in the natural environment of the biomolecule the minimization of free energy is believed to be the most important force of structure formation.

The constraints of thermodynamics yields a guideline for constructing models of structure formation in which the structure prediction problem is cast as a minimization problem of the free energy over the possible conformations of the biomolecule. The specification of a model of structure formation following this guideline, hereafter called a *free energy model*, should include the following:

- A model of the biomolecule, that is, an abstraction of the atoms in the biomolecule and the various bonds between them.
- A model of the possible conformations of the abstracted biomolecule, that is, a set of rules describing the legal conformations.
- A computable energy function that assigns a free energy to every legal conformation of the abstracted biomolecule.

The solution to the structure prediction problem in a free energy model is the conformations of the abstracted biomolecule that minimize the free energy function. A conformation that minimizes the free energy function is called a native conformation in the model. The relevance of the predicted native conformation in a model, and the computational resources such as time and space needed to compute it, depend entirely on the choice of model.

If a native conformation of a biomolecule in a model can be computed in polynomial time in the size of the abstraction of the biomolecule, then the model is called *tractable*; otherwise the model is called *intractable.* For a model to be *relevant* it has to reflect some of the properties of structure formation in the real world. An obvious property to reflect is *visual equivalence* between native conformations in the model and native conformations in the real world. For tertiary structure prediction, visual equivalence is to aim for the full three-dimensional structure. For secondary structure prediction, visual equivalence is to aim for the secondary structure which is similar to the secondary structure that can be inferred from the full three-dimensional structure. A more subtle, but still useful, property to reflect is *behavioral equivalence* between structure formation in the model and structure formation in the real world. Tractable and relevant models are of course in demand.

In the following sections we consider models for RNA secondary structure prediction and protein tertiary structure prediction which illustrate how the choice of model influences the relevance and tractability of the structure prediction problem. This exemplifies the interplay between modeling the biological reality and constructing a relevant algorithm mentioned in Chapter 1

4.2 RNA Secondary Structure Prediction

The secondary structure of a biomolecule describes structural elements that are important in the formation of the full three-dimensional structure. Recall that an RNA molecule is a single sequence of nucleotides that consists of a backbone of sugars linked together by phosphates with an amine base, adenine (A), uracil (U), guanine (G) or cytosine (C), attached to each sugar. The key force in the formation of the three-dimensional structure of an RNA molecule is hydrogen bonds, called *base pairs*, between the amine bases. Base pairs are also responsible for the formation of the double helix structure of a double stranded DNA molecule. The double helix structure is stabilized by base pairs between the complementary bases, A and T, and, C and G, on the two complementary

DNA sequences that form the molecule. Base pairs between complementary bases are called Watson-Crick base pairs. In an RNA molecule base pairs between the complementary bases, A and U, and, C and G, are the most common, but other base pairs, most frequently G, U base pairs, are also observed.

The secondary structure of an RNA molecule is the base pairs in the three-dimensional structure of the molecule. Since base pairs are formed between nucleotides, we talk about the secondary structure of an RNA sequence. The secondary structure of an RNA sequence $S \in \{A, G, C, U\}^*$ is a set $\mathcal{S}$ of base pairs $i \cdot j$ with $1 \leq i < j \leq |S|$, such that no base $S[i]$ is paired with more than one other base $S[j]$, that is, $\forall i \cdot j, i' \cdot j' \in \mathcal{S} : i = i' \Leftrightarrow j = j'$.

A secondary structure contains *pseudoknots* if it contains overlapping base pairs $i \cdot j$ and $i' \cdot j'$, where $i < i' < j < j'$. Pseudoknots occur in real world structures of RNA molecules. Recently Rivas and Eddy in [167] presented an algorithm that predicts the secondary structure of an RNA sequence of length n allowing certain pseudoknots in time $O(n^6)$. Most often pseudoknots are ignored in order to allow for faster algorithms which for large classes of RNA sequences still predict biological relevant secondary structures. (Recently we have considered the problem of pseudoknots in [125].)

In the rest of this section we focus on prediction of pseudoknot-free RNA secondary structures. To formulate this problem as a structure prediction problem in a free energy model, we have to specify an energy function that assigns a free energy to every possible pseudoknot-free RNA secondary structure.

From the definition of an RNA secondary structure it seems natural to specify the energy function in terms of the formed base pairs by assigning a free energy to every possible base pair, and state the free energy of a secondary structure as the sum of the free energies of its base pairs. This is somewhat similar to a classical score function for the pairwise alignment problem which assigns a cost to every possible column of an alignment. The algorithm by Nussinov *et al.* in [155] implements this idea by using a free energy function that is minimized when the secondary structure contains the maximum number of complementary base pairs. The algorithm takes time $O(n^3)$ for predicting the secondary structure of an RNA sequence of length n but is generally considered to be too simplistic to give accurate secondary structure predictions.

The most common way to specify an energy function that yields more accurate secondary structure predictions is to specify the free energy of a secondary structure in terms of more complex structural elements than just individual base pairs. The general idea is to consider a pseudoknot-free RNA secondary structure $\mathcal{S}$ as a collection of *loops* and *external* unpaired bases. To define these structural elements we need to introduce some terminology.

If $i < k < j$ and $i \cdot j \in \mathcal{S}$ then we say that k is *accessible* from $i \cdot j \in \mathcal{S}$ if there is no base pair between $i \cdot j$ and k, that is, if there is no base pair $i' \cdot j' \in \mathcal{S}$ such that $i < i' < k < j' < j$. The absence of pseudoknots implies that a base is accessible from at most one base pair. If a base is not accessible from any base pairs it is called an *external base*. For each base pair $i \cdot j \in \mathcal{S}$ we define the *loop* closed by that base pair to consist of all bases that are accessible from it. We say that $i \cdot j$ is the *exterior* base pair of the loop, and that all base pairs $i' \cdot j'$, where i' and j' are accessible from $i \cdot j$, are the *interior* base pairs of the

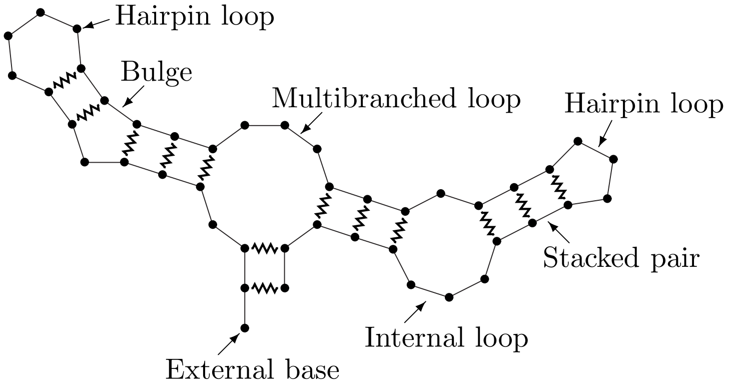


Figure 4.1: An example RNA structure. Bases are depicted by circles, the RNA backbone by straight lines and base pairings by zigzagged lines. This figure is copied from the paper in Chapter 9.

loop. The absence of pseudoknots implies that if i' is accessible from $i \cdot j$, and $i' \cdot j' \in \mathcal{S}$, then $i' \cdot j'$ is an interior base pair of the loop closed by $i \cdot j$.

As illustrated in Figure 4.1 loops are named according to the number of interior base pairs. If there is no interior base pairs then the loop is called a *hairpin loop*. If there is one interior base pair then there are three possibilities to consider; if there are no unpaired bases in the loop then the loop is called a *stacked base pair*; if there are unpaired bases between the exterior and interior base pair only on one side, then the loop is called a *bulge*; finally, if there are unpaired bases between the exterior and interior base pair on both sides, then the loop is called an *internal loop*. If there are more than one interior base pair then the loop is called a *multibranched loop*. The absence of pseudoknots implies that a secondary structure can be decomposed into a collection of loops and external bases that are are disjoint except if the exterior base pair of one loop is also an interior base pairs of another loop, in which case the two loops share a single base pair.

Tinoco *et al.* in [186] propose a model for the free energy of a secondary structure of an RNA molecule that is based on the decomposition of the structure into loops. The model states that the free energy of a secondary structure is the sum of independent energies for each loop in the structure. To specify this free energy function we must specify how to compute the free energy for each type of loop. To get a good secondary structure prediction, the loop specific free energy functions should reflect the biophysics that govern RNA structure formation in the real world. To achieve this, the first step is to decide on the mathematical structure of the free energy functions, and the second step is to determine the proper parameters of the free energy functions. Performing experiments to estimate the energies of the various types of loops in RNA structures is ongoing work. Recent results are published in [134].

Using a loop dependent free energy function, Zuker and Stiegler in [213], and Nussinov and Jacobsen in [154], present a recursive algorithm that finds the minimum free energy of a secondary structure of an RNA sequence S of length n in time $O(n^3)$. An optimal structure of S, i.e. a secondary structure of S with the minimum free energy, can be found by backtracking the recursive computation. The general idea of both algorithms is to find for every substring $S[i\,..\,j]$ the minimum free energy of the secondary structure of the substring that contains the base pair $i \cdot j$. This is done by minimizing, over every possible loop that can be closed by the base pair $i \cdot j$, the sum of the free energy of that loop and the free energies of the optimal structures that are closed by the interior base pairs of that loop.

It is convenient to think of this minimization as being split according to the type of loop being closed by $i \cdot j$, that is, first we find the minimum free energy of a secondary structure of $S[i\,..\,j]$ where base pair $i \cdot j$ close a specific type of loop, then we minimize over the possible types of loops. We say that we for every base pair $i \cdot j$ handle every type of loop. Since there is only a fixed number of types of loops, the running time of the algorithm is determined by the time it takes to handle a specific type of loop. The full algorithm using dynamic programming is outlined in Section 9.2. The algorithm is usually referred to as the mfold algorithm because it forms the basis of the mfold server for RNA secondary structure prediction [214].

The general idea of the mfold algorithm is independent of the loop specific free energy functions, but to obtain the stated running time of $O(n^3)$ certain assumptions are necessary. By counting follows that a base pair $i \cdot j$ can close an exponential number of multibranched loops. Hence, if the free energy function of multibranched loops is such that we have to consider every possible multibranched loop that is closed by a specific base pair in order to determine the loop of minimum free energy, then the running time of the entire mfold algorithm becomes exponential. To avoid this, the free energy of multibranched loops is, cf. Equation 9.5 on page 165, usually specified as a linear combination of the number of unpaired bases in the loop and the number of interior base pairs in the loop, plus a constant term. This makes it possible to handle all multibranched loops closed by the base pair $i \cdot j$ in time $O(j - i)$, which implies that we use time $O(n^3)$ in total to handle all multibranched loops. The bottleneck of the computation then becomes the handling of internal loops. By counting follows that a base pair $i \cdot j$ can close $O((j-i)^2)$ internal loops, which implies that there are $O(n^4)$ possible internal loops to consider in total. To get a total running time of $O(n^3)$ we cannot afford to consider each of these internal loops explicitly. It is therefore often assumed that the size of internal loops, i.e. the number of bases in the loop, is upper bounded by some constant k, which is called the *cutoff size* of internal loops. Since a base pair $i \cdot j$ can close $O(k^2)$ internal loops of size at most k, it is possible to handle all internal loops with size at most k in time $O(k^2 n^2)$. The assumption of a cutoff size thus reduces the total running time of the mfold algorithm to $O(n^3)$.

For certain free energy functions it is possible to remove the restriction on the size of internal loops without increasing the asymptotic running time of the entire algorithm, that is, without making it necessary to consider every one of the $O(n^4)$ possible internal loops. Waterman and Smith in [199] describe how to handle internal loops in time $O(n^3)$ using a free energy function of internal loops that depends only on the size of the loop and the stacking of the exterior and interior base pair. Eppstein *et al.* in [51] describe how to reduce the time to $O(n^2 \log n)$ if the free energy function of internal loops is a convex function of the size of the loop. If the free energy function of internal loops is an affine function of the size of the loop, the time can be reduced to $O(n^2)$ by adapting the technique from the method by Gotoh [67] for pairwise alignment with affine gap cost. Unfortunately none of these free energy functions are sufficiently close to reality to give accurate secondary structure predictions.

Experiments has led to the acceptance of a free energy function for internal loops that is a sum of independent terms contributed by the size of the loop, the base pair stacking of the exterior and interior base pair, and the asymmetry of the loop, that is, the relation between the number of unpaired bases on each side of the exterior and interior base pair. These terms are illustrated in Figure 9.2 on page 166. The asymmetry term implies that we cannot use the method by Waterman and Smith [199] to handle internal loops in time $O(n^3)$. Papanicolaou *et al.* in [156] propose that the asymmetry term should be given by a Ninio type asymmetry function cf. Equation 9.15 on page 171, which is the most commonly used type of asymmetry functions in practice.

In Chapter 9 we describe how to handle internal loops with cutoff size k using a free energy function cf. Figure 9.2, and a Ninio type asymmetry function, in time $O(kn^2)$. By setting the cutoff size to n this reduces the time it takes to handle unrestricted internal loops using a realistic free energy function to $O(n^3)$. The general idea is to observe that if the number of unpaired bases on each side of the two base pairs in an internal loop is larger than some small constant, then the asymmetry term given by an asymmetry function of the Ninio type only depends on the lopsidedness of the internal loop, that is, it only depends on the difference between the number of unpaired bases on each side of the exterior and interior base pair. This observation makes it possible to determine the optimal interior base pair of most equal sized internal loops closed by a specific base pair $i \cdot j$ in constant time, which implies the reduced running time. The details are in Section 9.3. The method can also be used to improve the handling of internal loops when computing the full equilibrium partition function by the method described by McCaskill in [136]. The details are in Section 9.3.3.

In Section 9.5 we describe several experiments that examine if removing the upper bound on the size of internal loops improves the quality of the predicted secondary structures. In the first experiment we construct an artificial RNA sequence with a secondary structure that cannot be predicted properly with an upper bound on the size of internal loops. In the second experiment we examine a real RNA sequence for which an upper bound on the size of internal loops seems to influence proper secondary structure prediction, when the sequence is folded at high temperatures, where “temperature” is modeled by the parameters of the free energy function. The two experiments indicate that a cutoff size of 30 (the most commonly used in practice) is reasonable for most practical applications. However, our method removes any speculations about a reasonable cutoff size without increasing the asymptotic running time.

Free energy minimization methods to predict the secondary structure of an RNA molecule based on its primary structure are useful, but unfortunately they seldom predict the true secondary structure. Mathews *et al.* in [134] report that on average only 73 percent of the base pairs in the true secondary structure are found in a secondary structure predicted by free energy minimization methods, while a secondary structure that contains more true base pairs is usually found among the structures with energy close to the minimum free energy. Zuker in [211] presents a method for finding suboptimal secondary structures of an RNA sequence which can be used to search for these structures.

If related RNA sequences are available, the fact that structure similarity is more conserved than sequence similarity can be used to improve the quality of the structure prediction. One approach is to construct the secondary structure from the parsing of a stochastic context-free grammar which parameters are estimated from a multiple alignment of the related RNA sequences, e.g. [104, 172]. Another approach is to combine the alignment of the related RNA sequences with the prediction of a consensus secondary structure, e.g. [66, 50].

4.3 Protein Tertiary Structure Prediction

The structural elements that are important in the formation of the three-dimensional structure of a protein, i.e. the secondary structure of a protein, are segments of consecutive amino acids forming regular structural patterns called α-helices and β-strands. An α-helix is a winding spiral that stabilizes by forming hydrogen bonds between neighboring turns, and a β-strand is a straight segment of the amino acids sequence that stabilizes by forming hydrogen bonds between neighboring β-strands that run in parallel forming β-sheets.

Predicting protein secondary structure is usually addressed by pattern recognition methods such as neural networks [164, 168] or hidden Markov models [15]. The general idea when using hidden Markov models to predict protein secondary structure is to construct a model in which the most likely sequence of states that generates a given amino acid sequence can be interpreted as deciding for each amino acid whether it is located on a α-helix or on a β-strand. Knowledge about the secondary structure of a protein is useful, but knowing its tertiary structure, i.e. the three-dimensional structure of the native state of the protein, is much more useful. The tertiary structure contains the full description of the secondary structure, and is an important step towards understanding the quaternary structure, and hopefully the functionality of the protein.

4.3.1 Modeling Protein Structure

To computationally predict the structure of a protein using a free energy model, we must decide how to model the protein, how to model the possible conformations of the protein, and how to model the free energy of a conformation. These decisions influence the level of detail of the predicted protein structure. Figure 4.2 shows three possible levels of detail. In the following we will consider possible free energy models for protein structure prediction which gradually decrease the level of detail of the predicted protein structure.

In chemistry a molecule is usually viewed as a collection of atoms connected by bonds. Using this as the model for proteins, we can specify the tertiary struc-

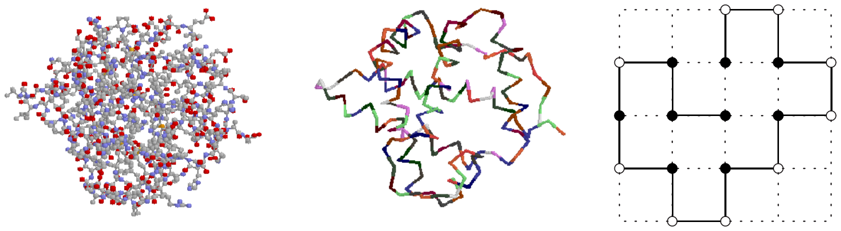


Figure 4.2: The two leftmost figures shows the structure of a hemoglobin protein from the same angle but with a different level of detail. In the figure to the left all atoms except hydrogen atoms are present. In the middle figure only the backbone of the protein is shown. The figure to the right is an example of a structure in the two-dimensional hydrophobic-hydrophilic lattice model.

ture of a protein by stating the angle, the length, and the torsion of every bond in its structure. This is a very complex description that involves information about every atom in the protein. To reduce the complexity of the description some atoms can be omitted, or some atoms can be grouped together into larger units which are treated as single atoms in the model. Such reductions affect the level of detail by reducing the visual equivalence between native conformations in the model and native conformations in the real world.

A model with a detailed description of the protein structure which involves information about individual atoms, is often referred to as an *analytic model*. The free energy function in an analytic model is most often specified by terms contributed by bonded and non-bonded atoms. For bonded atoms the terms depend on the bond lengths, the bond angles, and the bond torsions. For non-bonded atoms the terms depend on physical principles such as Coulombic and van der Walls forces, or statistical information inferred from known structures, such as mean force potentials [174]. The detailed description of the protein structure, and the many parameters of the free energy function, hint that the structure prediction problem in an analytic model is computationally hard. It is therefore not surprising that Ngo and Marks in [150] prove that the structure prediction problem in a typical analytic model is NP hard.

One way to reduce the complexity of an analytic model is to limit the bond lengths, angles and torsions to fixed sets of legal values. This makes it possible to enumerate all legal conformations of a protein, and, if time permits, to solve the structure prediction problem by considering each of the finitely many legal conformations of a protein. Pedersen and Moult in [160] suggest that he sets of legal values should be compiled from known protein structures.

Another approach to protein structure prediction is the *threading problem* suggested by Jones, Taylor and Thornton in [100]. This approach can be thought of as an analytic model in which bond lengths, angles and torsions are limited to those occurring in a fixed set of known structures. The general idea of threading is to compare a protein with unknown structure against a set of known structures, and to predict its structure as the known structure that fits best according to some energy function. The legal conformations of a protein is thus limited to a set of known structures. The name threading follows from the mental picture of threading the amino acids of the protein with unknown structure onto every one of the known structures in order to determine the known structure that fits best. Threading is motivated by the belief that the number of essential different protein structures is small compared to the number of different proteins [37], which should make it possible to collect a representative collection of known protein structures to compare against. Treading methods using branch-and-bound techniques [115] and genetic algorithms [206] have been used successfully. A polynomial time algorithm has been proposed for a specific class of threading problems [205], but the general threading problem has been proven NP complete by Lathrop in [114].

For computational and modeling purposes it might be desirable to limit the legal bond lengths, angles and torsions to the point where legal conformations of a protein are reduced to embeddings of its atoms into a lattice. A model with this property is often referred to as a *lattice model.* A lattice model is most often subject to further simplifications than an analytic model with similar bounds on legal bond lengths, angles and torsions. Typically the protein is modeled only as a sequence of bonded amino acids, the legal conformations limited to self-avoiding embeddings of the sequence of amino acids into a lattice, where bonded amino acids are required to occupy neighboring lattice points, and the free energy of a conformation specified only in terms of the pairs of non-bonded amino acids that occupy neighboring lattice points. In short, lattice models aim to simplify analytic models by avoiding the atomic details.

The most widely used type of lattice is the two- or three-dimensional square lattice. The rightmost figure in Figure 4.2 shows a conformation of a protein in a two-dimensional square lattice. It is obvious that there is little visual equivalence between the native conformation in a square lattice model and the native conformation in the real world. But Dill *et al.* in [45] describe experiments that support some behavioral equivalence between the structure formation process in square lattice models and the structure formation process in the real world. The behavioral equivalence is also addressed by Sali *et al.* in [170], who based on experiments using square lattice models suggest that only proteins for which the structure of minimum free energy has significantly less free energy than other structures of the protein, can be expected to fold into the structure of minimum free energy. Lattice models have become popular because of their simplicity which allows the structure prediction problem to be solved by considering each of the finitely many legal conformations of a protein. This, of course, is only feasible for small proteins, but still useful for studying the behavioral equivalences. The structure prediction problem has been proven NP complete in several square lattice models, e.g. [57, 157, 190, 26, 40].

The hydrophobic-hydrophilic model proposed by Dill in [44] is one of the simplest, but most popular square lattice models. It models the belief that a major contribution to the free energy of the native conformation of a protein is due to interactions between hydrophobic amino acids, which tend to form a core in the spatial structure that is shielded from the surrounding solvent by hydrophilic amino acids. The model is called the HP model, where H stands for hydrophobic, and P stands for polar. In the HP model a protein is abstracted as a sequence of amino acids where each amino acid is classified as being either hydrophobic or hydrophilic. If we use "0" to denote a hydrophilic amino acid, and "1" to denote a hydrophobic amino acid, then we can describe this abstraction of a protein as a string over the alphabet $\{0, 1\}$. A legal conformation, or folding, of a protein in the HP model is a self-avoiding embedding of its abstraction as a string into the two- or three-dimensional square lattice such that adjacent characters in the string, i.e. adjacent amino acids in the sequence, occupy adjacent lattice points. The free energy of a conformation is proportional to the number of non-local hydrophobic bonds. A non-local hydrophobic bond is a pair of non-adjacent hydrophobic amino acids that occupy adjacent lattice points. The HP model is described in more details in Section 10.2.

The rightmost figure in Figure 4.2 shows a conformation in the 2D HP model with nine non-local hydrophobic bonds. Several heuristics have been applied to predict the native conformation of a protein in the HP model, e.g. [191, 207], but most interestingly from our point of view is that the HP model was the first reasonable model for protein structure formation in which approximation algorithms for the protein structure prediction problem were formulated. This was done by Hart and Istrail in [80]. For a while it was even believed that the protein structure prediction problem in the HP model would be solvable in polynomial time, but recently it has been shown NP complete in [26, 40].

The hardness of the protein structure prediction problem, even in very simple free energy models such as the two-dimensional HP model, makes it apparent that in order to computationally investigate protein structure formation one has to think about heuristics, approximation algorithms, or other modeling approaches. If someone claims that an efficient method for predicting the atomic details of a protein structure will be available soon, he or she certainly risks being classified according to the quote beginning this chapter.

4.3.2 Approximating Protein Structure

An approximation algorithm is an algorithm that gives solutions guaranteed close to the optimal solution. Besides the algorithmic challenges, the motivation for studying approximation algorithms for the protein structure prediction problem is very well captured by Ngo *et al.* in [151, page 46]:

> "An approximation algorithm (for the protein structure prediction problem in any reasonable model) might be of significant practical use in protein-structure prediction, because exactness is not an absolute requirement. If the guaranteed error bound were sufficiently small, an approximation algorithm might be useful for generating crude structures that, though not necessarily folded correctly, are close enough to the native structure to be useful. If not, merely knowing that the energy of the optimal structure is below a certain threshold (by running the approximation algorithm) could still be of use as part of a larger scheme."

The first approximation algorithms for the protein structure prediction problem were formulated in the HP model by Hart and Istrail in [80]. As explained in Section 10.2, the free energy of a conformation in the HP model is proportional to the number of non-local hydrophobic bonds. We say that the number of non-local hydrophobic bonds in a conformation is the score of the conformation. Let $\mathrm{OPT}(S)$ be the optimal, i.e. maximum, number of non-local hydrophobic bonds possible in a conformation of $S \in \{0,1\}^*$ in the HP model. A conformation of S with this maximum number of non-local hydrophobic bonds is a native conformation of S in the 2D HP model.

An approximation algorithm for the structure prediction problem in the HP model should find a conformation of S with score, $\mathcal{A}(S)$, that is guaranteed close to the optimal score $\mathrm{OPT}(S)$. We focus on the case where the guarantee is given by an *approximation ratio* that specifies the possible deviation from the optimal solution by a multiplicative constant. We say that an approximation ratio α is *absolute* if $\forall S \in \{0,1\}^* : \mathcal{A}(S) \geq \alpha \cdot \mathrm{OPT}(S)$. Sometimes an absolute approximation ratio cannot be guarantee because of an additive term that becomes negligible small to $\mathrm{OPT}(S)$ when $\mathrm{OPT}(S)$ becomes large. To handle this situation we say that an approximation ratio α is *asymptotic* if $\forall \epsilon > 0 \, \exists k \, \forall S \in \{0,1\}^* : \mathrm{OPT}(S) \geq k \Rightarrow \mathrm{A}(S) \geq (\alpha - \epsilon) \cdot \mathrm{OPT}(S)$.

Hart and Istrail in [80] present an approximation algorithm for the structure prediction problem in the 2D HP model with an absolute approximation ratio of 1/4, and an approximation algorithm for the structure prediction problem in the 3D HP model with an asymptotic approximation ration of 3/8. The running time of both algorithms is $O(n)$, where n is the length of the string to fold. The general idea of both algorithms is the same. The better approximation ratio in the 3D HP model follows because of the additional degree of freedom in the 3D square lattice compared to the 2D square lattice. In the following we focus on the algorithm in the 2D HP model and some possible improvements.

The approximation algorithm in the 2D HP model is based on dividing the hydrophobic amino acids in S, i.e. the 1's, into two sets depending on their position: $\mathrm{EVEN}(S)$ is the set of even-indexed positions in S that contain a hydrophobic amino acid, and $\mathrm{ODD}(S)$ is the set of odd-indexed positions in S that contain a hydrophobic amino acid. Hart and Istrail show that $2 \cdot \min\{|\mathrm{EVEN}(S)|, |\mathrm{ODD}(S)|\} + 2$ is an upper bound on $\mathrm{OPT}(S)$, and that there is a position in S that splits S into two parts, such that the one part contains at least half of the positions in $\mathrm{EVEN}(S)$, and the other part contains at least half of the positions in $\mathrm{ODD}(S)$. They construct a folding of S by bending it at that position such that the two parts of S are placed opposite of each other in the 2D square lattice forming the two stems of an "U". The two stems are contracted by loops such that every other lattice point along the face of each contracted stem is occupied by an hydrophobic amino acid. The contraction is performed such that these hydrophobic amino acids occupy positions in S that are either totally contained in $\mathrm{EVEN}(S)$, or totally contained in $\mathrm{ODD}(S)$, depending on the stem. The resulting fold is an U-fold such as illustrated in Figure 10.2 on page 185, where at least $\min\{|\mathrm{EVEN}(S)|, |\mathrm{ODD}(S)|\}/2$ non-local hydrophobic bonds are formed between the contracted stems. The upper bound on $\mathrm{OPT}(S)$ implies an asymptotic approximation ratio of 1/4. By being a little bit more careful it is possible to achieve an absolute approximation ratio of 1/4.

In Chapter 10 we present three attempts to improve on the approximation ratio of the Hart and Istrail folding algorithm in the 2D HP model. The first attempt is based on the observation that the Hart and Istrail folding algorithm only attempts to maximize the number of non-local hydrophobic bonds between the two contracted stems under the restriction that all non-local bonded hydrophobic amino acids on each contracted stem must occupy positions in S that have indices of equal parity. It seems like an obvious improvement to remove this restriction, and simply compute the U-fold that has the maximum number of non-local hydrophobic bonds between the two contracted stems. As explained in Section 10.3 this can be done in time $O(n^2)$, but as illustrated in Figure 10.4 on page 187 it does not improve on the approximation ratio.

Another way to try to improve on the approximation ratio is to consider a larger set of legal conformations than U-folds. Figure 10.5 on page 187 shows two such generalizations that we consider in Section 10.3. The first generalization, S-folds, is to allow multiple bends combined with loops on the outer stems. The second generalization, C-folds, is to allow two bends that fold the two ends of the string towards each other combined with loops to contract the two stems. Computing an S- or C-fold that has the maximum number of non-local hydrophobic bonds between the stems can be done in time $O(n^3)$ by using dynamic programming. The S-fold generalization does not improve on the approximation ratio. The C-fold generalization is more tricky. We have not been able to prove, or disprove, that the approximation ratio of the C-fold generalization is better than 1/4, but in Section 10.4 we present a transformation of the folding problem in the 2D HP model into a circular matching problem by which we can prove that the approximation ratio of the C-fold generalization is at most 1/3. Experiments based on the circular matching problem suggest that the approximation ratio of the C-fold generalization is indeed 1/3.

The C-fold generalization is also mentioned by Hart and Istrail in [80]. They present an algorithm that computes a C-fold with the maximum number of non-local hydrophobic bonds between the stems under the restriction that all non-local bonded hydrophobic amino acids on each contracted stem must occupy positions in S that have indices of equal parity. Under this restriction they can prove that the C-fold generalization does not improve on the approximation ratio of 1/4. Mauri, Pavesi and Piccolboni in [135] present an approximation algorithm for the structure prediction problem in the 2D HP model formulated in terms of a stochastic context-free grammar for $\{0, 1\}^*$. The grammar is constructed such that the most probable parsing can be interpreted as a C-fold with the maximum number of non-local hydrophobic bonds. Using a standard parsing algorithm they can find this C-fold in time $O(n^3)$. Based on experiments they conjecture that their folding algorithm has an approximation ratio of 3/8. Our 1/3 upper bound does not disprove their conjecture because our upper bound holds for C-folds where only non-local hydrophobic bonds that occur between the stems are counted. However, we believe that the upper bound also holds for C-folds where all non-local hydrophobic bonds are counted.

Whether or not the presented approximation algorithms for the structure prediction problem in the 2D HP model can be used as suggested by the quote from Ngo *et al.* [151, page 46] at the beginning of this section is not clear. A worst case approximation ratio between 1/4 and 1/3 is not spectacular. It is however perfectly possible that an approximation algorithm performs better one the average than its stated worst case approximation ratio. However, all the folding algorithms presented in this section attempt to make only one non-local hydrophobic bond for each hydrophobic amino acid instead of the maximum of two, on the average one should thus not expect a predicted structure to contain much more than half of the optimal number of non-local hydrophobic bonds. Whether or not this is sufficient to generate structures that are close enough to an optimal conformation to be useful is not clear. Hart and Istrail in [80] argue that the structures that are generated by their folding algorithms resemble the molten globule state, which is an intermediate state on the path towards the native state that is observed in real protein structure formation. Whether or not this is true is questionable. To what extend the presented approximation algorithms for the structure prediction problem in the 2D HP model are biological relevant is thus debatable. Hart and Istrail in [79, 81] describe how to transform foldings in the HP model into foldings in more complex lattice models. This may, or may not, increase the biological relevance of approximation algorithms in the HP model. But no matter what, the problem of formulating anything better than a 1/4 approximation algorithm for the structure prediction problem in the simple 2D HP model is certainly yet another witness of the challenges that face everyone who is interested in protein structure prediction.

Part II

Papers

Chapter 5

Comparison of Coding DNA

The paper *Comparison of Coding DNA* presented in this chapter has been published in part as a technical report [158] and a conference paper [159].

[158] C. N. S. Pedersen, R. B. Lyngsø, and J. Hein. Comparison of coding DNA. Technical Report RS-98-3, BRICS, January 1998.

[159] C. N. S. Pedersen, R. B. Lyngsø, and J. Hein. Comparison of coding DNA. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 153–173, 1998.

Except for minor typographical changes and the addition of the appendix after Section 5.7, the content of this chapter is equal to the conference paper [159]. An implementation of the alignment method presented in this chapter is available at `www.daimi.au.dk/∼cstorm/combat`.

Comparison of Coding DNA

Christian N. S. Pedersen[*] Rune B. Lyngsø[†] Jotun Hein[‡]

Abstract

We discuss a model for the evolutionary distance between two coding DNA sequences which specializes to the DNA/Protein model previously proposed by Hein. We discuss the DNA/Protein model in details and present a quadratic time algorithm that computes an optimal alignment of two coding DNA sequences in the model under the assumption of affine gap cost. We believe that the constant factor of the quadratic running time is sufficiently small to make the algorithm usable in practice.

5.1 Introduction

A straightforward model of the evolutionary distance between two coding DNA sequences is to ignore the encoded proteins and compute the distance in some evolutionary model of DNA. We say that such a model is a DNA level model. The evolutionary distance between two sequences in a DNA level model can most often be formulated as a classical alignment problem and be efficiently computed using a dynamic programming approach, e.g. [149, 171, 173, 196].

It is well known that proteins evolve slower than its coding DNA, so it is usually more reliable to describe the evolutionary distance based on a comparison of the encoded proteins rather than on a comparison of the coding DNA itself. Hence, most often the evolutionary distance between two coding DNA sequences is modeled in terms of amino acid events, such as substitution of a single amino acid and insertion-deletion of consecutive amino acids, necessary to transform the one encoded protein into the other encoded protein. We say that such a model is a protein level model. The evolutionary distance between two coding DNA sequences in a protein level model can most often be formulated as a classical alignment problem of the two encoded proteins. Even though a protein level model is usually more reliable than a DNA level model, it falls short because it postulates that all insertions and deletions on the DNA occur at codon boundaries and because it ignores similarities on the DNA level.

[*] Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `cstorm@brics.dk`.

[†] Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `rlyngsoe@daimi.au.dk`.

[‡] Department of Ecology and Genetics, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `jotun@pop.bio.aau.dk`

In this paper we present a model of the evolutionary distance between two coding DNA sequences in which a nucleotide event is penalized by the change it induces on the DNA as well as on the encoded protein. The model is a natural combination of a DNA level model and a protein level model. The DNA/Protein model introduced by Hein in [83, 85] is a biological reasonable instance of the general model in which the evolution of coding DNA is idealized to involve only substitution of a single nucleotide and insertion-deletion of a multiple of three nucleotides. Hein in [83, 85] presents an $\mathrm{O}(n^2m^2)$ time algorithm for computing the evolutionary distance in the DNA/Protein model between two sequences of length n and m. This algorithm assumes certain properties of the cost function. We discuss these properties and present an $\mathrm{O}(nm)$ time algorithm that solves the same problem under the assumption of affine gap cost.

The practicality of an algorithm not only depends on the asymptotic running time but also on the constant factor hidden by the use of O-notation. To determine the distance between two sequences of length n and m our algorithm computes $400nm$ table entries. Each computation involves a few additions, table lookups and comparisons. We believe the constant factor is sufficiently small to make the algorithm usable in practice.

The problem of comparing coding DNA is also discussed by Arvestad in [14] and by Hua, Jiang and Wu in [89]. The models discussed in these papers are inspired by the DNA/Protein model, but differ in the interpretation of gap cost. A heuristic algorithm for solving the alignment problem in the DNA/Protein model is described by Hein in [84]. A related problem of how to compare a coding DNA sequence with a protein has been discussed in [161, 208].

The rest of this paper is organized as follows: In Section 5.2 we introduce and discuss the DNA/Protein model. In Section 5.3 we describe how to determine the cost of an alignment. In Section 5.4 we present the simple alignment algorithm of Hein [83]. In Section 5.5 we present a quadratic time alignment algorithm. Finally, in Section 5.7 we discuss future work.

5.2 The DNA/Protein Model

A DNA sequence can be described as a string over the alphabet $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, where each symbol represents one of the four possible nucleotides. We will use $a = a_1a_2a_3 \ldots a_{3n-2}a_{3n-1}a_{3n}$ to denote a sequence of coding DNA of length $3n$ with a reading frame starting at nucleotide a_1. We will use $a^i_1a^i_2a^i_3$ to denote the ith codon $a_{3i-2}a_{3i-1}a_{3i}$ in a, and A_i to denote the amino acid encoded by this codon. The amino acid sequence $A = A_1A_2 \ldots A_n$ thus describes the protein encoded by a. Similarly we will use $b = b_1b_2b_3 \ldots b_{3m-2}b_{3m-1}b_{3m}$, $b^i_1b^i_2b^i_3$ and $B = B_1B_2 \ldots B_m$ to denote another sequence of coding DNA.

5.2.1 A General Model

We want to quantify the evolutionary distance between two sequences of coding DNA. According to the parsimony principle, it is biological reasonable to define the distance as the minimum cost of a sequence of evolutionary events that transforms the one DNA sequence into the other DNA sequence. An evolutionary event e on the coding DNA that transforms a to a' will also change the encoded protein from A to A'. Since the same amino acid is encoded by several different codons, the proteins A and A' can be identical.

When assigning a cost to an evolutionary event, it should reflect the changes on the DNA as well as the changes on the encoded protein implied by the event. We will define the cost of an event e that change a to a' and A to A' as the sum of its DNA level cost and its protein level cost, that is

$$\mathit{cost}(a \xrightarrow{e} a') = \mathit{cost}_d(a \xrightarrow{e} a') + \mathit{cost}_p(A \xrightarrow{e} A') \,. \tag{5.1}$$

We have assumed that the total cost of an evolutionary event is the sum of its DNA level cost and its protein level cost, but other combination of functions are of course also possible. We will define the cost of a sequence E of evolutionary events $e_1, e_2, \ldots, e_k$ that transforms $a^{(0)} \xrightarrow{e_1} a^{(1)} \xrightarrow{e_2} a^{(2)} \xrightarrow{e_3} \cdots \xrightarrow{e_k} a^{(k)}$ as some function of the costs of each event. Throughout this paper, we will assume that the function is the sum of the costs of each event, that is

$$\mathit{cost}(a^{(0)} \xrightarrow{E} a^{(k)}) = \sum_{i=1}^{k} \mathit{cost}(a^{(i-1)} \xrightarrow{e_i} a^{(i)}) \,. \tag{5.2}$$

According to the parsimony principle, we define the evolutionary distance between two coding sequences of DNA a and b as the minimum cost of a sequence of evolutionary events that transforms a to b, that is

$$\mathit{dist}(a,b) = \min\{\mathit{cost}(a \xrightarrow{E} b) \mid E \text{ is a sequence of events}\} \,. \tag{5.3}$$

This is a general model of the evolutionary distance between two sequences of coding DNA, to compute $\mathit{dist}(a,b)$ we have to specify a set of allowed evolutionary events and define the cost of each event on the DNA and protein level. The choice of allowed events and their cost influences both the biological relevance of the measure and the complexity of computing the distance.

5.2.2 A Specific Model

The DNA/Protein model introduced by Hein in [83] can be described as an instance of the above general model, where the evolution of coding DNA is idealized to involve only substitutions of single nucleotides and insertion-deletions of a multiple of three consecutive nucleotides. The reason for restricting the insertion-deletion lengths is because an insertion or deletion of a length not divisible by three will change the reading frame and cause a frame shift. Figure 5.1 shows that a frame shift can change the entire encoded amino acid sequence. Since frame shifts are believed to be rare in coding regions, ignoring them is not an unreasonable assumption.

The DNA level cost of an event is defined in the classical way by specifying a substitution cost and a gap cost. More precisely, the DNA level cost of substituting a nucleotide σ with another nucleotide σ' is $c_d(\sigma, \sigma')$, for some metric c_d on nucleotides, and the cost of inserting or deleting $3k$ consecutive

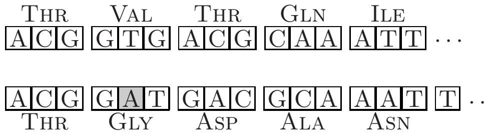


Figure 5.1: An insertion or deletion of a number of nucleotides that is not divisible by three changes the reading frame and causes a frame shift.

$$\begin{bmatrix} T & T & G & C & T & - & - & - & C \\ T & - & - & - & C & A & T & G & C \end{bmatrix}$$

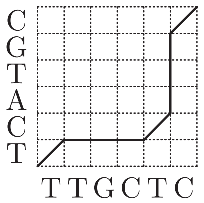


Figure 5.2: An alignment can be described by a matrix or a path in the alignment graph. The above alignment describes three matches and two gaps of combined length six.

nucleotides is $g_d(3k)$, for some sub-additive[1] gap cost function $g_d : \mathbb{N} \rightarrow \mathbb{R}^+$. The protein level cost of an event that changes the encoded protein from A to A' is defined to reflect the difference between the proteins A and A'. More precisely, as the cost $dist_p(A, A')$ of an optimal distance alignment of A and A', where the substitution cost is given by a metric c_p on amino acids, and the gap cost is given by a sub-additive gap cost function $g_p : \mathbb{N} \rightarrow \mathbb{R}^+$. Additional restrictions on the protein level cost will be given in Section 5.2.3.

Except for the restriction on insertion-deletion lengths, the DNA/Protein model allows the traditional set of symbol based nucleotide events. This allows us to use the notion of an alignment. An alignment of two sequences describes a set of substitution or insertion-deletion events necessary to transform one of the sequences into the other sequence. As illustrated in Figure 5.2, an alignment is most often described by a matrix or as a path in a grid-graph. We will define the cost of an alignment as the optimal cost of a sequence of the events described by the alignment. The evolutionary distance $dist(a, b)$ between two coding DNA sequences‘a and b in the DNA/Protein model is thus the cost of an optimal alignment of a and b in the DNA/Protein model. In the rest of this paper we will address the problem of computing the cost of an optimal alignment of a and b in the DNA/Protein model.

If the cost of any sequence of events is independent of the order but only depends on the set of events, an optimal alignment can be computed efficiently using dynamic programming, e.g. [149, 171, 173, 196]. In the DNA/Protein model the cost of an event is the sum of the DNA level cost and the protein level cost. The DNA level cost of a sequence of events is independent of the order the events, but that the protein level cost of a sequence of event depends on the order of the events, see [83, Figure 2] for further details. This implies that we

[1] A function is sub-additive if $f(i + j) \leq f(i) + f(j)$. A sub-additive gap cost function implies that an insertion-deletion of a block of nucleotides is best explained as a single event.

$$\begin{bmatrix} A_1\, A_2 \cdots A_{i-1} & A_i & A_{i+1} \cdots & A_n \\ A_1\, A_2 \cdots A_{i-1} & A'_i & A_{i+1} \cdots & A_n \end{bmatrix}$$

(a) A substitution in the ith codon. The cost is $c_p(A_i, A'_i)$.

$$\begin{bmatrix} A_1\, A_2 \cdots A_{i-1}\, A_i & A_{i+1} \cdots A_{i+k} & A_{i+k+1} \cdots A_n \\ A_1\, A_2 \cdots A_{i-1}\, A_i & - \quad \cdots \quad - & A_{i+k+1} \cdots A_n \end{bmatrix}$$

(b) An insertion-deletion of $3k$ nucleotides affecting exactly k codons. The cost is $g_p(k)$.

$$\begin{bmatrix} A_1\, A_2 \cdots A_{i-1} & A_i \cdots A_{j-1}\, A_j\, A_{j+1} \cdots A_{i+k} & A_{i+k+1} \cdots A_n \\ A_1\, A_2 \cdots A_{i-1} & - \;\cdots\; - \quad \upsilon \quad - \;\cdots\; - & A_{i+k+1} \cdots A_n \end{bmatrix}$$

(c) An insertion-deletion of $3k$ nucleotides affecting $k+1$ codons. The remaining amino acid υ is matched with one of the amino acids affected by the deletion. The cost is $\min_{j=0,1,\ldots,k}\{g_p(j) + c_p(A_{i+j}, \upsilon) + g_p(k-j)\}$.

Figure 5.3: The protein level cost of a nucleotide event can be determined by considering only the amino acids affected by the event.

cannot use one of the classical alignment algorithms to compute an optimal alignment in the DNA/Protein model. To construct an efficient alignment algorithm, we have to examine the protein level cost in more details.

5.2.3 Restricting the Protein Level Cost

A single nucleotide event affects nucleotides in one or more consecutive codons. We observe that since a nucleotide event in the DNA/Protein model cannot change the reading frame, only amino acids encoded by the affected codons are affected by the nucleotide event. A single nucleotide event thus changes protein $A = UXV$ to protein $A' = UX'V$, where X and X' are the amino acids affected by nucleotide event. Hein in [83] implicitly assumes that $dist_p(A, A')$ is the cost of a distance alignment of X and X' that describes the minimum number of insertion-deletions. This assumption implies that $dist_p(A, A')$ is the cost of the minimum cost alignment among the alignments shown in Figure 5.3.

To compute $dist_p(A, A')$, it is thus sufficient to search for the minimum cost alignment among the alignments in Figure 5.3. This reduction of alignments to consider when computing $dist_p(A, A')$ is essential to the alignment algorithms in Sections 5.4 and 5.5. However, if the minimum cost alignment of A and A' among the alignments in Figure 5.3 is not a global minimum cost alignment of A and A', then the assumption conflicts with the definition of $dist_p(A, A')$ as the minimum cost of an alignment of A and A'. Lemma 5.1 states restrictions on the protein level substitution and gap cost for avoiding this conflict. In the special case where the protein level gap cost function is affine, that is $g_p(k) = \alpha_p + \beta_p k,$, for some $\alpha_p, \beta_p \geq 0$ (and define $g_p(0)$ to be zero), the restrictions in Lemma 5.1 become $c_p(\sigma, \tau) + \alpha_p + \beta_p k \leq 2\alpha_p + \beta_p(k + 2l)$, for any amino acids σ, τ, and all lengths $0 < l \leq n - k$. This simplifies to $c_p(\sigma, \tau) \leq \alpha_p + 2\beta_p$ for all amino acids σ and τ, which is biological reasonable because insertions and deletions are rare compared to substitutions.

Lemma 5.1 *Assume a nucleotide event changes $A = UXV$ to $A' = UX'V$. Let $n = |A|$ and $k = \big||A| - |A'|\big|$. If there for any amino acids σ, τ and for all $0 < l \leq n - k$ exists $0 \leq j \leq k$ such that $c_p(\sigma, \tau) + g_p(j) + g_p(k - j) \leq g_p(l) + g_p(l + k)$, then $dist_p(A, A')$ is the cost of an alignment describing exactly k insertions or deletions. Furthermore $dist_p(A, A')$ only depends on X and X'.*

Proof. First observe that the alignments in Figure 5.3 describe the minimum number of insertion-deletions, and that their costs only depend on the sub-alignments of X and X' illustrated by the shaded parts. We will argue that the assumption on c_p and g_p stated in the lemma implies that $dist_p(A, A')$ is the cost of one of the alignments in Figure 5.3. We argue depending on the event that has transformed A to A'. Since $dist_p(A, A')$ is equal to $dist_p(A', A)$, the cost of an insertion transforming A to A' is equal to the cost of a deletion transforming A' to A. We thus only consider substitutions and deletions.

A substitution of a nucleotide in the ith codon of A transforms A_i to A'_i. The alignment in Figure 5.3(a) describes no insertion-deletions and has cost $c_p(A_i, A'_i)$. Any other alignment of A and A' must describe an equal number of insertions and deletions, so by sub-additivity of g_p the cost is at least $2g_p(l)$ for some $0 < l \leq n$. The assumption in the lemma implies that $c_p(A_i, A'_i) \leq 2g_p(l)$ for any $0 < l \leq n$. The alignment in Figure 5.3(a) is thus optimal and the protein level cost of the substitution is $c_p(A_i, A'_i)$.

A deletion of $3k$ nucleotides affecting k codons transforms $A = A_1 A_2 \cdots A_n$ to $A' = A_1 A_2 \cdots A_i A_{i+k+1} A_{i+k+2} \cdots A_n$. Any alignment of A and A' must describe l insertions and $l + k$ deletions for some $0 \leq l \leq n - k$, so the cost is at least $g_p(l) + g_p(l + k)$. The alignment in Figure 5.3(b) describes k deletions and has cost $g_p(k)$. The assumption in the lemma and the sub-additivity of g_p implies that $g_p(k) \leq g_p(j) + g_p(k - j) \leq g_p(l) + g_p(l + k)$ for all $l > 0$. The alignment in Figure 5.3(b) is thus optimal and the protein level cost of the deletion is $g_p(k)$.

A deletion of $3k$ nucleotides affecting $k + 1$ codons, say a deletion of the $3k$ nucleotides $a_3^i a_1^{i+1} a_2^{i+1} a_3^{i+1} \cdots a_1^{i+k} a_2^{i+k}$, transforms $A = A_1 A_2 \cdots A_n$ to $A' = A_1 A_2 \cdots A_{i-1} \upsilon A_{i+k+1} \cdots A_n$, where υ is the amino acid encoded by $a_1^i a_2^i a_3^{i+k}$. We say that υ is the remaining amino acid, and that $a_1^i a_2^i a_3^{i+k}$ is the remaining codon. Any alignment of A and A' describing exactly k deletions must align υ with A_{i+j} for some $0 \leq j \leq k$, so by sub-additivity of g_p the cost is at least $g_p(j) + c_p(A_{i+j}, \upsilon) + g_p(k - j)$. Figure 5.3(c) illustrates one of the $k + 1$ possible alignments of A and A' where υ is aligned with an affected amino acid and all non-affected amino acids are aligned. Such an alignment describes exactly k deletions, and the cost of an optimal alignment among them has cost

$$\min_{j=0,1,\ldots,k} \{g_p(j) + c_p(A_{i+j}, \upsilon) + g_p(k - j)\}, \tag{5.4}$$

$$\begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{E} & \mathrm{F} & \mathrm{C} & \mathrm{D} \\ \mathrm{A} & \mathrm{B} & \mathrm{G} & - & \mathrm{C} & \mathrm{D} \end{bmatrix} \qquad \begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{E} & \mathrm{F} & - & \mathrm{C} & \mathrm{D} \\ \mathrm{A} & \mathrm{B} & - & - & \mathrm{G} & \mathrm{C} & \mathrm{D} \end{bmatrix}$$

Figure 5.4: Two alignments of the amino acids ABEFCD and ABGCD.

and is thus optimal for any alignment describing exactly k deletions. Any other alignment of A and A' must describe l insertions and $l + k$ deletions for some $0 < l \leq n - k$, so the cost is at least $g_p(l) + g_p(l + k)$. The assumption in the lemma implies that the cost given by Equation 5.4 is less than or equal to $g_p(l) + g_p(l + k)$. The protein level cost of the deletion is thus given by Equation 5.4. □

The assumption in Lemma 5.1 is sufficient to ensure that we can compute the protein level cost of a nucleotide event efficiently, but the formulation of the lemma is to general to make the assumption necessary. The following example however suggests when the assumption is necessary. Consider a deletion of three nucleotides that transforms the six amino acids ABEFCD to ABGCD, i.e. $X =$ EF and $X' =$ G. Consider the two alignments shown in Figure 5.4. If we assume that $c_p(\mathrm{E}, \mathrm{G}) \leq c_p(\mathrm{F}, \mathrm{G})$ then the cost of the alignment in Figure 5.4 (left) is $c_p(\mathrm{E}, \mathrm{G}) + g_p(1)$ while the cost of the alignment in Figure 5.4 (right) is $g_p(2) + g_p(1)$. If the assumption in Lemma 5.1 does not hold then $g_p(2) + g_p(1)$ might be less than $c_p(\mathrm{E}, \mathrm{G}) + g_p(1)$ because $c_p(\mathrm{E}, \mathrm{G})$ can be arbitrary large. Hence, the protein level cost of the deletion would not be the cost of an alignment describing the minimum number of insertion-deletions.

5.3 The Cost of an Alignment

Before describing how to compute the cost of an optimal alignment of two sequence in the DNA/Protein model, we need to know how to compute the cost of a given alignment in the DNA/Protein model.

An alignment of two sequences describes a set of events necessary to transform one of the sequences into the other sequence, but it does not describe the order of the events. As observed in Section 5.2.2 the DNA level cost of an alignment is independent of the order of the events whereas the protein level cost depends on the order of the events. This implies that the DNA level cost of an alignment is just the sum of the DNA level costs of the events described by the alignment whereas the protein level cost of the same alignment is somewhat harder to determine. An obvious way to determine the protein level cost of an alignment is to minimize over all possible sequences of the events described by the alignment. This method is however not feasible in practice due to the factorial number of possible sequences one has to consider.

If Lemma 5.1 is fulfilled, the protein level cost of a nucleotide event only depends on the affected codons. This allows us to decompose the computation of the protein level cost of an alignment into smaller subproblems. The idea is to decompose the alignment into *codon alignments*. A codon alignment is a minimal part of the alignment that corresponds to a path connecting two nodes

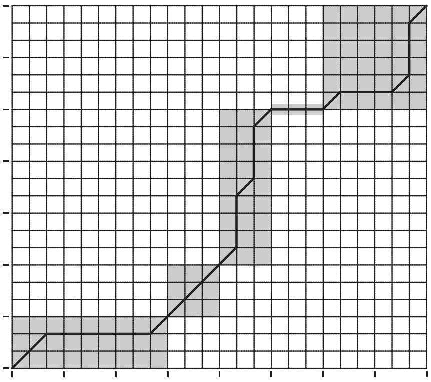


Figure 5.5: An alignment of two sequences decomposed into codon alignments.

$(3i', 3j')$ and $(3i, 3j)$ in the alignment graph. As illustrated in Figure 5.5, we can decompose an alignment uniquely into codon alignments.

The assumption that the length of an insertion or deletion is a multiple of three implies that a codon alignment either describes no substitutions (see Type 2 and 3 in Figure 5.7) or exactly three substitutions. If a codon alignment describes exactly three substitutions then it can also describe one or more insertion-deletions. More precisely, between any two consecutive substitutions in the codon alignment there can be an alternating sequence of insertions and deletions each of length a multiple of three. Such a sequence of insertion-deletions corresponds to a "staircase" in the alignment graph. Figure 5.6 illustrates the set of codon alignments that describe three substitutions and at most one insertion and deletion between two consecutive substitutions, i.e. the set of codon alignments where the "staircase" is limited to at most one "step". A particular codon alignment in this set corresponds to choosing which sides of the two dotted rectangles to traverse.

We observe that nucleotide events described by two different codon alignments in the decomposition do not affect the same codons. Hence, the protein level cost of a codon alignment can be computed independently of the other codon alignments in the decomposition. We can thus compute the protein level cost of an alignment as the sum of the protein level cost of each of the codon alignments in the decomposition. Since a codon alignment can describe an alternating sequence of insertion-deletions between two consecutive substitutions, it is possible that a decomposition of an alignment of two sequences of length n and m contains codon alignments that describe $\Theta(n + m)$ events. This implies that the problem of computing the cost of the codon alignments in a decomposition of an alignment is not any easier in the worst case than computing the cost of the alignment itself.

One way to circumvent this problem is to consider only alignments that can be decomposed into (or built of) codon alignments with at most some maximum number of insertion-deletions between any two consecutive substitutions. This upper bounds the number of events described by any codon alignment by some constant, which implies that we can determine the cost of a codon alignment

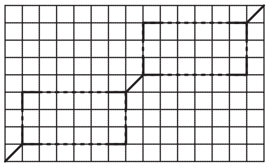


Figure 5.6: A summary of all possible codon alignments with at most one insertion and one deletion between two consecutive substitutions.

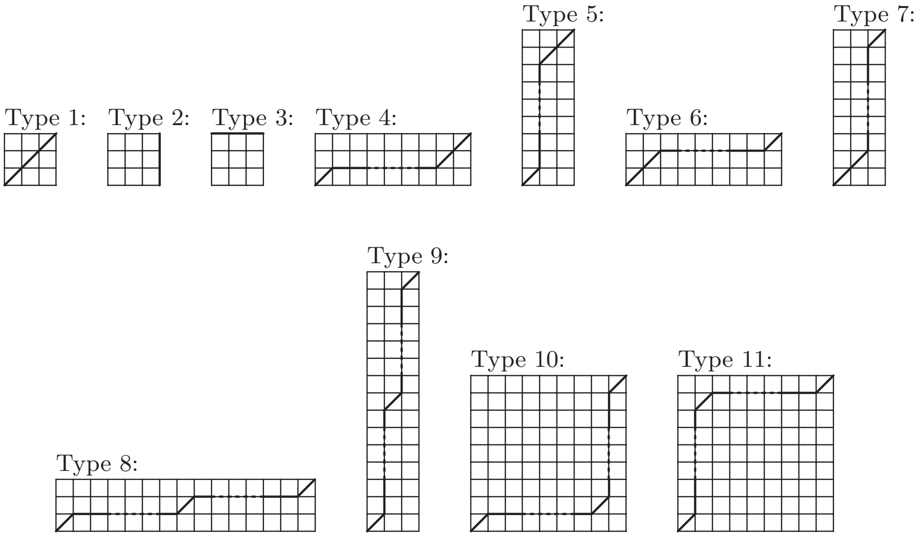


Figure 5.7: The eleven types of codon alignments with at most one insertion or one deletion between two consecutive substitutions.

in constant time simply by minimizing over all possible sequences of the events described by the codon alignment. The protein level cost of an alignment can thus be determined in time proportional to the number of codon alignments in the decomposition of the alignment.

Hein in [83] allows at most one insertion *or* one deletion between any two consecutive substitutions in a codon alignment. Besides the two codon alignments that describe no substitutions, this corresponds to the set of codon alignments obtainable from Figure 5.6 when either the width or the height of each of the two dotted rectangles must be zero. As illustrated in Figure 5.7 this implies that there are eleven different types of codon alignments. In the appendix after Section 5.7 we consider the problem of restricting the set of possible codon alignments in further details.

5.4 A Simple Alignment Algorithm

Let $a_1 a_2 \cdots a_{3n}$ and $b_1 b_2 \cdots b_{3m}$ be two coding sequences of DNA. Hein in [83] describes how the decomposition into codon alignments makes it possible to compute the cost of an optimal alignment of a and b in the DNA/Protein model in time $\mathrm{O}(n^2m^2)$. The algorithm assumes that Lemma 5.1 is fulfilled, and that we only allow codon alignments with some maximum number of insertion-deletions between two consecutive substitutions, e.g. the eleven types of codon alignments in Figure 5.7. The algorithm can be summarized as follows.

Let $D(i,j)$ denote the cost of an optimal alignment of $a_1a_2\cdots a_{3i}$ and $b_1b_2\cdots b_{3j}$. We define $D(0,0)$ to be zero and $D(i,j)$ to be infinity for $i<0$ or $j<0$. An optimal alignment of $a_1a_2\cdots a_{3i}$ and $b_1b_2\cdots b_{3j}$ can be decomposed into codon alignments $ca_1, ca_2, \ldots, ca_k$. We say that ca_k is the last codon alignment and that $ca_1, ca_2, \ldots, ca_{k-1}$ is the remaining alignment.

If the last codon alignment ca_k is an alignment of $a_{3i'+1}a_{3i'+2}\cdots a_{3i}$ and $b_{3j'+1}b_{3j'+2}\cdots b_{3j}$ for some $(i',j')<(i,j)$[2], then $D(i,j)$ is equal to $D(i',j') + \mathit{cost}(ca_k)$. This is the cost of the last codon alignment plus the cost of the remaining alignment. We can compute $D(i,j)$ by minimizing the expression $D(i',j') + \mathit{cost}(ca)$ over all $(i',j')<(i,j)$ and all possible codon alignments ca of $a_{3i'+1}a_{3i'+2}\cdots a_{3i}$ and $b_{3j'+1}b_{3j'+2}\cdots b_{3j}$.

The upper bound on the number of insertion-deletions in a codon alignment implies that the number of possible codon alignments of $a_{3i'+1}a_{3i'+2}\cdots a_{3i}$ and $b_{3j'+1}b_{3j'+2}\cdots b_{3j}$, for all $(i',j')<(i,j)$, is upper bounded by some constant. For example, if we only consider the eleven types of codon alignments in Figure 5.7 there are at most three possible codon alignments of $a_{3i'+1}a_{3i'+2}\cdots a_{3i}$ and $b_{3j'+1}b_{3j'+2}\cdots b_{3j}$. Hence, if we assume that $D(i',j')$ is known for all $(i',j')<(i,j)$ then we can compute $D(i,j)$ in time $\mathrm{O}(ij)$. By dynamic programming this implies that we can compute $D(n,m)$ in time $\mathrm{O}(n^2m^2)$ and space $\mathrm{O}(nm)$. By back-tracking we can also get an optimal alignment (and not only the cost) within the same time and space bound.

5.5 An Improved Alignment Algorithm

Let a and b be coding sequences of DNA as introduced above. We will describe how to compute the cost of an optimal alignment of a and b in the DNA/Protein model in time $\mathrm{O}(nm)$ and space $\mathrm{O}(n)$. Besides the assumptions of the simple algorithm described in the previous section, we also assume that the function $g(k) = g_d(3k) + g_p(k)$ is affine, i.e. $g(k) = \alpha + \beta k$ for some $\alpha, \beta \geq 0$. We say that g is the *combined gap cost function*.

The idea behind the improved algorithm is similar to the idea behind the simple algorithm in the sense that we compute the cost of an optimal alignment by minimizing the cost over all possible last codon alignments. We define $D^t(i,j)$ to be the cost of an optimal alignment of $a_1a_2\cdots a_{3i}$ and $b_1b_2\cdots b_{3j}$ under the assumption that the last codon alignment is of type t. Remember that we only allow codon alignments with some maximum number of insertion-deletions between two consecutive substitutions. This implies that the number of possible codon alignment, i.e. types of codon alignments, is upper bounded by some constant. We define $D^t(0,0)$ to be zero and $D^t(i,j)$ to be infinity for

[2] We say that $(i',j')<(i,j)$ iff $i' \leq i \wedge j' \leq j \wedge (i' \neq i \vee j' \neq j)$.

$i < 0$ or $j < 0$. We can compute $D(i, j)$ as

$$D(i, j) = \min_t D^t(i, j)\,. \tag{5.5}$$

Lemma 5.1 ensures that $D^t(i, j)$ is the cost of some last codon alignment (of type t) plus the cost of the corresponding remaining alignment. This allows us to compute $D^t(i, j)$ by minimizing the cost over all possible last codon alignments of type t. The assumption that g is affine makes it possible to compute $D^t(i, j)$ in constant time if $D^{t'}(k, l)$ is known for some $(k, l) < (i, j)$ and some t'. Since we only consider a constant number of possible codon alignments (e.g. the types in Figure 5.7) this implies that we can compute $D(n, m)$ in time $\mathrm{O}(nm)$ and space $\mathrm{O}(n)$. The technique described Hirschberg in [86], or the variant described in Durbin *et al.* in [46, page 35–36], allows us to get an optimal alignment (and not only the cost) within the same time and space bound.

Our method for computing $D^t(i, j)$ is applicable to any type of last codon alignment, but the bookkeeping, and thereby the constant overhead of the computation increases with the number of gaps described by the last codon alignment. By restricting ourselves to the eleven types of codon alignments in Figure 5.7, the computation of an optimal alignment has a reasonable constant overhead. We therefore focus on these eleven types of codon alignments. The detailed description of how to compute $D^t(i, j)$ for $t = 1, 2, \ldots, 11$ is divided into three cases depending on the number of gaps within a codon – *internal gaps* – described by a codon alignment of type t. Codon alignments of type 1–3 describe no internal gaps, codon alignments of type 4–7 describe one internal gap, and codon alignments of type 8–11 describe two internal gaps.

Case I – No Internal Gaps

We introduce the function $c_p^* : \{\mathrm{A, C, G, T}\}^3 \times \{\mathrm{A, C, G, T}\}^3 \rightarrow \mathbb{R}$, where $c_p^*(\sigma_1\sigma_2\sigma_3, \tau_1\tau_2\tau_3)$ is the distance between the codons $\sigma_1\sigma_2\sigma_3$ and $\tau_1\tau_2\tau_3$ in the DNA/Protein model. This distance is computable in constant time as the minimum over the costs of the six possible sequences of the three substitutions $\sigma_1 \rightarrow \tau_1$, $\sigma_2 \rightarrow \tau_2$ and $\sigma_3 \rightarrow \tau_3$, where we use the term *cost* to denote the DNA level cost plus the protein level cost.

The cost $D^1(i, j)$ is the cost of the last codon alignment of type 1 plus the cost of the remaining alignment. The last codon alignment is an alignment of $a_1^i a_2^i a_3^i$ and $b_1^j b_2^j b_3^j$, which by definition of c_p^* has cost $c_p^*(a_1^i a_2^i a_3^i, b_1^j b_2^j b_3^j)$. The cost of the remaining alignment is $D(i - 1, j - 1)$, that is

$$D^1(i, j) = D(i - 1, j - 1) + c_p^*(a_1^i a_2^i a_3^i, b_1^j b_2^j b_3^j)\,. \tag{5.6}$$

A codon alignment of type 2 or 3 describes a gap between two codons. Since the combined gap cost function is affine, we can apply the technique by Gotoh in [67] saying that a gap ending in (i, j) is either a continuation of an existing gap ending in $(i - 1, j)$ or $(i, j - 1)$, or the start of a new gap, that is

$$D^2(i, j) = \min\{D(i, j - 1) + \alpha + \beta, D^2(i, j - 1) + \beta\}\,, \tag{5.7}$$

$$D^3(i, j) = \min\{D(i - 1, j) + \alpha + \beta, D^3(i - 1, j) + \beta\}\,. \tag{5.8}$$

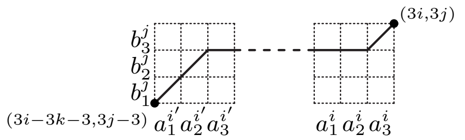


Figure 5.8: The last codon alignment of type 6.

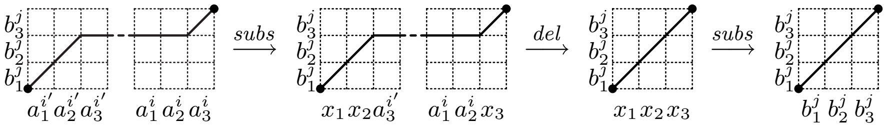


Figure 5.9: The evolution of $a_1^{i'} a_2^{i'} a_3^{i'} \cdots a_1^i a_2^i a_3^i$ to $b_1^j b_2^j b_3^j$ described by the last codon alignment.

Case II – One Internal Gap

We will only describe how to compute $D^6(i,j)$. The other cases where the last codon alignment describes a single internal gap can be handled similarly. The cost $D^6(i,j)$ is the cost of the last codon alignment of type 6 plus the cost of the remaining alignment. The last codon alignment of type 6 describes three substitutions and one deletion. If the deletion has length k (a deletion of $3k$ nucleotides) then the cost of the remaining alignment is $D(i-k-1, j-1)$ and the last codon alignment is an alignment of $a_1^{i'} a_2^{i'} a_3^{i'} \cdots a_1^i a_2^i a_3^i$ and $b_1^j b_2^j b_3^j$, where $i' = i - k$. This situation is illustrated in Figure 5.8.

The cost of the last codon alignment is the minimum cost of a sequence of the three substitutions and the deletion. Any sequence of these four events can be divided into three steps: The substitutions occurring before the deletion, the deletion, and the substitutions occurring after the deletion. Figure 5.9 illustrates the three steps of the evolution of $a_1^{i'} a_2^{i'} a_3^{i'} \cdots a_1^i a_2^i a_3^i$ to $b_1^j b_2^j b_3^j$, where x_1, x_2 and x_3 are the result of the up to three substitutions before the deletion. For example, if the substitution $a_1^{i'} \to b_1^j$ occurs before the deletion, then x_1 is b_1^j, otherwise it is $a_1^{i'}$. We say that $x_1 \in \{a_1^{i'}, b_1^j\}$, $x_2 \in \{a_2^{i'}, b_2^j\}$ and $x_3 \in \{a_3^i, b_3^j\}$ are the status of the three substitutions before the deletion, and observe that the codon $x_1 x_2 x_3$ is the remaining codon of the deletion.

The substitutions occurring before the deletion change codon $a_1^{i'} a_2^{i'} a_3^{i'}$ to $x_1 x_2 a_3^{i'}$, and codon $a_1^i a_2^i a_3^i$ to $a_1^i a_2^i x_3$. The substitutions occurring after the deletion change codon $x_1 x_2 x_3$ to $b_1^j b_2^j b_3^j$. Recall that the cost of changing codon $\sigma_1 \sigma_2 \sigma_3$ to codon $\tau_1 \tau_2 \tau_3$ by a sequence of the substitutions $\sigma_1 \to \tau_1$, $\sigma_2 \to \tau_2$ and $\sigma_3 \to \tau_3$ is given by $c_p^*(\sigma_1 \sigma_2 \sigma_3, \tau_1 \tau_2 \tau_3)$. Since an identical substitution has cost zero, the cost of the three substitutions in the last codon alignment is equal to the cost of the induced codon changes, that is

$$\begin{multline} cost(subs) = c_p^*(a_1^{i'} a_2^{i'} a_3^{i'}, x_1 x_2 a_3^{i'}) + \\ c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i x_3) + c_p^*(x_1 x_2 x_3, b_1^j b_2^j b_3^j) \,. \quad (5.9) \end{multline}$$

The cost of deleting $3k$ nucleotides in the last codon alignment is the sum of the DNA level cost $g_d(3k)$ and the protein level cost cf. Equation 5.4. Since the combined gap cost function $g(k) = g_d(3k) + g_p(k) = \alpha + \beta k$ is affine, and the remaining codon of the deletion is $x_1x_2x_3$, we can write this sum as

$$cost(del) = \min \begin{cases} \alpha + \beta k + c_p(a_1^i a_2^i x_3, x_1x_2x_3) \\ \alpha_p + \alpha + \beta k + \min_{0<l<k} c_p(a_1^{i-l} a_2^{i-l} a_3^{i-l}, x_1x_2x_3) \\ \alpha + \beta k + c_p(x_1x_2a_3^{i'}, x_1x_2x_3) \end{cases} \tag{5.10}$$

where $c_p(\sigma_1\sigma_2\sigma_3, \tau_1\tau_2\tau_3)$ is used as convenient notation for $c_p(\sigma, \tau)$, where σ and τ are the amino acids encoded by the codons $\sigma_1\sigma_2\sigma_3$ and $\tau_1\tau_2\tau_3$. The cost of the deletion depends on the deletion length, the remaining codon $x_1x_2x_3$ and a *witness*. The witness can be the start-codon $x_1x_2a_3^{i'}$, the end-codon $a_1^i a_2^i x_3$, or one of the internal codons $a_1^{i-l}a_2^{i-l}a_3^{i-l}$ for some $0 < l < k$. The witness encodes the amino acid aligned with the remaining amino acid.

Assuming a deletion length k and remaining codon $x_1x_2x_3$ of the deletion in the last codon alignment, we can compute $D^6(i,j)$ as the sum $cost(subs) + cost(del) + D(i-k-1, j-1)$. We can thus compute $D^6(i,j)$ by minimizing this sum over all possible combinations of deletion length k and remaining codon $x_1x_2x_3$. A combination of deletion length k and remaining codon $x_1x_2x_3$ is possible if $x_1 \in \{a_1^{i'}, b_1^j\}$, $x_2 \in \{a_2^{i'}, b_2^j\}$, and $x_3 \in \{a_3^i, b_3^j\}$ where $i' = i - k$. Since the terms $c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i x_3)$ and $c_p^*(x_1x_2x_3, b_1^j b_2^j b_3^j)$ of $cost(subs)$ do not depend on the deletion length, we can split the minimization as

$$\begin{aligned} D^6(i,j) = \min_{x_1x_2x_3} \{ & c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i x_3) + \\ & c_p^*(x_1x_2x_3, b_1^j b_2^j b_3^j) + D^6_{x_1x_2x_3}(i,j) \} \end{aligned} \tag{5.11}$$

where

$$\begin{aligned} D^6_{x_1x_2x_3}(i,j) = \min_{0<k<i} \{ & D(i-k-1, j-1) + \\ & c_p^*(a_1^{i-k} a_2^{i-k} a_3^{i-k}, x_1x_2a_3^{i-k}) + cost(del) \} \end{aligned} \tag{5.12}$$

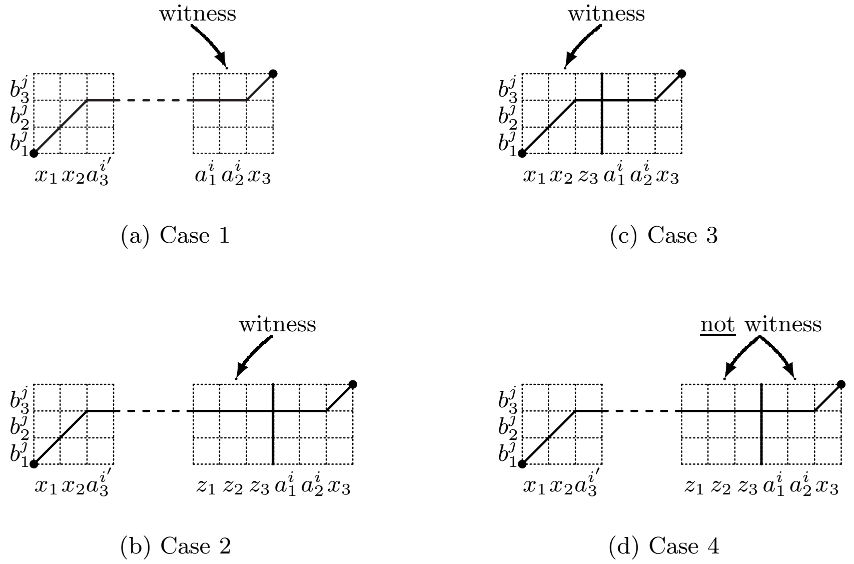
is the minimum cost of the terms that depend on both the deletion length and the remaining codon under the assumption that the remaining codon is $x_1x_2x_3$. If we expand the term $cost(del)$ we get

$$\begin{aligned} D^6_{x_1x_2x_3}(i,j) = \min_{0<k<i} \{ & \mathrm{len}^6_{x_1x_2}(i,j,k) + \\ & \min \begin{cases} c_p(a_1^i a_2^i x_3, x_1x_2x_3) \\ \alpha_p + \min_{0<l<k} c_p(a_1^{i-l} a_2^{i-l} a_3^{i-l}, x_1x_2x_3) \\ c_p(x_1x_2a_3^{i-k}, x_1x_2x_3) \end{cases} \} \end{aligned} \tag{5.13}$$

where

$$\begin{aligned} \mathrm{len}^6_{x_1x_2}(i,j,k) = D(i-k-1, j-1) + & \\ c_p^*(a_1^{i-k} a_2^{i-k} a_3^{i-k}, x_1x_2a_3^{i-k}) + \alpha + \beta k & \end{aligned} \tag{5.14}$$

is the cost of the remaining alignment plus the part of the cost of the last codon alignment that does not depend on the codon $a_1^i a_2^i a_3^i$ and the witness.



(a) Case 1

(c) Case 3

(b) Case 2

(d) Case 4

Figure 5.10: The four cases in the computation of $D^6_{x_1x_2x_3}(i,j)$. We use $z_1z_2z_3$ as short notation for $a_1^{i-1}a_2^{i-1}a_3^{i-1}$.

The cost $\text{len}^6_{x_1x_2}(i,j,k)$ is defined if $x_1 \in \{a_1^{i-k}, b_1^j\}$ and $x_2 \in \{a_2^{i-k}, b_2^j\}$. The cost $D^6_{x_1x_2x_3}(i,j)$ is defined if there exists a deletion length k such that k and $x_1x_2x_3$ is a possible combination of deletion length and remaining codon. We observe that there are at most 32 possible remaining codons $x_1x_2x_3$. This observation follows because we known that x_3 must be one of the two known nucleotides a_3^i or b_3^j. If we can compute $D^6_{x_1x_2x_3}(i,j)$ in constant time for each of the possible remaining codons then we can also compute $D^6(i,j)$ in constant time. To compute $D^6_{x_1x_2x_3}(i,j)$ we must determine a combination of witness and deletion length that minimizes the cost. We say that we must determine the witness and deletion length of $D^6_{x_1x_2x_3}(i,j)$.

The combination of witness and deletion length that minimizes $D^6_{x_1x_2x_3}(i,j)$ must be one of the four cases illustrated in Figure 5.10. We can thus compute $D^6_{x_1x_2x_3}(i,j)$ by minimizing the cost over these four cases. The cost of Cases 1–3 can be computed by simplifying Equation 5.13 for the particular combination of witness and deletion length, whereas the cost of Case 4 is more difficult to compute because both the witness and the deletion length are unknown.

Case 1. The end-codon is the witness and the deletion length is at least one. The cost is $\min\limits_{0<k<i} \text{len}^6_{x_1x_2}(i,j,k) + c_p(a_1^ia_2^ix_3, x_1x_2x_3)$.

Case 2. The last internal codon is the witness and the deletion length is at least two. The cost is $\min\limits_{1<k<i} \text{len}^6_{x_1x_2}(i,j,k) + \alpha_p + c_p(a_1^{i-1}a_2^{i-1}a_3^{i-1}, x_1x_2x_3)$.

Case 3. The start-codon is the witness and the deletion length is one. The cost is $\text{len}^6_{x_1x_2}(i,j,1) + c_p(x_1x_2a_3^{i-1}, x_1x_2x_3)$.

Case 4. The witness is neither the end-codon nor the last internal codon and the deletion length is at least two. If the witness of $D^6_{x_1x_2x_3}(i-1,j)$ is not the end-codon $a_1^{i-1}a_2^{i-1}x_3$, we observe from optimality of $D^6_{x_1x_2x_3}(i-1,j)$ that the cost of Case 4 is $D^6_{x_1x_2x_3}(i-1,j)+\beta$.

The observation in Case 4 implies that we can use dynamic programming to keep track of $D^6_{x_1x_2x_3}(i,j)$ under the assumption that the end-codon is not the witness, i.e. use dynamic programming to keep track of the minimum cost of Cases 2–4. For this purpose, we introduce tables $F^6_{x_1x_2x_3}$ corresponding to the 64 combinations of $x_1x_2x_3$, such that if $x_1x_2x_3$ is a possible remaining codon, and the end-codon $a_1^ia_2^ix_3$ is not the witness of $D^6_{x_1x_2x_3}(i,j)$, then entry $F^6_{x_1x_2x_3}(i,j)$ is equal to $D^6_{x_1x_2x_3}(i,j)$. If we define $F^6_{x_1x_2x_3}(0,j)$ to infinity, then

$$F^6_{x_1x_2x_3}(i,j)=\min\begin{cases}\text{cost of } \textit{Case 2}\\ \text{cost of } \textit{Case 3}\\ F^6_{x_1x_2x_3}(i-1,j)+\beta\end{cases} \tag{5.15}$$

In order to compute the cost of case 1 and 2 in constant time we maintain the minimum of $\text{len}^6_{x_1x_2}(i,j,k)$ over k by dynamic programming. We introduce tables $L^6_{x_1x_2}$ corresponding to the 16 combinations of x_1x_2 such that $L^6_{x_1x_2}(i,j)$ is equal to $\min_{0<k<i}\text{len}^6_{x_1x_2}(i,j,k)$. If we define $L^6_{x_1x_2}(0,j)$ to infinity, then

$$L^6_{x_1x_2}(i,j)=\min\begin{cases}\text{len}^6_{x_1x_2}(i,j,1)\\ L^6_{x_1x_2}(i-1,j)+\beta\end{cases} \tag{5.16}$$

We can now compute $D^6_{x_1x_2x_3}(i,j)$ in constant time as the minimum cost of Cases 1–4. The cost of Case 1 is $L^6_{x_1x_2}(i,j)+c_p(a_1^ia_2^ix_3,x_1x_2x_3)$, and the minimum cost of case 2–4 is $F^6_{x_1x_2x_3}(i,j)$, that is

$$D^6_{x_1x_2x_3}(i,j)=\min\begin{cases}L^6_{x_1x_2}(i,j)+c_p(a_1^ia_2^ix_3,x_1x_2x_3)\\ F^6_{x_1x_2x_3}(i,j)\end{cases} \tag{5.17}$$

To compute $D^6(i,j)$ using Equation 5.11, we must compute $D^6_{x_1x_2x_3}(i,j)$ using Equation 5.17 for all 32 possible remaining codons, which involves computing entry (i,j) in the 16 $L^6_{x_1x_2}$ tables and the 64 $F^6_{x_1x_2x_3}$ tables. Since the computation of each table entry takes constant time, the total computation of $D^6(i,j)$ takes constant time. The other three cases where the last codon alignment describes one internal gap (codon alignments of type 4, 5 and 7) can be handled similarly. However, if the last codon alignment is of type 4 or 5, it follows that only the first nucleotide x_1 in the remaining codon depends on the deletion (or insertion) length. This limits the number of possible remaining codons to 16, and implies that only four tables are needed in order to keep track of values $\min_{0<k<i}\text{len}^t_{x_1}(i,j,k)$. Hence, to compute $D^t(i,j)$, for $t=4,5,6,7$, we compute $2\cdot 4+2\cdot 16+4\cdot 64=296$ table entries in total.

Case III – Two Internal Gaps

We will only describe how to compute $D^8(i,j)$. The other cases where the last codon alignment describes two internal gaps can be handled similarly. The

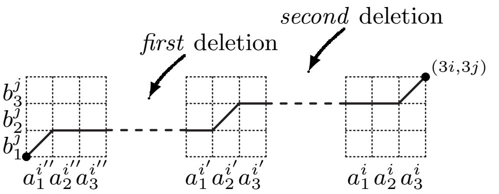


Figure 5.11: The last codon alignment of type 8

cost $D^8(i,j)$ is the cost of the last codon alignment of type 8 plus the cost of the remaining alignment. The last codon alignment of type 8 describes three substitutions and two deletions. If the first deletion has length k' and the second deletion has length k, then the cost of the remaining alignment is $D(i-k-k'-1, j-1)$, and the last codon alignment is an alignment of $a_1^{i''}a_2^{i''}a_3^{i''}\cdots a_1^{i'}a_2^{i'}a_3^{i'}\cdots a_1^{i}a_2^{i}a_3^{i}$ and $b_1^jb_2^jb_3^j$, where $i' = i-k$ and $i'' = i'-k'$. This situation is illustrated in Figure 5.11.

Similar to computing $D^6(i,j)$, we will compute $D^8(i,j)$ by minimizing the cost over all possible combinations of deletion length, k, and remaining codon, $x_1x_2x_3$, of the second deletion. Computing $D^8(i,j)$ is then reduced to computing $D^8_{x_1x_2x_3}(i,j)$, which is the cost under the assumption that the remaining codon of the second deletion is $x_1x_2x_3$. There are 32 possible remaining codons of the second deletion. The method used to compute $D^6_{x_1x_2x_3}(i,j)$ can be adapted to compute $D^8_{x_1x_2x_3}(i,j)$. An inspection of Equations 5.15, 5.16 and 5.17 reveals that all we have to do is to replace the term $\text{len}^6_{x_1x_2}(i,j,1)$ with the corresponding part of the cost $D^8(i,j)$, which we denote $\text{len}^8_{x_1x_2x_3}(i,j,1)$.

More precisely, if we assume that the second deletion has length k and remaining codon $x_1x_2x_3$, then $\text{len}^8_{x_1x_2x_3}(i,j,k)$ is the part of $D^8(i,j)$ that does not depend on the codon $a_1^ia_2^ia_3^i$ and the witness of the second deletion. We observe that this cost depends on the order of the two deletions in the last codon alignment. Hence, we introduce $\text{len}^{8'}_{x_1x_2x_3}(i,j,k)$ and $\text{len}^{8''}_{x_1x_2x_3}(i,j,k)$ to denote the cost when the first deletion occurs before the second deletion and vice versa, and define $\text{len}^8_{x_1x_2x_3}(i,j,k)$ as the minimum $\min\{\text{len}^{8'}_{x_1x_2x_3}(i,j,k), \text{len}^{8''}_{x_1x_2x_3}(i,j,k)\}$. Because we only have to compute $\text{len}^8_{x_1x_2x_3}(i,j,1)$, we can restrict ourselves to the situation where the second deletion has length one and the first deletion has length k'. We split the explanation of how to compute $\text{len}^8_{x_1x_2x_3}(i,j,1)$ into two cases depending on the order of the deletions. For notational convenience, we will use i' and i'' to denote $i-1$ and $i'-k'$ respectively.

Figure 5.12 illustrates the evolution of the last codon alignment (of type 8) when the second deletion has length one and occurs after the first deletion. The nucleotides y_1, y_2 and y_3 are the status of the substitutions before the first deletion, and the nucleotides x_1, x_2 and x_3 are the status of the substitutions before the second deletion. Similar to Equation 5.9, we can compute the cost of the three substitutions as the cost of the induced codon changes. It thus follows from Figure 5.12 that the cost of the three substitutions is

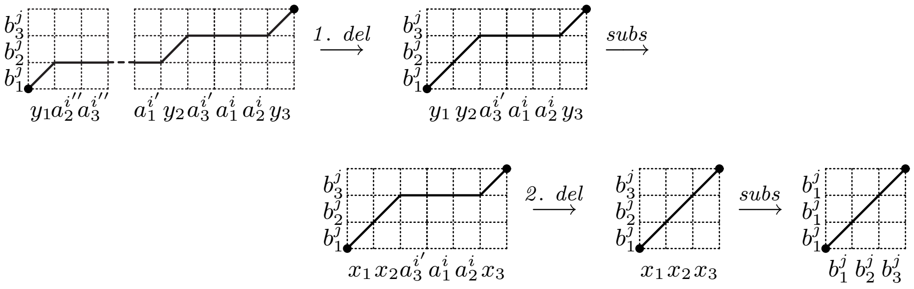

b_3^j
b_2^j
b_1^j
$y_1 a_2^{i''} a_3^{i''}$
$a_1^{i'}\, y_2 a_3^{i'}\, a_1^i\, a_2^i\, y_3$
1. del
$y_1\, y_2 a_3^{i'}\, a_1^i\, a_2^i\, y_3$
subs
$x_1 x_2 a_3^{i'}\, a_1^i\, a_2^i\, x_3$
2. del
$x_1 x_2 x_3$
subs
b_1^j
$b_1^j\, b_2^j\, b_3^j$

Figure 5.12: The first deletion occurs before the second deletion and the second deletion has length one.

$$\begin{aligned}
cost(subs) = {} & c_p^*(a_1^{i''} a_2^{i''} a_3^{i''}, y_1 a_2^{i''} a_3^{i''}) + c_p^*(a_1^{i'} a_2^{i'} a_3^{i'}, a_1^{i'} y_2 a_3^{i'}) + \\
& c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i y_3) + c_p^*(y_1 y_2 a_3^{i'}, x_1 x_2 a_3^{i'}) + \\
& c_p^*(a_1^i a_2^i y_3, a_1^i a_2^i x_3) + c_p^*(x_1 x_2 x_3, b_1^j b_2^j b_3^j)\,. \qquad (5.18)
\end{aligned}$$

We can compute the cost of the two deletions similar to Equation 5.10. If we recall that the first deletion has length k', the second deletion has length one, and the notation $i' = i - 1$ and $i'' = i' - k'$, we can write the costs as

$$cost(del_1) = \min \begin{cases} \alpha + \beta k' + c_p(a_1^{i'} y_2 a_3^{i'}, y_1 y_2 a_3^{i'}) \\ \alpha_p + \alpha + \beta k' + \min_{0<l<k'} c_p(a_1^{i'-l} a_2^{i'-l} a_3^{i'-l}, y_1 y_2 a_3^{i'}) \\ \alpha + \beta k' + c_p(y_1 a_2^{i''} a_3^{i''}, y_1 y_2 a_3^{i'}) \end{cases} \qquad (5.19)$$

$$cost(del_2) = \alpha + \beta + \min \begin{cases} c_p(x_1 x_2 a_3^{i'}, x_1 x_2 x_3) \\ c_p(a_1^i a_2^i x_3, x_1 x_2 x_3) \end{cases} \qquad (5.20)$$

If we assume that the first deletion occurs before the second deletion, and that the second deletion has length one and remaining codon $x_1x_2x_3$, then $D^8(i,j)$ is $cost(subs) + cost(del_1) + cost(del_2) + D(i'-k'-1, j-1)$ minimized over all possible combinations of $y_1y_2y_3$ and k'. Recall that the cost $\text{len}^{8'}_{x_1x_2x_3}(i,j,1)$ is the part of $D^8(i,j)$ that neither depends on $a_1^i a_2^i a_3^i$ nor the witness of the second deletion. Inspecting the above expressions shows that $\text{len}^{8'}_{x_1x_2x_3}(i,j,1)$ must include everything but $c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i y_3)$ and $c_p^*(a_1^i a_2^i y_3, a_1^i a_2^i x_3)$ of $cost(subs)$, and $\min\{c_p(x_1 x_2 a_3^{i'}, x_1x_2x_3), c_p(a_1^i a_2^i x_3, x_1x_2x_3)\}$ of $cost(del_2)$. It can be verified that the value $D^4_{y_1 y_2 a_3^{i'}}(i',j)$ is equal to the sum $D(i'-k'-1, j-1) + c_p^*(a_1^{i''} a_2^{i''} a_3^{i''}, y_1 a_2^{i''} a_3^{i''}) + cost(del_1)$ minimized over the deletion length k' of the first deletion, which makes it possible to compute $\text{len}^{8'}_{x_1x_2x_3}(i,j,1)$ as

$$\begin{aligned}
\text{len}^{8'}_{x_1x_2x_3}(i,j,1) = {} & \alpha + \beta + c_p^*(x_1x_2x_3, b_1^j b_2^j b_3^j) + \\
& \min_{y_1 y_2} \{ c_p^*(a_1^{i'} a_2^{i'} a_3^{i'}, a_1^{i'} y_2 a_3^{i'}) + D^4_{y_1 y_2 a_3^{i'}}(i',j) + c_p^*(y_1 y_2 a_3^{i'}, x_1 x_2 a_3^{i'}) \}\,, \qquad (5.21)
\end{aligned}$$

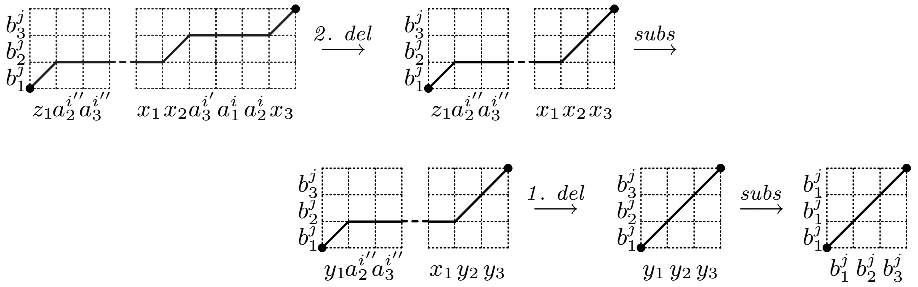


Figure 5.13: The second deletion occurs before the first deletion and the second deletion has length one.

where $y_1 \in \{a_1^{i''}, x_1\}$ and $y_2 \in \{a_2^{i'}, x_2\}$. The cost $\mathrm{len}^{8'}_{x_1x_2x_3}(i,j,1)$ is defined if the remaining codon $x_1x_2x_3$ allows the second deletion to have length one, i.e. if $x_1 \in \{a_1^{i''}, b_1^j\}$, $x_2 \in \{a_2^{i'}, b_2^j\}$, and $x_3 \in \{a_3^i, b_3^j\}$. The nucleotide $a_1^{i''}$ depends on the unknown deletion length of the first deletion, so we must assume that it can be any of the four nucleotides.

Figure 5.13 illustrates the evolution of the last codon alignment when the second deletion has length one and occurs before the first deletion. The nucleotides z_1, x_2 and x_3 are the status of the substitutions before the second deletion, and the nucleotides y_1, y_2 and y_3 are the status of the substitutions before the first deletion. We observe that x_1 is just $a_1^{i'}$.

The cost of this case can be described as above. This would reveal that $\mathrm{len}^{8''}_{x_1x_2x_3}(i,j,1)$ includes everything but $c_p^*(a_1^ia_2^ia_3^i, a_1^ia_2^ix_3)+c_p^*(a_1^ia_2^ix_3, a_1^ia_2^iy_3)+c_p(x_1x_2x_3, w_1w_2w_3)$, where $w_1w_2w_3$ is the witness of the second deletion. Furthermore, it would reveal that $D^4_{y_1y_2y_3}(i',j)$ is equal to the sum of the cost of the remaining alignment, the cost of the first deletion, and the cost of changing codon $a_1^{i''}a_2^{i''}a_3^{i''}$ to $z_1a_2^{i''}a_3^{i''}$ to $y_1a_2^{i''}a_3^{i''}$, minimized over the deletion length k' of the first deletion. This makes it possible to compute $\mathrm{len}^{8''}_{x_1x_2x_3}(i,j,1)$ as

$$\begin{aligned}\mathrm{len}^{8''}_{x_1x_2x_3}(i,j,1) &= \alpha+\beta+c_p^*(a_1^{i'}a_2^{i'}a_3^{i'}, x_1x_2a_3^{i'})+\\ &\min_{y_1y_2y_3}\left\{c_p^*(a_1^{i'}x_2x_3, a_1^{i'}y_2y_3)+D^4_{y_1y_2y_3}(i',j)+c_p^*(y_1y_2y_3, b_1^jb_2^jb_3^j)\right\}, \quad (5.22)\end{aligned}$$

where $y_1 \in \{z_1, b_1^j\}$, $y_2 \in \{x_2, b_2^j\}$ and $y_3 \in \{x_3, b_3^j\}$. The nucleotide z_1 depends on the unknown deletion length of the first deletion, so we must assume that z_1 can be any of the four nucleotides. The cost $\mathrm{len}^{8''}_{x_1x_2x_3}(i,j,1)$ is defined if the remaining codon $x_1x_2x_3$ allows the second deletion to have length one, i.e. if $x_1 = a_1^{i'}$, $x_2 \in \{a_2^{i'}, b_2^j\}$ and $x_3 \in \{a_3^i, b_3^j\}$.

We are finally in a position where we can describe how to use the method from the previous section to compute $D^8(i,j)$. The cost $\mathrm{len}^8_{x_1x_2x_3}(i,j,k)$ depends on x_1, x_2 and x_3, so instead of 16 tables we need 64 tables, $L^8_{x_1x_2x_3}$, to keep track of $\min_{0<k<i}\mathrm{len}^8_{x_1x_2x_3}(i,j,k)$. We still need 64 tables, $F^8_{x_1x_2x_3}$, to keep track of the cost under the assumption that the end-codon $a_1^ia_2^ix_3$ is not the witness (of the second deletion). We compute entry (i,j) in these tables as

$$L^8_{x_1x_2x_3}(i,j) = \min \left\{ \begin{array}{l} \mathrm{len}^8_{x_1x_2x_3}(i,j,1) \\ L^8_{x_1x_2x_3}(i-1,j) + \beta \end{array} \right. \tag{5.23}$$

$$F^8_{x_1x_2x_3}(i,j) = \min \left\{ \begin{array}{l} L^8_{x_1x_2x_3}(i-1,j) + \beta + \alpha_p + c_p(a_1^{i-1}a_2^{i-1}a_3^{i-1}, x_1x_2x_3) \\ \mathrm{len}^8_{x_1x_2x_3}(i,j,1) + c_p(x_1x_2a_3^{i-1}, x_1x_2x_3) \\ F^8_{x_1x_2x_3}(i-1,j) + \beta \end{array} \right. \tag{5.24}$$

We compute $D^8_{x_1x_2x_3}(i,j)$ using the above tables, and $D^8(i,j)$ by minimizing over the 32 possible remaining codons of the second deletion, that is

$$D^8_{x_1x_2x_3}(i,j) = \min \left\{ \begin{array}{l} L^8_{x_1x_2x_3}(i,j) + c_p(a_1^i a_2^i x_3, x_1x_2x_3) \\ F^8_{x_1x_2x_3}(i,j) \end{array} \right. \tag{5.25}$$

$$D^8(i,j) = \min_{x_1x_2x_3} \{ c_p^*(a_1^i a_2^i a_3^i, a_1^i a_2^i x_3) + D^8_{x_1x_2x_3}(i,j) \} \tag{5.26}$$

The compute $D^8(i,j)$, we compute entry (i,j) in 128 tables. Each computation takes constant time. The other three cases where the last codon alignment describes two internal gaps can be handled similarly. To compute $D^t(i,j)$, for $t = 8, 9, 10, 11$, we thus compute $4 \cdot 128 = 512$ table entries in total.

5.6 Reducing Space Consumption

The computations of $D^6(i,j)$ and $D^8(i,j)$ are very similar, the essential difference is the computations of $\mathrm{len}^6_{x_1x_2}(i,j,1)$ and $\mathrm{len}^8_{x_1x_2x_3}(i,j,1)$. This similarity follows because codon alignments of type 6 and type 8 end in the same way. By "end in the same way" we mean that the events described on the codon $a_1^i a_2^i a_3^i$ are the same. It follows from Figure 5.7 that a codon alignment of type 11 also ends in the same way as codon alignments of type 6 and 8.

The similarities between the computations of $D^t(i,j)$ for $t = 6, 8, 11$ can be used to reduce the total number of tables necessary to maintain. We can replace the three tables $L^6_{x_1x_2}$, $L^8_{x_1x_2x_3}$ and $L^{11}_{x_1x_2x_3}$ by a single table, $L^{6,8,11}_{x_1x_2x_3}$, where entry (i,j) holds the minimum over entry (i,j) in the three tables it replaces. Similarly, we can replace the tables $F^6_{x_1x_2x_3}$, $F^8_{x_1x_2x_3}$ and $F^{11}_{x_1x_2x_3}$ by a single table $F^{6,8,11}_{x_1x_2x_3}$. We can compute $L^{6,8,11}_{x_1x_2x_3}(i,j)$ and $F^{6,8,11}_{x_1x_2x_3}(i,j)$ by expressions similar to Equations 5.23 and 5.24, all we essentially have to do is to replace the term $\mathrm{len}^8_{x_1x_2x_3}(i,j,1)$ by

$$\mathrm{len}^{6,8,11}_{x_1x_2x_3}(i,j,1) = \min_{t=6,8,11} \mathrm{len}^t_{x_1x_2x_3}(i,j,1)\,, \tag{5.27}$$

where we in order to ensure that the terms $\mathrm{len}^t_{x_1x_2x_3}(i,j,1)$, for $t = 6, 8, 11$, all describe the same part of the total cost need to redefine $\mathrm{len}^6_{x_1x_2x_3}(i,j,1)$ as $\mathrm{len}^6_{x_1x_2}(i,j,1) + c_p^*(x_1x_2x_3, b_1^j b_2^j b_3^j)$. We introduce $D^{6,8,11}(i,j)$ as the minimum of $D^t(i,j)$ over $t = 6, 8, 11$. We can compute $D^{6,8,11}(i,j)$ by using $L^{6,8,11}_{x_1x_2x_3}$ and

$F^{6,8,11}_{x_1x_2x_3}$ in expressions similar to Equations 5.25 and 5.26. The computation of $D^{6,8,11}(i,j)$ requires us to compute only $64 + 64 = 128$ table entries while the individual computations of $D^t(i,j)$, for $t = 6, 8, 10$, require us to compute $80 + 128 + 128 = 336$ table entries.

Further inspections of Figure 5.7 shows that codon alignments of type 7, 9 and 10 also end in the same way. Hence, we can also combine the computation of $D^t(i,j)$, for $t = 7, 9, 10$, into the computation of $D^{7,9,10}(i,j)$. Finally, to compute $D(i,j)$ by Equation 5.5 we must minimize over $D^1(i,j)$, $D^2(i,j)$, $D^3(i,j)$, $D^4(i,j)$, $D^5(i,j)$, $D^{6,8,11}(i,j)$ and $D^{7,9,10}(i,j)$. In total this computation requires us to compute $1 + 7 + 68 + 68 + 128 + 128 = 400$ table entries.

5.7 Conclusion

We are working on implementing the alignment algorithm described in the previous section in order to compare it to the heuristic alignment algorithm described in [84]. The heuristic algorithm allows frame shifts, so an obvious extension of our exact algorithm would be to allow frame shifts, e.g. to allow insertion-deletions of arbitrary length. This however makes it difficult to split the evaluation of the alignment cost into small independent subproblems (codon alignments) of known size.

Another interesting extension would be to annotate the DNA sequence with more information. For example, if the DNA sequence codes in more than one reading frame (overlapping reading frames) then the DNA sequence should be annotated with all the amino acid sequences encoded and the combined cost of a nucleotide event should summarize the cost of changes induced on all the amino acid sequences encoded by the DNA sequence. This extension also makes it difficult to split the evaluation of the alignment cost into small independent subproblems. To implement these extensions efficiently it might be fruitful to investigate reasonable restrictions of the cost functions.

Appendix – Number of Codon Alignments

In Section 5.3 we observe that there between any two consecutive substitutions in a codon alignment can be an alternating sequence of insertion-deletions. In general it is difficult (if not impossible) to say anything useful about the number of insertion-deletions in a codon alignment. If we however assume that the combined gap cost function is affine, i.e. $g(k) = \alpha + \beta k$, then we can bound the number of insertion-deletions in a codon alignment as follows.

Consider the two substrings $a_{3i'+1}a_{3i'+2}\cdots a_{3i}$ and $b_{3j'+1}b_{3j'+2}\cdots b_{3j}$ of a and b. One possible alignment of the two substrings is a deletion of length $k_1 = i - i'$ and an insertion of length $k_2 = j - j'$. The cost of this alignment is

$$2\alpha + \beta(k_1 + k_2). \tag{5.28}$$

Another possible alignment of the two substrings is a codon alignment that describes three substitutions and c insertion-deletions. Observe that the combined length of the c insertion-deletions is $k_1 + k_2 - 2$. Remember that the cost of

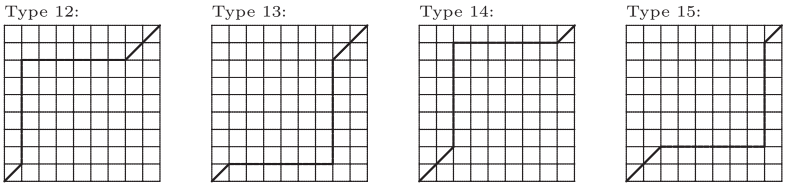


Figure 5.14: The addition four codon alignments

an insertion-deletion within a codon alignment not only depends on the length but also on the witness. If we use use *cost*(*subs*) to denote the cost of the three substitutions and *cost*(*witnesses*) to denote the cost of the c witnesses, then we can write the cost of the codon alignment of the two substrings as

$$cost(subs) + cost(witnesses) + c\alpha + \beta(k_1 + k_2 - 2). \tag{5.29}$$

If the codon alignment is part of the optimal alignment of a and b then the cost given by Equation 5.29 must be less than the cost given by Equation 5.28. Since *cost*(*subs*) and *cost*(*witnesses*) are at least zero, this is only possible if

$$c\alpha + \beta(k_1 + k_2 - 2) < 2\alpha + \beta(k_1 + k_2). \tag{5.30}$$

If we assume that $\alpha > 0$, this simplifies to

$$c < 2(1 + \frac{\beta}{\alpha}). \tag{5.31}$$

Hence, if we assume that $\alpha \geq 2\beta$ then we only have to consider codon alignments that describes at most two insertion-deletions. In addition to the eleven types of codon alignments in Figure 5.7 this includes the four types of codon alignments in Figure 5.14. Extending the algorithm in Section 5.5 to handle the additional four types of codon alignments, i.e. to compute $D^t(i, j)$ for $t = 12, 13, 14, 15$, can be done by the technique used to handle the other types of codon alignment that describes two gaps. The extension roughly doubles the number of table entries that is computed during the computation of an optimal alignment.

Chapter 6

Measures on Hidden Markov Models

The paper *Measures on Hidden Markov Models* presented in this chapter has been published in part as a technical report [126] and a conference paper [127].

[126] R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Measures on hidden Markov models. Technical Report RS-99-6, BRICS, April 1999.

[127] R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden Markov models. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 178–186, 1999.

The technical report extends the conference paper by adding a section about generalizing the measures to other types of hidden Markov models than left-right models. Except for minor typographical changes the content of this chapter is equal to the technical report [126]. An implementation of the method for comparison of left-right hidden Markov models presented in this chapter is available at `www.daimi.au.dk/∼cstorm/hmmcomp`.

Measures on Hidden Markov Models

Rune B. Lyngsø[*] Christian N. S. Pedersen[†] Henrik Nielsen[‡]

Abstract

Hidden Markov models were introduced in the beginning of the 1970's as a tool in speech recognition. During the last decade they have been found useful in addressing problems in computational biology such as characterizing sequence families, gene finding, structure prediction and phylogenetic analysis. In this paper we propose several measures between hidden Markov models. We give an efficient algorithm that computes the measures for left-right models, e.g. profile hidden Markov models, and discuss how to extend the algorithm to other types of models. We present an experiment using the measures to compare hidden Markov models for three classes of signal peptides.

6.1 Introduction

A hidden Markov model describes a probability distribution over a potentially infinite set of sequences. It is convenient to think of a hidden Markov model as generating a sequence according to some probability distribution by following a first order Markov chain of states, called the path, from a specific start-state to a specific end-state and emitting a symbol according to some probability distribution each time a state is entered. One strength of hidden Markov models is the ability efficiently to compute the probability of a given sequence as well as the most probable path that generates a given sequence. Hidden Markov models were introduced in the beginning of the 1970's as a tool in speech recognition. In speech recognition the set of sequences might correspond to digitized sequences of human speech and the most likely path for a given sequence is the corresponding sequence of words. Rabiner [166] gives a good introduction to the theory of hidden Markov models and their applications to speech recognition.

Hidden Markov models were introduced in computational biology in 1989 by Churchill [38]. Durbin et al. [46] and Eddy [48, 49] are good overviews of the use of hidden Markov models in computational biology. One of the most popular applications is to use them to characterize sequence families by using

[*] Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `rlyngsoe@daimi.au.dk`.

[†] Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `cstorm@brics.dk`.

[‡] Center for Biological Sequence Analysis, Center of the Danish National Research Foundation, Technical University of Denmark, Denmark. E-mail: `hnielsen@cbs.dtu.dk`.

so called profile hidden Markov models introduced by Krogh et al. [111]. For a profile hidden Markov model the probability of a given sequence indicates how likely it is that the sequence is a member of the modeled sequence family, and the most likely path for a given sequence corresponds to an alignment of the sequence against the modeled sequence family.

An important advance in the use of hidden Markov models in computational biology within the last two years, is the fact that several large libraries of profile hidden Markov models have become available [49]. These libraries not only make it possible to classify new sequences, but also open up the possibility of comparing sequence families by comparing the profiles of the families instead of comparing the individual members of the families, or of comparing entire sequence families instead of the individual members of the family to a hidden Markov model constructed to model a particular feature. To our knowledge little work has been published in this area, except for alignment of profiles [69].

In this paper we propose measures for hidden Markov models that can be used to address this problem. The measures are based on what we call the co-emission probability of two hidden Markov models. We present an efficient algorithm that computes the measures for profile hidden Markov models and observe that the left-right architecture is the only special property of profile hidden Markov models required by the algorithm. We describe how to extend the algorithm to broader classes of models and how to approximate the measures for general hidden Markov models. The method can easily be adapted to various special cases, e.g. if it is required that paths pass through certain states.

As the algorithm we present is not limited to profile hidden Markov models, we have chosen to emphasize this generality by presenting an application to a set of hidden Markov models for signal peptides. These models do not strictly follow the profile architecture and consequently cannot be compared using profile alignment [69].

The rest of the paper is organized as follows. In Section 6.2 we discuss hidden Markov models in more detail. In Section 6.3 we introduce the co-emission probability of two hidden Markov models and formulate an algorithm for computing this probability of two profile hidden Markov models. In Section 6.4 we use the co-emission probability to formulate several measures between hidden Markov models. In Section 6.5 we discuss extensions to more general models. In Section 6.6 we present an experiment using the method to compare three classes of signal peptides. Finally in Section 6.7 we briefly discuss how to compute relaxed versions of the co-emission probability.

6.2 Hidden Markov Models

Let M be a hidden Markov model that generates sequences over some finite alphabet Σ with probability distribution P_M, i.e. $P_M(s)$ denotes the probability of $s \in \Sigma^*$ under model M. Like a classical Markov model, a hidden Markov model consists of a set of interconnected states. We use $P_q(q')$ to denote the probability of a transition from state q to state q'. These probabilities are usually called *state transition probabilities*. The transition structure of a hidden

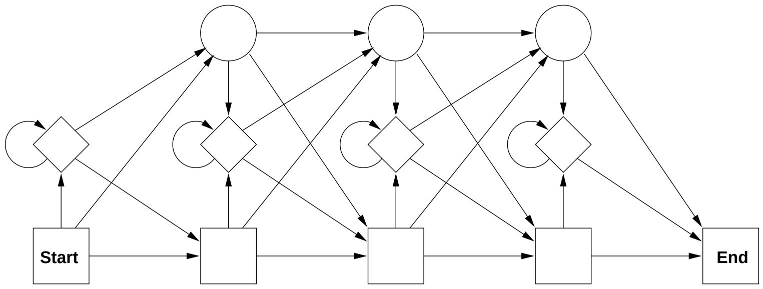


Figure 6.1: The transition structure of a profile hidden Markov model. The squares are the match-states, the diamonds are the insert-states and the circles are the silent delete-states.

Markov model is often shown as a directed graph with a node for each state, and an edge between two nodes if the corresponding state transition probability is non-zero. Figure 6.1 shows an example of a transition structure. Unlike a classical Markov model, a state in a hidden Markov model can generate or emit a symbol according to a local probability distribution over all possible symbols. We use $P_q(\sigma)$ to denote the probability of generating or emitting symbol $\sigma \in \Sigma$ in state q. These probabilities are usually called *symbol emission probabilities*. If a state does not have symbol emission probabilities we say that the state is a silent state.

It is often convenient to think of a hidden Markov model as a generative model, in which a run generates or emits a sequence $s \in \Sigma^*$ with probability $P_M(s)$. A run of a hidden Markov model begins in a special start-state and continues from state to state according to the state transition probabilities until a special end-state is reached. Each time a non-silent state is entered, a symbol is emitted according to the symbol emission probabilities of the state. A run thus results in a Markovian sequence of states as well as a generated sequence of symbols. The name "hidden Markov model" comes from the fact that the Markovian sequence of states, also called the path, is hidden, while only the generated sequence of symbols is observable.

Hidden Markov models have found applications in many areas of computational biology, e.g. gene finding [110] and protein structure prediction [175], but probably the most popular use is as *profiles* for sequence families. A profile is a position-dependent scoring scheme that captures the characteristics of a sequence family, in the sense that the score peaks around members of the family. Profiles are useful when searching for unknown members of a sequence family and several methods have been used to construct and use profiles [71, 120, 180]. Krogh *et al.* [111] realized that simple hidden Markov models, which they called profile hidden Markov models, were able to capture all other profile methods.

The states of a profile hidden Markov model are divided into match-, insert- and delete-states. Figure 6.1 illustrates the transition structure of a simple profile hidden Markov model. Note the highly repetitive transition structure. Each of the repeated elements consisting of a match-, insert- and delete-state models a position in the consensus sequence for the sequence family. The silent delete-state makes it possible to skip a position while the self-loop on the insert-state makes it possible to insert one or more symbols between two positions. Another distinctive feature of the structure of profile hidden Markov models is the absence of cycles, except for the self-loops on the insert-states. Hidden Markov models with this property are generally referred to as left-right [99] (or sometimes Bakis [20]) models, as they can be drawn such that all transitions go from left to right.

The state transition and symbol emission probabilities of a profile hidden Markov model (the parameters of the model) should be such that $P_M(s)$ is significant if s is a member of the sequence family. These probabilities can be derived from a multiple alignment of the sequence family, but more importantly, several methods exist to estimate them (or train the model) if a multiple alignment is not available [21, 46, 49].

6.3 Co-Emission Probability of Two Models

When using a profile hidden Markov model, it is sometimes sufficient just to focus on the most probable path through the model, e.g. when using a profile hidden Markov model to generate alignments. It is, however, well known that profile hidden Markov models possess a lot more information than the most probable paths, as they allow the generation of an infinity of sequences, each by a multitude of paths. Thus, when comparing two profile hidden Markov models, one should look at the entire spectrum of sequences and probabilities.

In this section we will describe how to compute the probability that two profile hidden Markov models independently generate the same sequence, that is for models M_1 and M_2 generating sequences over an alphabet Σ we compute

$$\sum_{s \in \Sigma^*} P_{M_1}(s) P_{M_2}(s). \tag{6.1}$$

We will call this the *co-emission probability* of the two models. The algorithm we present to compute the co-emission probability is a dynamic programming algorithm similar to the algorithm for computing the probability that a hidden Markov model will generate a specific sequence [46, Chapter 3]. We will describe how to handle the extra complications arising when exchanging the sequence with a profile hidden Markov model.

When computing the probability that a hidden Markov model M generates a sequence $s = s_1 \ldots s_n$, a table indexed by a state from M and an index from s is usually built. An entry (q, i) in this table holds the probability of being in the state q in M and having generated the prefix $s_1 \ldots s_i$ of s. We will use a similar approach to compute the co-emission probability. Given two hidden Markov models M_1 and M_2, we will describe how to build a table A indexed by states from the two hidden Markov models, such that the entry $A(q, q')$, where q is a state of M_1 and q' is a state of M_2, holds the probability of being in state q in M_1 and q' in M_2 and having independently generated identical sequences on the paths to q and q'. The entry indexed by the two end-states will then hold the probability of being in the end-states and having generated identical sequences, that is the co-emission probability.

To build the table, A, we have to specify how to fill out all entries of A. For a specific entry $A(q,q')$ this depends on the types of states q and q'. As explained in the previous section, a profile hidden Markov model has three types of states (insert-, match- and delete-states) and two special states (start and end). We postpone the treatment of the special states until we have described how to handle the other types of states. For reasons of succinctness we will treat insert- and match-states as special cases of a more general type, which we will call a *generate*-state; this type encompasses all non-silent states of the profile hidden Markov models.

The generate-state will be a merging of match-states and insert-states, thus both allowing a transition to itself and having a transition from the previous insert-state; a match-state can be viewed as a generate-state with probability zero of choosing the transition to itself, and an insert-state can be viewed as a generate-state with probability zero of choosing the transition from the previous insert-state. Note that this merging of match- and insert-states is only conceptual; we do not physically merge any states, but just handle the two types of states in a uniform way. This leaves two types of states and thus four different pairs of types. By observing that the two cases of a generate/delete-pair are symmetric and can be handled in the same way, the number of different pairs of types can be reduced to three.

The rationale behind the algorithm is to split paths up in the last transition(s)[1] and all that preceded this. We will thus need to be able to refer to the states with transitions to q and q'. In the following, m, i and d will refer to the match-, insert- and delete-state with a transition to q, and m', i' and d' to those with a transition to q'. Observe that if q (or q') is an insert-state, then i (or i') is the *previous* insert-state which, by the generate-state generalization, has a transition to q (or q') with probability zero. Let us consider each of the three different pairs of types individually.

delete/delete entry Assume that q and q' are both delete-states. As these states don't emit symbols, we just have to sum over all possible combinations of immediate predecessors of q and q', of the probability of being in these states and having independently generated identical sequences, multiplied by the joint probability of independently choosing the transitions to q and q'. For the calculation of $A(q,q')$ we thus get the equation

$$\begin{aligned}
&A(q,q') = \\
&\quad A(m,m')P_m(q)P_{m'}(q') + A(m,i')P_m(q)P_{i'}(q') + A(m,d')P_m(q)P_{d'}(q') \\
&\quad + A(i,m')P_i(q)P_{m'}(q') + A(i,i')P_i(q)P_{i'}(q') + A(i,d')P_i(q)P_{d'}(q') \\
&\quad + A(d,m')P_d(q)P_{m'}(q') + A(d,i')P_d(q)P_{i'}(q') + A(d,d')P_d(q)P_{d'}(q').
\end{aligned} \tag{6.2}$$

delete/generate entry Assume that q is a delete-state and q' is a generate-state. Envision paths leading to q and q' respectively while independently generating the same sequence. As q does not emit symbols while q' does, the path

[1] In some of the cases explained below, we will only extend the path in one of the models with an extra transition, hence the unspecificity.

to q's immediate predecessor (that is, the path to q with the actual transition to q removed) must also have generated the same sequence as the path to q'. We thus have to sum over all immediate predecessors of q, of the probability of being in this state and in q' and having generated identical sequences, multiplied by the probability of choosing the transition to q. For the calculation of $A(q, q')$ in this case we thus get the following equation

$$A(q, q') = A(m, q')P_m(q) + A(i, q')P_i(q) + A(d, q')P_d(q). \tag{6.3}$$

generate/generate entry Assume that q and q' are both generate-states. The last character in sequences generated on the paths to q and q' are generated by q and q' respectively. We will denote the probability that these two states independently generate the same symbol by p, and it is an easy observation that

$$p = \sum_{\sigma \in \Sigma} P_q(\sigma)P_{q'}(\sigma). \tag{6.4}$$

The problem with generate/generate entries is that due to the self-loops of generate states the last transitions on paths to q and q' might actually come from q and q' themselves. It thus seems that we need $A(q, q')$ to be able to compute $A(q, q')$.

So let us start out by assuming that at most one of the paths to q and q' has a self-loop transition as the last transition. Then we can easily compute the probability of being in q and q' and having independently generated the same sequence on the paths to q and q', by summing over all combinations of states with transitions to q and q' (including combinations with either q or q' but not both) the probabilities of these combinations, multiplied by p (for independently generating the same symbol at q and q') and the joint probability of independently choosing the transitions to q and q'. We denote this probability by $A_0(q, q')$, and by the above argument the equation for computing it is

$$\begin{aligned} A_0(q, q') =& p(A(m, m')P_m(q)P_{m'}(q') + A(m, i')P_m(q)P_{i'}(q') \\ &+ A(m, d')P_m(q)P_{d'}(q') + A(m, q')P_m(q)P_{q'}(q') \\ &+ A(i, m')P_i(q)P_{m'}(q') + A(i, i')P_i(q)P_{i'}(q') \\ &+ A(i, d')P_i(q)P_{d'}(q') + A(i, q')P_i(q)P_{q'}(q') \\ &+ A(d, m')P_d(q)P_{m'}(q') + A(d, i')P_d(q)P_{i'}(q') \\ &+ A(d, d')P_d(q)P_{d'}(q') + A(d, q')P_d(q)P_{q'}(q') \\ &+ A(q, m')P_q(q)P_{m'}(q') + A(q, i')P_q(q)P_{i'}(q') \\ &+ A(q, d')P_q(q)P_{d'}(q')). \end{aligned} \tag{6.5}$$

Now let us cautiously proceed, by considering a pair of paths where one of the paths has exactly one self-loop transition in the end, and the other path has at least one self-loop transition in the end. The probability – that we surprisingly call $A_1(q, q')$ – of getting to q and q' along such paths while generating the same sequences is the probability of getting to q and q' along paths that do not both have a self-loop transition in the end, multiplied by the joint probability of independently choosing the self-loop transitions, and the probability of q and q' emitting the same symbols. But this is just

$$A_1(q,q') = rA_0(q,q'), \tag{6.6}$$

where

$$r = pP_q(q)P_{q'}(q') \tag{6.7}$$

is the probability of independently choosing the self-loop transitions and emitting the same symbols in q and q'. Similarly we can define $A_k(q,q')$, and by induction it is easily proven that

$$A_k(q,q') \;=\; rA_{k-1}(q,q') \;=\; r^k A_0(q,q'). \tag{6.8}$$

As any finite path ending in q or q' must have a finite number of self-loop transitions in the end, we get

$$A(q,q') \;=\; \sum_{k=0}^{\infty} A_k(q,q') \;=\; \sum_{k=0}^{\infty} r^k A_0(q,q') \;=\; \frac{1}{1-r}A_0(q,q'). \tag{6.9}$$

Despite the fact that there is an infinite number of cases to consider, we observe that the sum over the probabilities of all these cases comes out as a geometric series that can easily be computed.

Equations 6.2, 6.3 and 6.9 give that each of the entries of table A pertaining to match- insert- and delete-states can be computed in constant time using the above equations. As for the start-states (denoted by s and s') we initialize $A(s,s')$ to 1 (as we have not started generating anything and the empty sequence is identical to itself). Otherwise, even though they do not generate any symbols, we will treat the start-states as generate states; this allows for choosing an initial sequence of delete-states in one of the models. The start-states are the only possible immediate predecessors for the first insert-states, and together with the first insert-states the only immediate predecessors of the first match- and delete-states; the equations for the entries indexed by any of these states can trivially be modified according to this. The end-states (denoted by e and e') do not emit any symbols and are thus akin to delete-states, and can be treated the same way.

The co-emission probability of M_1 and M_2 is the probability of being in the states e and e' and having independently generated the same sequences. This probability can be found by looking up $A(e,e')$. In the rest of this paper we will use $A(M_1,M_2)$ to denote the co-emission probability of M_1 and M_2.

As all entries of A can be computed in constant time, we can compute the co-emission probability of M_1 and M_2 in time $\mathrm{O}(n_1 n_2)$ where n_i denotes the number of states in M_i. The straightforward space requirement is also $\mathrm{O}(n_1 n_2)$ but can be reduced to $\mathrm{O}(n_1)$ by a standard trick [74, Chapter 11].

6.4 Measures on Hidden Markov Models

Based on the co-emission probability we define two metrics that hopefully, to some extent, express how similar the families of sequences represented by two hidden Markov models are. A problem with the co-emission probability is that the models having the largest co-emission probability with a specific model, M, usually will not include M itself, as shown by the following proposition.

Proposition 6.1 *Let M be a hidden Markov model and $p = \max\{P_M(s) \mid s \in \Sigma^*\}$. The maximum co-emission probability with M attainable for any hidden Markov model is p. Furthermore, the hidden Markov models attaining this co-emission probability with M, are exactly those models, M', for which $P_{M'}(s) > 0 \Leftrightarrow P_M(s) = p$ for all $s \in \Sigma^*$.*

Proof. Let M' be a hidden Markov model with $P_{M'}(s) > 0 \Leftrightarrow P_M(s) = p$. Then

$$\sum_{s \in \Sigma^*, P_M(s) = p} P_{M'}(s) = 1 \tag{6.10}$$

and thus the co-emission probability of M and M' is

$$\sum_{s \in \Sigma^*} P_M(s) P_{M'}(s) \;=\; \sum_{s \in \Sigma^*, P_M(s) = p} P_M(s) P_{M'}(s) \;=\; p. \tag{6.11}$$

Now let M' be a hidden Markov model with $P_{M'}(s') = p' > 0$ for some $s' \in \Sigma^*$ with $P_M(s') = p'' < p$. Then the co-emission probability of M and M' is

$$\begin{aligned} \sum_{s \in \Sigma^*} P_M(s) P_{M'}(s) &= p'p'' + \sum_{s \in \Sigma^* \setminus \{s'\}} P_M(s) P_{M'}(s) \\ &\leq p'p'' + (1 - p')p \\ &< p. \end{aligned} \tag{6.12}$$

This proves that a hidden Markov model M' has maximum co-emission probability p with M if and only if the assertion of the proposition is fulfilled. □

Proposition 6.1 indicates that the co-emission probability of two models not only depends on how alike they are, but also on how 'self-confident' the models are, that is, to what extent the probabilities are concentrated to a small subset of all possible sequences.

Another way to explain this undesirable property of the co-emission probability, is to interpret hidden Markov models – or rather the probability distribution over finite sequences of hidden Markov models – as vectors in the infinite dimensional space spanned by all finite sequences over the alphabet. With this interpretation the co-emission probability, $A(M_1, M_2)$, of two hidden Markov models, M_1 and M_2, simply becomes the inner product,

$$\langle M_1, M_2 \rangle = |M_1||M_2| \cos v, \tag{6.13}$$

of the models. In the expression on the right hand side, v is the angle between the models – or vectors – and $|M_i| = \sqrt{\langle M_i, M_i \rangle}$ is the length of M_i. One observes the direct proportionality between the co-emission probability and the length (or 'self-confidence') of the models being compared. If the length is to be completely ignored, a good measure of the distance between two hidden Markov models would be the angle between them – two models are orthogonal, if and only if they can not generate identical sequences, and parallel (actually identical as the probabilities have to sum to 1) if they express the same probability distribution. This leads to the definition of our first metric on hidden Markov models.

Definition 6.1 *Let M_1 and M_2 be two hidden Markov models, and let $A(M, M')$ denote the co-emission probability of two hidden Markov models M and M'. We define the* angle *between M_1 and M_2 as*

$$D_{angle}(M_1, M_2) = \arccos\left(A(M_1, M_2) \Big/ \sqrt{A(M_1, M_1)A(M_2, M_2)}\right).$$

Having introduced the vector interpretation of hidden Markov models, another obvious metric to consider is the standard metric on vector spaces, that is, the (Euclidean) norm of the difference between the two vectors

$$|M_1 - M_2| = \sqrt{\langle M_1 - M_2, M_1 - M_2\rangle}. \tag{6.14}$$

Considering the square of this, we obtain

$$\begin{aligned} |M_1 - M_2|^2 &= \langle M_1 - M_2, M_1 - M_2\rangle \\ &= \sum_{s\in\Sigma^*} \left(P_{M_1}(s) - P_{M_2}(s)\right)^2 \\ &= \sum_{s\in\Sigma^*} \left(P_{M_1}(s)^2 + P_{M_2}(s)^2 - 2P_{M_1}(s)P_{M_2}(s)\right) \\ &= A(M_1, M_1) + A(M_2, M_2) - 2A(M_1, M_2). \end{aligned} \tag{6.15}$$

Thus this norm can be computed based on co-emission probabilities, and we propose it as a second choice for a metric on hidden Markov models.

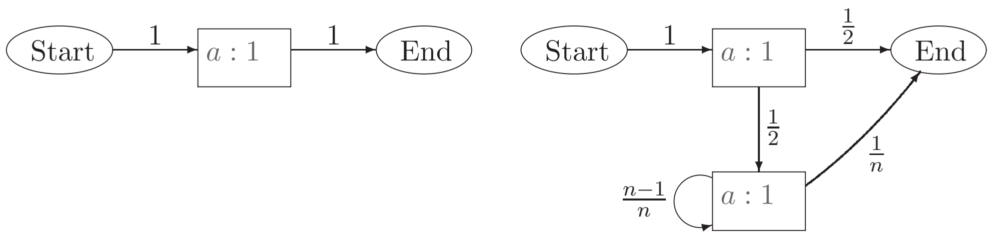
Definition 6.2 *Let M_1 and M_2 be two hidden Markov models, and $A(M, M')$ be the co-emission probability of M and M'. We define the* difference *between M_1 and M_2 as*

$$D_{diff}(M_1, M_2) = \sqrt{A(M_1, M_1) + A(M_2, M_2) - 2A(M_1, M_2)}.$$

One problem with the D_{diff} metric is that $|M_1| - |M_2| \leq D_{\text{diff}}(M_1, M_2) \leq |M_1| + |M_2|$. If $|M_1| \gg |M_2|$ we therefore get that $D_{\text{diff}}(M_1, M_2) \approx |M_1|$, and we basically only get information about the length of M_1 from D_{diff}.

The metric D_{angle} is not prone to this weakness, as it ignores the length of the vectors and focuses on the sets of most probable sequences in the two models and their relative probabilities. But this metric can also lead to undesirable situations, as can be seen from Figure 6.2 which shows that D_{angle} might not be able to discern two clearly different models. Choosing what metric to use, depends on what kind of differences one wants to highlight.

For some applications one might want a similarity measure instead of a distance measure. Based on the above metrics or the co-emission probability one can define a variety of similarity measures. We decided to examine the following two similarity measures.



(a) Hidden Markov model M_1 with $P_{M_1}(a) = 1$.

(b) Hidden Markov model M_2 with $P_{M_2}(a) = 1/2$ and $P_{M_2}(a^k) = \frac{1}{2n}(\frac{n-1}{n})^{k-2}$ for $k > 1$.

Figure 6.2: Two distinctly different models can have an arbitrarily small distance in the D_{angle} metric. It is easy to see that $A(M_1, M_1) = 1$, $A(M_1, M_2) = 1/2$ and $A(M_2, M_2) = 1/4 + 1/(8n - 4)$; for $n \to \infty$ one thus obtains $D_{\text{angle}}(M_1, M_2) \to 0$ but $D_{\text{diff}}(M_1, M_2) \to 1/2$.

Definition 6.3 *Let M_1 and M_2 be two hidden Markov models and $A(M, M')$ be the co-emission probability of M and M'. We define the* similarity *between M_1 and M_2 as*

$$\begin{aligned} S_1(M_1, M_2) &= \cos\left(D_{angle}(M_1, M_2)\right) \\ &= A(M_1, M_2) \Big/ \sqrt{A(M_1, M_1)A(M_2, M_2)} \end{aligned}$$

and

$$S_2(M_1, M_2) = 2A(M_1, M_2) / (A(M_1, M_1) + A(M_2, M_2)) .$$

One can easily prove that these two similarity measures possess the following nice properties.

1. $0 \leq S_i(M_1, M_2) \leq 1$.
2. $S_i(M_1, M_2) = 1$ if and only if $\forall s \in \Sigma^* : P_{M_1}(s) = P_{M_2}(s)$.
3. $S_i(M_1, M_2) = 0$ if and only if $\forall s \in \Sigma^* : P_{M_i}(s) > 0 \Rightarrow P_{M_{3-i}}(s) = 0$, that is, there are no sequences that can be generated by both M_1 and M_2.

The only things that might not be immediately clear are that S_2 satisfies properties 1 and 2. This however follows from

$$A(M_1, M_1) + A(M_2, M_2) - 2A(M_1, M_2) = \sum_{s \in \Sigma^*} (P_{M_1}(s) - P_{M_2}(s))^2, \quad (6.16)$$

cf. Equation 6.15, wherefore $2A(M_1, M_2) \leq A(M_1, M_1) + A(M_2, M_2)$, and equality only holds if for all sequences their probabilities in the two models are equal.

6.5 Other Types of Hidden Markov Models

Profile hidden Markov models are not by far the only type of hidden Markov models used in computational biology. Other types of hidden Markov models have been constructed for e.g. gene prediction [110] and recognition of transmembrane proteins [175]. We observe that the properties of the metrics and similarity measures introduced in the previous section do not depend on the structure of the underlying models, so once we can compute the co-emission probability of two models, we can also compute the distance between and similarity of the two models. The question is if our method can be extended to compute the co-emission probability for other types of hidden Markov models.

The first thing one can observe, is that the only feature of the underlying structure of profile hidden Markov models we use, is that they are left-right models, i.e. we can number the states such that if there is a transition from state i to state j then $i \leq j$ (if the inequality is strict, that is $i < j$, then we do not even need the geometric sequence calculation, and the calculation of the co-emission probability reduces to a calculation similar to the forward/backward calculations [46, Chapter 3]). For all left-right hidden Markov models, e.g. profile hidden Markov models extended with free insertion modules [22, 92], we can thus use recursions similar to those specified in Section 6.3 to compute the co-emission probability.

With some work the method can even be extended to all hidden Markov models where each state is part of at most one cycle, even if this cycle consists of more than the one state of the self-loop case. We will denote such models as hidden Markov models with only simple cycles. This extension can be useful when comparing models of coding DNA, that will often contain cycles with three states, or models describing a variable number of small domains. For general hidden Markov models we will have to resort to approximating the co-emission probability. In the rest of this section we will describe these two generalizations.

6.5.1 Hidden Markov Models with Only Simple Cycles

Assume that we can split M and M' into a number of disjoint cycles and single nodes, $\{C_i\}_{i \leq k}$ and $\{C'_i\}_{i \leq k'}$, such that $\{C_i\}$ and $\{C'_i\}$ are topologically sorted, i.e. for $p \in C_i$ ($p' \in C'_i$) and $q \in C_j$ ($q' \in C'_j$) and $i < j$ there is no path from q to p in M (from q' to p' in M'). To compute the co-emission probability of M and M', we will go from considering pairs of single nodes to considering pairs of cycles, i.e. we look at all nodes in a cycle at the same time.

Let C_i and $C'_{i'}$ be cycles[2] in M and M' respectively. Assume that we have already computed the co-emission probability, $A(q, q')$, for all pairs of nodes, q, q', where $q \in C_j$, $q' \in C'_{j'}$, $j \leq i$, $j' \leq i'$ and $(i, i') \neq (j, j')$. We will now describe how to compute the co-emission probability, $A(p, p')$, for all pairs of nodes, p, p', with $p \in C_i$ and $p' \in C'_{i'}$.

As with the profile hidden Markov models, cf. Section 6.3, we will proceed in a step by step fashion. We start by restricting the types of paths we consider, to get some intermediate results; we then expand the types of paths allowed – using the intermediate results – until we have covered all possible paths.

[2] If C_i or $C'_{i'}$ is not a cycle but a single node, the calculations of the co-emission probabilities pertaining to pairs of nodes from C_i and $C'_{i'}$ trivializes to calculations similar to Equation 6.17 below.

The first types of paths we consider are paths, π and π', generating identical sequences that ends in p and p', but where the immediate predecessor of p on π is not in C_i, or the immediate predecessor of p' on π' is not in $C'_{i'}$. We will denote the co-emission probability at p, p' of paths of this type as $A_e(p, p')$, as it covers the co-emission probability of paths entering the pair of cycles, $C_i, C'_{i'}$, at p, p'; it can easily be computed as

$$A_e(q, q') = \sum_{\substack{r \to q, r' \to q' \\ (r, r') \notin C_i \times C'_{i'}}} P_r(q) P_{r'}(q') A(r, r') \sum_{\sigma \in \Sigma} P_q(\sigma) P_{q'}(\sigma), \tag{6.17}$$

where $r \to q$ ($r' \to q'$) denotes that there is a transition from r to q in M (from r' to q' in M'). Here we assume that both q and q' are non-silent states; if both are silent, the sum over all symbols factor, $\sum_{\sigma \in \Sigma} P_q(\sigma) P_{q'}(\sigma)$ (the probability that q and q' generates identical symbols), should be omitted, and if one is silent and the other non-silent, the sum should furthermore only be over non-C_i (or non-$C'_{i'}$) predecessors of the silent state.

Before we proceed further, we will need some definitions that allow us to talk about successors of states and successors of pairs of states in $C_i, C'_{i'}$, and some related probabilities.

Definition 6.4 *Let $q \in C_i$ ($q' \in C'_{i'}$). The* successor *of q in C_i (q' in $C'_{i'}$) is the unique state $r \in C_i$ ($r' \in C'_{i'}$) for which there is a transition from q to r (from q' to r').*

The uniqueness of the successor follows from the requirement that the models only have simple cycles. For successors of pairs of states things are a little bit more complicated, as we want the successor of a pair to be the unique pair to which we can get to, generating the same number of symbols (zero or one) using one transition in one or both models. This is captured by Definition 6.5.

Definition 6.5 *Let $q \in C_i$ and $q' \in C'_{i'}$. The* successor *of q, q' in $C_i, C'_{i'}$,* suc(q, q'), *is the pair of states r, r' where*

- *if the successor of q in C_i is silent or the successor of q' in $C'_{i'}$ is non-silent, then r is the successor of q; otherwise $r = q$.*
- *if the successor of q' in $C'_{i'}$ is silent or the successor of q in C_i is non-silent, then r' is the successor of q'; otherwise $r' = q'$.*

By this definition the successor of a pair of states, q, q', is the pair of successors of q and q' if both successors are silent or both successors are non-silent states. If the successor of q is a non-silent state and the successor of q' is a silent state then the successor of q, q' is the pair consisting of q and the successor of q'.

We will use $P_{q,q'}(\text{suc}(q, q'))$ to denote the probability of getting from q, q' to $\text{suc}(q, q')$ generating identical symbols. If $r, r' = \text{suc}(q, q')$ are both non-silent, then $P_{q,q'}(r, r') = P_q(r) P_{q'}(r') \sum_{\sigma \in \Sigma} P_r(\sigma) P_{r'}(\sigma)$; if one or both are silent, the sum over all symbols factor, $\sum_{\sigma \in \Sigma} P_r(\sigma) P_{r'}(\sigma)$, should be omitted, and if only r (r') is silent, the $P_{q'}(r')$ factor ($P_q(r)$ factor) should furthermore be omitted as $q' = r'$ (as $q = r$).

More generally we will use $P_{q,q'}(r,r')$, where $q,r \in C_i$ and $q',r' \in C'_{i'}$, to denote the probability of getting from q,q' to r,r' generating identical sequences without cycling, i.e. by just starting in q,q' and going through successors until we reach r,r' the first time. We resolve the ambiguity of the meaning of $P_{q,q'}(q,q')$ by setting $P_{q,q'}(q,q') = 1$. To ease notation in the following, we furthermore define $P'_{q,q'}(r,r') = P_{q,q'}(\mathrm{suc}(q,q'))P_{\mathrm{suc}(q,q')}(r,r')$. The probability $P'_{q,q'}(q,q')$ is thus the probability of going through one full cycle of successors to q,q' until we are back at q,q'; if $r,r' \neq q,q'$ then $P'_{q,q'}(r,r') = P_{q,q'}(r,r')$.

One can observe that r,r' might not be anywhere in the sequence of successors starting at q,q'. If we can get to r,r' from q,q' going through consecutive successors, then there is a pair of paths from q to r and from q' to r', respectively, generating an equal number of symbols. Such a pair of paths does not necessarily exist. E.g. assume there is an even number of non-silent states in both C_i and $C'_{i'}$, and that the successor, r, of q in C_i is non-silent. Any path that starts in q and ends in r will generate an uneven number of symbols, while any path starting and ending in $q' \in C'_{i'}$ will generate an even number of symbols. It is thus impossible to get from q,q' to r,q' going through successors.

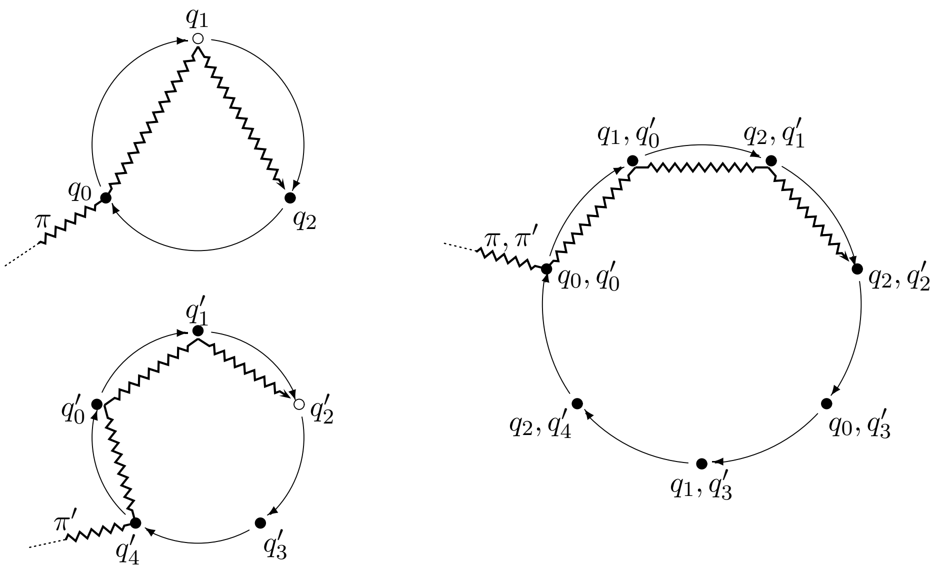
More formally, let $d_q(r)$ (resp. $d_{q'}(r')$) denote the number of non-silent states we go through going from q to r in C_i (from q' to r' in $C'_{i'}$), and let h (h') denote the total number of non-silent states in C_i (in $C'_{i'}$). Then by similar reasoning as in the above, we can get from q,q' to r,r' going through successors if and only if $d_q(r) \equiv d_{q'}(r') \mod \gcd(h,h')$. It is evident that we can always get back to q,q' when starting in q,q', and thus the pairs of states from $C_i, C'_{i'}$ can be partitioned into cycles of consecutive successors, cf. Figure 6.3. If it is possible to get from q,q' to p,p' generating an equal number of symbols, i.e. q,q' and p,p' are in the same cycle of pairs, we will say that q,q' and p,p' belong to the same *class*, as the partition of pairs in this manner is actually a partition into equivalence classes.

We are now ready to compute the probability of getting simultaneously to p and p' having generated identical sequences, without having been simultaneously in p and p' previously on the paths. This is

$$A_0(p,p') = \sum_{\substack{q,q' \text{ belongs to the} \\ \text{same class as } p,p'}} A_e(q,q')P_{q,q'}(p,p') \tag{6.18}$$

as we sum over all possible pairs, q,q', where paths ending in p,p' can have entered $C_i, C'_{i'}$. It is similar to $A_0(p,p')$ for profile hidden Markov models in the sense, that it is the probability of reaching p and p' having generated identical sequences without having looped through p,p' previously.

To compute the A_0 entries efficiently for all pairs of states in a class, we exploit the fact that $P_{q,q'}(p,p') = P_{q,q}(\mathrm{suc}(q,q'))P_{\mathrm{suc}(q,q')}(p,p')$ (for $q,q' \neq p,p'$); we can thus compute $A_0(p,p')$ in an incremental way, starting at the successor of p,p' and going through the cycle of successors, adding the A_e values and



(a) Two example cycles, $C_i = \{q_0, q_1, q_2\}$ and $C'_{i'} = \{q'_0, q'_1, q'_2, q'_3, q'_4\}$. Hollow circles denote silent states and filled circles denote non-silent states.

(b) The cycle of the class of pairs in $C_i \times C'_{i'}$ containing q_0, q'_0.

Figure 6.3: An example of a pair of cycles in M and M' and one of the induced cycles of pairs. A path, π, ending in q_2 in C_i and a path, π', ending in q'_2 in $C'_{i'}$ are shown with zigzagged lines. If we assume that the two paths generate identical sequences, then the co-emission path, π, π', ends in q_2, q'_2 in $C_i, C'_{i'}$. Though π' enters $C'_{i'}$ at q'_4, the co-emission path, π, π', enters $C_i, C'_{i'}$ at q_0, q'_0, as the first symbol in the sequence generated by π and π' that is generated by states in both C_i and $C'_{i'}$, the second last symbol of the sequence, is generated by q_0 and q'_0 respectively.

multiplying by the probability of getting to the next successor. Furthermore, as

$$\begin{aligned}
A_0(p,p')&P_{p,p'}(\mathrm{suc}(p,p')) + A_e(\mathrm{suc}(p,p')) \\
&= \sum_{\substack{q,q' \text{ belongs to the} \\ \text{same class as } p,p'}} A_e(q,q')P_{q,q'}(p,p')P_{p,p'}(\mathrm{suc}(p,p')) \\
&\qquad\qquad + A_e(\mathrm{suc}(p,p'))P_{\mathrm{suc}(p,p')}(\mathrm{suc}(p,p')) \\
&= A_0(\mathrm{suc}(p,p')) + A_e(\mathrm{suc}(p,p'))P'_{\mathrm{suc}(p,p')}(\mathrm{suc}(p,p'))
\end{aligned} \tag{6.19}$$

we do not need to start from scratch when computing A_0 for the other pairs that belong to the same class as p, p' – which would require time proportional to the square of the number of pairs in the class – but can reuse $A_0(p,p')$ to compute $A_0(\mathrm{suc}(p,p'))$ in constant time. Finally we observe that

$$A(p,p') = \sum_{i=0}^{\infty} P'_{p,p'}(p,p')^i A_0(p,p') = \frac{1}{1 - P'_{p,p'}(p,p')} A_0(p,p') \tag{6.20}$$

Algorithm 6.1 Computation of the co-emission probabilities at all pairs of states that are in the same class as p, p'.

```
$q, q' = p, p'$
$AccumulatedP = A_e(p, p')$
$r = 1$
while $\mathrm{suc}(q, q') \neq p, p'$ do
    $AccumulatedP = AccumulatedP \cdot P_{q,q'}(\mathrm{suc}(q, q')) + A_e(\mathrm{suc}(q, q'))$
    $r = r \cdot P_{q,q'}(\mathrm{suc}(q, q'))$
    $q, q' = \mathrm{suc}(q, q')$
end while
$r = r \cdot P_{q,q'}(\mathrm{suc}(q, q'))$
repeat /* $AccumulatedP = A_0(q, q')$ and $r = P'_{q,q'}(q, q')$ */
    $A(q, q') = AccumulatedP \cdot \frac{1}{1-r}$
    $AccumulatedP$
            $= AccumulatedP \cdot P_{q,q'}(\mathrm{suc}(q, q')) + (1 - r) \cdot A_e(\mathrm{suc}(q, q'))$
    $q, q' = \mathrm{suc}(q, q')$
until $\mathrm{suc}(q, q') = p, p'$
```

and

$$P'_{p,p'}(p, p') = P'_{q,q'}(q, q') \tag{6.21}$$

for all q, q' that belong to the same class as p, p'. This allows us to formulate Algorithm 6.1 for computing the co-emission probability at all pairs in a cycle.

It is an easy observation that we run through all pairs of the class twice – once in the *while*-loop and once in the *repeat*-loop – thus using time proportional to the number of pairs in the class to compute the co-emission probabilities at each pair. Therefore, the overall time for handling the entries pertaining to the pair of cycles, $C_i, C'_{i'}$, is $\mathrm{O}(|C_i||C'_{i'}|)$ once we have computed the A_e entries; thus the time used to compute the co-emission probability of two hidden Markov models with only simple cycles is proportional to the product of the number of transitions in the two models. This is comparable to the complexity of $\mathrm{O}(n_1 n_2)$ for profile hidden Markov models, as this result relied on there only being a constant number of transitions to each state. In general we can compute the co-emission probability of two hidden Markov models, M_1 and M_2, with only simple cycles – including left-right hidden Markov models – in time $\mathrm{O}(m_1 m_2)$, where m_i denotes the number of transitions in M_i.

6.5.2 General Hidden Markov Models

For more complex hidden Markov models, let us examine what is obtained by iterating the calculations. Let $A'_i(q, q')$ be the value computed for entry (q, q') in the i'th iteration. If we assume that q and q' are either both silent or both non-silent states, then we can compute the new entry for (q, q') as

$$A'_{i+1}(q, q') = p \sum_{\substack{r \to q \\ r' \to q'}} A'_i(r, r') P_r(q) P_{r'}(q'), \tag{6.22}$$

where p is as defined in Equation 6.4 if q and q' are non-silent states, and is 1 if q and q' are silent states. If q and q' are of different types, the summation should only be over the predecessors of the silent state as in Equation 6.3. In each iteration we thus extend co-emission paths with one pair of states, and $A'_i(q, q')$ is the probability of getting to q, q' having generated identical sequences on a co-emission path of length i.

The resemblance of this iterated computation to the previous calculation of A_i is evident, but a well-known mathematical sequence is not easily recognizable in Equation 6.22. Instead we observe that $A'_i(q, q')$ holds the probability of being in states q and q' and having generated identical prefixes in the two models after i iterations. If we assume that the only transitions from the end-states are self-loops with probability 1 (this makes the $A'_i(e, e')$ entry accumulate the probabilities of generating identical sequences after at most i iterations), then

$$A'_i(e, e') \;\leq\; A(M_1, M_2) \;\leq \sum_{q \in M_1,\, q' \in M_2} A'_i(q, q') \tag{6.23}$$

where $A(M_1, M_2)$ is the true co-emission probability of M_1 and M_2. This follows from the fact, that to generate identical sequences we must either already have done so, or at least have generated identical prefixes so far.

Now assume that for any two states, we can choose transitions to non-silent states (or the end-states) and emit different symbols with probability at least $1 - c$ where $c < 1$. Then the total weight with which $A'_i(q, q')$ contributes to the entries – not counting the special (e, e') entry – of A'_{i+1} is at most c. Thus

$$\sum_{\substack{q \in M_1,\, q' \in M_2 \\ (q,q') \neq (e,e')}} A'_{i+1}(q, q') \leq c \sum_{\substack{q \in M_1,\, q' \in M_2 \\ (q,q') \neq (e,e')}} A'_i(q, q') \tag{6.24}$$

and by induction we get

$$\sum_{q \in M_1,\, q' \in M_2} A'_i(q, q') - A'_i(e, e') \;=\; \sum_{\substack{q \in M_1,\, q' \in M_2 \\ (q,q') \neq (e,e')}} A'_i(q, q') \;\leq\; c^i, \tag{6.25}$$

which shows that the iteration method approximates the co-emission probability exponentially fast.

Though our assumption about the non-zero probability of choosing transitions and emissions such that we generate different symbols in the two models is valid for most, if not all, hidden Markov models used in practice, it is not even necessary. If d is the minimum number of paired transitions we have to follow from q and q' to get to the end-states[3] or states where we can emit different symbols after having generated identical prefixes, and c' is the probability of staying on this path and emit different symbols, we still get the exponential approximation of Equation 6.25 with $c = (c')^{1/d}$. By these arguments we can approximate the co-emission probabilities and thus the metrics and similarity measures presented in Section 6.4 of arbitrary hidden Markov models exponentially fast.

[3] The end-states ensures that d exists – if we can not get to e and e', then we can not pass through q and q' and generate identical sequences. Therefore we may just as well ignore the (q, q') entry.

6.6 Experiments

We have implemented the method described in the previous sections for computing the co-emission probabilities of two left-right models. The program also computes the two metrics and two similarity measures described in this paper and is currently available at `www.brics.dk/~cstorm/hmmcomp`. The program was used to test the four measures in a comparison of hidden Markov models for three classes of secretory signal peptides – cleavable N-terminal sequences which target secretory proteins for translocation over a membrane.

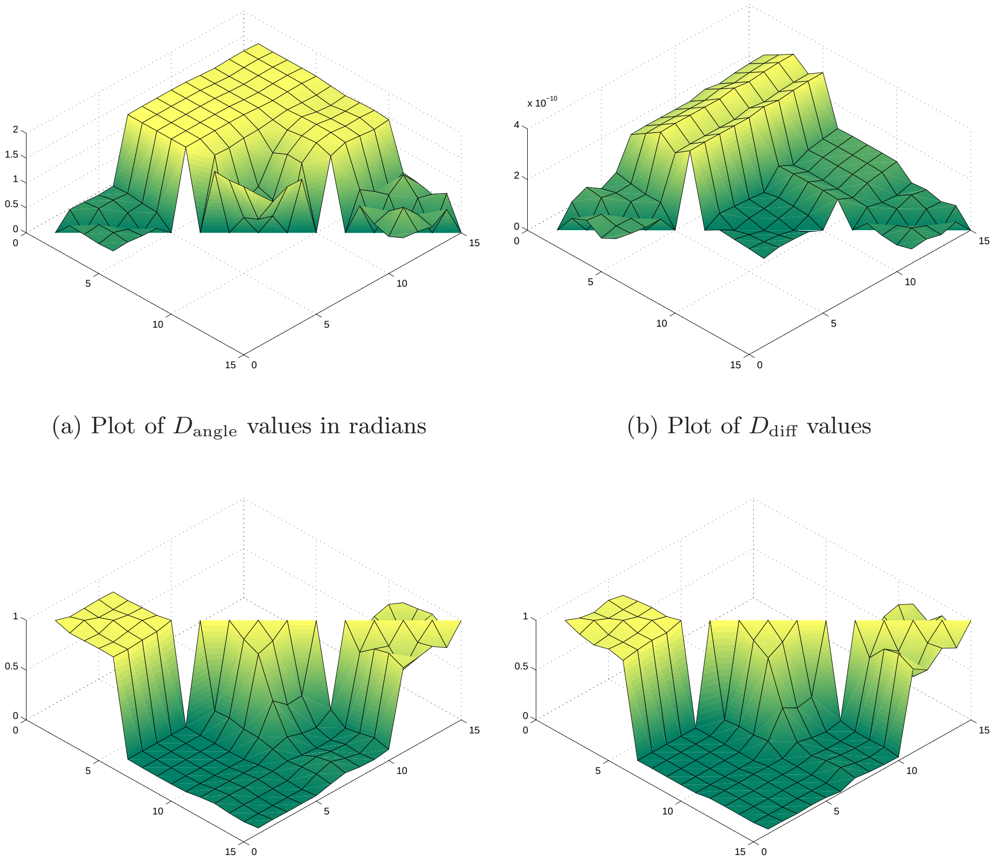
Signal peptides do not have a well-defined consensus motif, but they do share a common structure: an N-terminal region with a positive charge, a stretch of hydrophobic residues, and a region of more polar regions containing the cleavage site, where two positions are partially conserved [193]. There are statistical differences between prokaryotic and eukaryotic signal peptides concerning the length and composition of these regions [194, 152], but the distributions overlap, and in some cases, eukaryotic and prokaryotic signal peptides are found to be functionally interchangeable [25].

The hidden Markov model used here is not a profile HMM, since signal peptides of different proteins are not necessarily related, and therefore do not constitute a sequence family that can be aligned in a meaningful way. Instead, the signal peptide model is composed of three region models, each having a characteristic amino acid composition and length distribution, plus seven states modeling the cleavage site – see Nielsen and Krogh [153] for a detailed description. A combined model with three branches was used to distinguish between signal peptides, signal anchors (a subset of transmembrane proteins), and non-secretory proteins; but only the part modeling the signal peptide plus the first few positions after the cleavage site has been used in our comparisons.

The same architecture was used to train models of three different signal peptide data sets: eukaryotes, Gram-negative bacteria (with a double membrane), and Gram-positive bacteria (with a single membrane). For cross-validation of the predictive performance, each model was trained on five different training/test set partitions, with each training set comprising 80% of the data – i.e., any two training sets have 75% of the sequences in common.

The comparisons of the models are shown in Figures 6.4 and 6.5. In general, models trained on cross-validation sets of the same group are more similar than models trained on data from different groups, and the two groups of bacteria are more similar to one another than to the eukaryotes. However, there are some remarkable differences between the measures. According to D_{diff}, the two bacterial groups are almost as similar as the cross-validation sets, but according to D_{angle} and the similarity measures, they are almost as dissimilar as the bacterial/eukaryotic comparisons.

This difference actually reflects the problem with the D_{diff} measure discussed in Section 6.4. The distribution of sequences for models trained on eukaryotic data are longer in the vector interpretation, i.e. the probabilities are more concentrated, than the distributions for models trained on bacterial data. What we mainly see in the D_{diff} values for bacterial/eukaryotic comparisons is thus the



(a) Plot of D_{angle} values in radians

(b) Plot of D_{diff} values

(c) Plot of S_1 values

(d) Plot of S_2 values

Figure 6.4: Plots of the results obtained with the different measures. Models 1 through 5 are the models trained on eukaryotic sequences, models 6 through 10 are the models trained on Gram-positive bacterial sequences, and models 11 through 15 are the models trained on Gram-negative bacterial sequences. This gives 9 blocks, each of 25 entries, of different pairs of groups of organisms compared, but as all the measures are symmetric we have left out half the blocks showing comparisons between different groups of organisms. This should increase clarity, as no parts of the plots are hidden behind peaks.

length of the eukaryotic models. This reflects two properties of eukaryotic signal peptides: they have a more biased amino acid composition in the hydrophobic region that comprises a large part of the signal peptide sequence; and they are actually *shorter* than their bacterial counterparts, thus raising the probability of the most probable sequences generated by this model.

D_{angle} also shows that the differences within groups are larger in the Gram-positive group than in the others. This may simply reflect the smaller sample size in this group (172 sequences vs. 356 for the Gram-negative bacteria and 1137 for the eukaryotes).

	Euk	G_{pos}	G_{neg}
Euk	0.231	1.56	1.52
G_{pos}		0.864	1.47
G_{neg}			0.461

(a) Table of D_{angle} values

	Euk	G_{pos}	G_{neg}
Euk	$6.77 \cdot 10^{-11}$	$2.56 \cdot 10^{-10}$	$2.67 \cdot 10^{-10}$
G_{pos}		$1.95 \cdot 10^{-11}$	$9.09 \cdot 10^{-11}$
G_{neg}			$4.43 \cdot 10^{-11}$

(b) Table of D_{diff} values

	Euk	G_{pos}	G_{neg}
Euk	0.967		
G_{pos}	$1.06 \cdot 10^{-2}$	0.547	
G_{neg}	$4.74 \cdot 10^{-2}$	0.102	0.866

(c) Table of S_1 values

	Euk	G_{pos}	G_{neg}
Euk	0.955		
G_{pos}	$1.78 \cdot 10^{-3}$	0.511	
G_{neg}	$2.93 \cdot 10^{-2}$	$4.78 \cdot 10^{-2}$	0.839

(d) Table of S_2 values

Figure 6.5: Tables of the average values of each block plotted in Figure 6.4. The empty entries corresponds to the blocks left out in the plots.

The values of D_{angle} in between-group comparisons are quite close to the maximal $\pi/2$. Thus the distributions over sequences for models of different groups are close to being orthogonal. This might seem surprising in the light of the reported examples of functionally interchangeable signal peptides; but it does not mean that no sequences can be generated by both eukaryotic and bacterial models, only that these sequences have low probabilities compared to those that are unique for one group. In other words: if a random sequence is generated from one of these models, it may with a high probability be identified which group of organisms it belongs to.

6.7 Conclusion

Recall that the co-emission probability is defined as the probability that two hidden Markov models, M_1 and M_2, generate *completely identical* sequences, i.e. as $\sum_{s_1,s_2 \in \Sigma} P_{M_1}(s_1)P_{M_2}(s_2)$ where $s_1 = s_2$. One problem with the co-emission probability – and measures based on it – is that it can be desirable to allow sequences to be slightly different. One might thus want to loosen the restriction of "$s_1 = s_2$" to, e.g., "s_1 is a substring (or subsequence) of s_2," or even "$|s_1| = |s_2|$" ignoring the symbols of the sequences and just comparing the length distributions of the two models.

Another approach is to take the view that the two hidden Markov models do not generate independent sequences, but instead generates alignments with two sequences. Inspecting the equations for computing the co-emission probability, one observes that we require that when one model emits a symbol the other model should emit an identical symbol. This corresponds to only allowing columns with identical symbols in the produced alignments. A less restrictive approach would be to allow other types of columns, i.e. columns with two different symbols or a symbol in only one of the sequences, and weighting a column according to the difference it expresses. The modifications proposed in the previous paragraph can actually be considered special cases of this approach. Our method for computing the co-emission probability can easily be modified to encompass these types of modifications.

Acknowledgements

This work was inspired by a talk by Xiaobing Shi on his work with David States on aligning profile hidden Markov models. The authors would like to thank Bjarne Knudsen and Jotun Hein for their valuable suggestions. Finally we would like to thank Anders Krogh for his help with the software used to train and parse the models. Christian N. S. Pedersen and Henrik Nielsen are supported by the Danish National Research Foundation.

Chapter 7

Finding Maximal Pairs with Bounded Gap

The paper *Finding Maximal Pairs with Bounded Gap* presented in this chapter has been published in part as a technical report [29], a conference paper [30] and a journal paper [31]

[29] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. Technical Report RS-99-12, BRICS, April 1999.

[30] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149, 1999.

[31] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. To appear in *Journal of Discrete Algorithms*, 2000.

The technical report and the journal paper extend the conference paper by adding a section describing how to find all maximal pairs with a lower bounded gap in linear time. Except for minor typographical changes the content of this chapter is equal to the journal paper [31].

Finding Maximal Pairs with Bounded Gap

Gerth Stølting Brodal[*] Rune B. Lyngsø[†]
Christian N. S. Pedersen[*] Jens Stoye[§]

Abstract

A pair in a string is the occurrence of the same substring twice. A pair is maximal if the two occurrences of the substring cannot be extended to the left and right without making them different, and the gap of a pair is the number of characters between the two occurrences of the substring. In this paper we present methods for finding all maximal pairs under various constraints on the gap. In a string of length n we can find all maximal pairs with gap in an upper and lower bounded interval in time $O(n \log n + z)$, where z is the number of reported pairs. If the upper bound is removed the time reduces to $O(n + z)$. Since a tandem repeat is a pair with gap zero, our methods is a generalization of finding tandem repeats. The running time of our methods also equals the running time of well known methods for finding tandem repeats.

7.1 Introduction

A pair in a string is the occurrence of the same substring twice. A pair is left-maximal (right-maximal) if the characters to the immediate left (right) of the two occurrences of the substring are different. A pair is maximal if it is both left- and right-maximal. The gap of a pair is the number of characters between the two occurrences of the substring, e.g. the two occurrences of the substring *ma* in the string *maximal* form a maximal pair of *ma* with gap two.

Gusfield in [74, Section 7.12.3] describes how to use a suffix tree to report all maximal pairs in a string of length n in time $O(n + z)$ and space $O(n)$, where z is the number of reported pairs. The algorithm presented by Gusfield allows no restrictions on the gaps of the reported maximal pairs, so many of the reported pairs probably describe occurrences of substrings that are either overlapping or far apart in the string. In many applications this is unfortunate because it leads to a lot of redundant output. The problem of finding occurrences of

[*] Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `{gerth,cstorm}@brics.dk`.

[†] Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `rlyngsoe@daimi.au.dk`.

[‡] Deutsches Krebsforschungszentrum (DKFZ), Theoretische Bioinformatik, Im Neuenheimer Feld 280, 69120 Heidelberg, Germany. E-mail: `j.stoye@dkfz-heidelberg.de`.

similar substrings not too far apart has been studied in several papers, e.g. [101, 116, 169].

In the first part of this paper we describe how to find all maximal pairs in a string with gap in an upper and lower bounded interval in time $O(n \log n + z)$ and space $O(n)$. The interval of allowed gaps can be chosen such that we report a maximal pair only if the gap is between two constants c_1 and c_2; but more generally, the interval can be chosen such that we report a maximal pair only if the gap is between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, where g_1 and g_2 are functions that can be computed in constant time and $|\alpha|$ is the length of the repeated substring. This, for example, makes it possible to find all maximal pairs with gap between zero and some fraction of the length of the repeated substring. In the second part of this paper we describe how removing the upper bound $g_2(|\alpha|)$ on the allowed gaps makes it possible to reduce the running time to $O(n + z)$. The methods we present all use the suffix tree as the fundamental data structure combined with efficient merging of search trees and heap-ordered trees.

Finding occurrences of repeated substrings in a string is a widely studied problem. Much work has focused on constructing efficient methods for finding occurrences of contiguously repeated substrings. An occurrence of a substring of the form $\alpha\alpha$ is called an occurrence of a square or a tandem repeat. Several methods have been presented that in time $O(n \log n + z)$ find all z occurrences of tandem repeats in a string of length n, e.g. [41, 11, 131, 109, 177]. Methods that in time $O(n)$ decide if a string of length n contains an occurrence of a tandem repeat have also been presented, e.g. [132, 42]. Extending on the ideas presented in [42], two methods [108, 75] have been presented that find a compact representation of all tandem repeats in a string of length n in time $O(n)$. The problem of finding occurrences of contiguous repeats of substrings that are within some Hamming- or edit-distance of each other is considered in e.g. [113].

In biological sequence analysis searching for tandem repeats is used to reveal structural and functional information [74, pp. 139–142]. However, searching for exact tandem repeats can be too restrictive because of sequencing and other experimental errors. By searching for maximal pairs with small gaps (maybe depending on the length of the substring) it could be possible to compensate for these errors. Finding maximal pairs with gap in a bounded interval is also a generalization of finding occurrences of tandem repeats. Stoye and Gusfield in [177] say that an occurrence of the tandem repeat $\alpha\alpha$ is a branching occurrence of the tandem repeat $\alpha\alpha$ if and only if the characters to the immediate right of the two occurrences of α are different, and they explain how to deduce the occurrence of all tandem repeats in a string from the occurrences of branching tandem repeats in time proportional to the number of tandem repeats. Since a branching occurrence of a tandem repeat is just a right-maximal pair with gap zero, the methods presented in this paper can be used to find all tandem repeats in time $O(n \log n + z)$. This matches the time bounds of previous published methods for this problem, e.g. [41, 11, 131, 109, 177].

The rest of this paper is organized in two parts which can be read independently. In Section 7.2 we present the preliminaries necessary to read either of the two parts; we define pairs and suffix trees and describe how in general to find pairs using the suffix tree. In the first part, Section 7.3, we present the methods to find all maximal pairs in a string with gap in an upper and lower bounded interval. This part also presents facts about efficient merging of search trees which are essential to the formulation of the methods. In the second part, Section 7.4, we present the methods to find all maximal pairs in a string with gap in a lower bounded interval. This part also includes the presentation of two novel data structures, the heap-tree and the colored heap-tree, which are essential to the formulation of the methods. Finally, in Section 7.5 we summarize our work and discuss open problems.

7.2 Preliminaries

Throughout this paper S will denote a string of length n over a finite alphabet Σ. We will use $S[i]$, for $i = 1, 2, \ldots, n$, to denote the ith character of S, and use $S[i\,..\,j]$ as notation for the substring $S[i]S[i+1]\cdots S[j]$ of S. To be able to refer to the characters to the left and right of every character in S without worrying about the first and last character, we define $S[0]$ and $S[n+1]$ to be two distinct characters not appearing anywhere else in S.

In order to formulate methods for finding repetitive structures in S, we need a proper definition of such structures. An obvious definition is to find all pairs of identical substrings in S. This, however, leads to a lot of redundant output, e.g. in the string that consists of n identical characters there are $\Theta(n^3)$ such pairs. To limit the redundancy without sacrificing meaningful structures Gusfield in [74] proposes maximal pairs.

Definition 7.1 (Pair) *We say that* $(i, j, |\alpha|)$ *is a* pair *of* α *in* S *formed by* i *and* j *if and only if* $1 \leq i < j \leq n - |\alpha| + 1$ *and* $\alpha = S[i\,..\,i + |\alpha| - 1] = S[j\,..\,j + |\alpha| - 1]$. *The pair is* left-maximal *(*right-maximal*) if the characters to the immediate left (right) of two occurrences of* α *are different, i.e. left-maximal if* $S[i-1] \neq S[j-1]$ *and right-maximal if* $S[i + |\alpha|] \neq S[j + |\alpha|]$. *The pair is* maximal *if it is right- and left-maximal. The gap of a pair* $(i, j, |\alpha|)$ *is the number of characters* $j - i - |\alpha|$ *between the two occurrences of* α *in* S.

The indices i and j in a right-maximal pair $(i, j, |\alpha|)$ uniquely determine $|\alpha|$. Hence, a string of length n contains in the worst case $O(n^2)$ right-maximal pairs. The string a^n contains the worst case number of right-maximal pairs but only $O(n)$ maximal pairs. However, the string $(aab)^{n/3}$ contains $\Theta(n^2)$ maximal pairs. This shows that the worst case number of maximal pairs and right-maximal pairs in a string are asymptotically equal.

Figure 7.1 illustrates the occurrence of a pair. In some applications it might be interesting only to find pairs that obey certain restrictions on the gap, e.g. to filter out pairs of substrings that are either overlapping or far apart and thus reduce the number of pairs to report. Using the “smaller-half trick” (see Section 7.3.1) and Lemma 7.1 it can be shown that a string of length n in the worst case contains $\Theta(n \log n)$ right-maximal pairs with gap in an interval of constant size.

In this paper we present methods for finding all right-maximal and maximal pairs $(i, j, |\alpha|)$ in S with gap in a bounded interval. These methods all use the

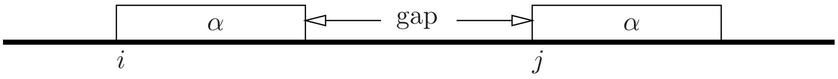


Figure 7.1: An occurrence of a pair $(i, j, |\alpha|)$ with gap $j - i - |\alpha|$.

suffix tree of S as the fundamental data structure. We briefly review the suffix tree and refer to [74] for a more comprehensive treatment.

Definition 7.2 (Suffix tree) *The* suffix tree *$T(S)$ of the string S is the compressed trie of all suffixes of $S\$$, where $\$ \notin \Sigma$. Each leaf in $T(S)$ represents a suffix $S[i \mathinner{..} n]$ of S and is annotated with the index i. We refer to the set of indices stored at the leaves in the subtree rooted at node v as the* leaf-list *of v and denote it $LL(v)$. Each edge in $T(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i \mathinner{..} n]$. We refer to the substring of S spelled by the path from the root to node v as the* path-label *of v and denote it $L(v)$.*

Several algorithms construct the suffix tree $T(S)$ in time $O(n)$, e.g. [203, 137, 189, 53]. It follows from the definition of a suffix tree that all internal nodes in $T(S)$ have out-degree between two and $|\Sigma|$. We can turn the suffix tree $T(S)$ into the binary suffix tree $T_B(S)$ by replacing every node v in $T(S)$ with out-degree $d > 2$ by a binary tree with $d - 1$ internal nodes and $d - 2$ internal edges in which the d leaves are the d children of node v. We label each new internal edge with the empty string such that the $d - 1$ nodes replacing node v all have the same path-label as node v has in $T(S)$. Since $T(S)$ has n leaves, constructing the binary suffix tree $T_B(S)$ requires adding at most $n - 2$ new nodes. Since each new node can be added in constant time, the binary suffix tree $T_B(S)$ can be constructed in time $O(n)$.

The binary suffix tree is an essential component of our methods. Definition 7.2 implies that there is an internal node v in $T(S)$ with path-label α if and only if α is the longest common prefix of $S[i \mathinner{..} n]$ and $S[j \mathinner{..} n]$ for some $1 \leq i < j \leq n$. In other words, there is a node v with path-label α if and only if $(i, j, |\alpha|)$ is a right-maximal pair in S. Since $S[i + |\alpha|] \neq S[j + |\alpha|]$ the indices i and j cannot be elements in the leaf-list of the same child of v. Using the binary suffix tree $T_B(S)$ we can thus formulate the following lemma.

Lemma 7.1 *There is a right-maximal pair $(i, j, |\alpha|)$ in S if and only if there is a node v in the binary suffix tree $T_B(S)$ with path-label α and distinct children w_1 and w_2, where $i \in LL(w_1)$ and $j \in LL(w_2)$.*

The lemma implies an approach to find all right-maximal pairs in S; for every internal node v in the binary suffix tree $T_B(S)$ consider the leaf-lists at its two children w_1 and w_2, and for every element (i, j) in $LL(w_1) \times LL(w_2)$ report a right-maximal pair $(i, j, |\alpha|)$ if $i < j$ and $(j, i, |\alpha|)$ if $j < i$. To find all maximal pairs in S the problem remains to filter out all right-maximal pairs that are not left-maximal.

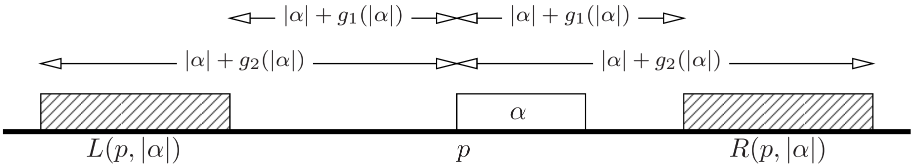

$|\alpha| + g_1(|\alpha|)$
$|\alpha| + g_1(|\alpha|)$
$|\alpha| + g_2(|\alpha|)$
$|\alpha| + g_2(|\alpha|)$
α
$L(p, |\alpha|)$
p
$R(p, |\alpha|)$

Figure 7.2: If $(p, q, |\alpha|)$ (respectively $(q, p, |\alpha|)$) is a pair with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, then one occurrence of α is at position p and the other occurrence is at a position q in the interval $R(p, |\alpha|)$ (respectively $L(p, |\alpha|)$) of positions.

7.3 Pairs with Upper and Lower Bounded Gap

We want to find all maximal pairs $(i, j, |\alpha|)$ in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, i.e. $g_1(|\alpha|) \leq j - i - |\alpha| \leq g_2(|\alpha|)$, where g_1 and g_2 are functions that can be computed in constant time. An obvious approach to solve this problem is to generate all maximal pairs in S but only report those with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. However, as explained in the previous section there might be asymptotically fewer maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ than maximal pairs in S in total. We therefore want to find all maximal pairs $(i, j, |\alpha|)$ in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ *without* generating and considering all maximal pairs in S.

A step towards finding all maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is to find all right-maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Figure 7.2 shows that if one occurrence of α in a pair with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is at position p, then the other occurrence of α must be at a position q in one of the two intervals:

$$L(p, |\alpha|) \;=\; [\, p - |\alpha| - g_2(|\alpha|) \,..\, p - |\alpha| - g_1(|\alpha|) \,] \qquad (7.1)$$

$$R(p, |\alpha|) \;=\; [\, p + |\alpha| + g_1(|\alpha|) \,..\, p + |\alpha| + g_2(|\alpha|) \,] \qquad (7.2)$$

Combined with Lemma 7.1 this gives an approach to find all right-maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$: for every internal node v in the binary suffix tree $T_B(S)$ with path-label α and children w_1 and w_2, we report for every p in $LL(w_1)$ the pairs $(p, q, |\alpha|)$ for all q in $LL(w_2) \cap R(p, |\alpha|)$ and the pairs $(q, p, |\alpha|)$ for all q in $LL(w_2) \cap L(p, |\alpha|)$.

To report the right-maximal pairs efficiently we must be able to find for every p in $LL(w_1)$ the proper elements q in $LL(w_2)$ to report it against, without looking at all the elements in $LL(w_2)$. It turns out that search trees make this possible. In this paper we use AVL trees, but other types of search trees, e.g. (a, b)-trees [90] or red-black trees [72], can also be used as long as they obey Lemmas 7.2 and 7.3 stated below. Before we can formulate algorithms we review some useful facts about AVL trees.

7.3.1 Data Structures

An AVL tree T is a balanced search tree that stores an ordered set of elements. AVL trees were introduced in [1], but are explained in almost every textbook on data structures. We say that an element e is in T, or $e \in T$, if it is stored at a node in T. For short notation we use e to denote both the element and the node at which it is stored in T. We can keep links between the nodes in T in such a way that we in constant time from the node e can find the nodes $\mathit{next}(e)$ and $\mathit{prev}(e)$ storing the next and previous element. We use $|T|$ to denote the size of T, i.e. the number of elements stored in T.

Efficient merging of two AVL trees is essential to our methods. Hwang and Lin [94] show how to merge two sorted lists using the optimal number of comparisons. Brown and Tarjan [34] show how to implement merging of two height-balanced search trees, e.g. AVL trees, in time proportional to the optimal number of comparisons. Their result is summarized in Lemma 7.2, which immediately implies Lemma 7.3.

Lemma 7.2 *Two AVL trees of size at most n and m, where $n \leq m$, can be merged together in time $O(n \log(m/n))$.*

Lemma 7.3 *Given a sorted list of elements $e_1, e_2, \ldots, e_n$ and an AVL tree T of size at most m, where $n \leq m$, we can find $q_i = \min\{x \in T \mid x \geq e_i\}$ for all $i = 1, 2, \ldots, n$ in time $O(n \log(m/n))$.*

Proof. Construct the AVL tree of the elements $e_1, e_2, \ldots, e_n$ in time $O(n)$. Merge this AVL tree with T according to Lemma 7.2, except that whenever the merge-algorithm would insert one of the elements $e_1, e_2, \ldots, e_n$ into T, we change the merge-algorithm to report the neighbor of the element in T instead. This modification does not increase the running time. □

The "smaller-half trick" is used in several methods for finding tandem repeats, e.g. [41, 11, 177]. It says that the sum over all nodes v in an arbitrary binary tree of size n of terms that are $O(n_1)$, where $n_1 \leq n_2$ are the numbers of leaves in the subtrees rooted at the two children of v, is $O(n \log n)$. Our methods for finding maximal pairs rely on a stronger version of the "smaller-half trick" hinted at in [138, Exercise 35] and used in [139, Chapter 5, page 84]; we summarize it in the following lemma.

Lemma 7.4 *If each internal node v in a binary tree with n leaves supplies a term $O(n_1 \log((n_1 + n_2)/n_1))$, where $n_1 \leq n_2$ are the number of leaves in the subtrees rooted at the two children of v, then the sum over all terms is $O(n \log n)$.*

Proof. As the terms are $O(n_1 \log((n_1+n_2)/n_1))$ we can find constants, a and b, such that the terms are upper bounded by $a + bn_1 \log((n_1 + n_2)/n_1)$. We will by induction in the number of leaves of the binary tree prove that the sum over all terms is upper bounded by $(n-1)a + bn \log n = O(n \log n)$.

If the tree is a leaf then the upper bound holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $n-1$ leaves. Consider a tree with n leaves where the number of leaves in the subtrees rooted at the two children of the root are n_1 and n_2 where $0 < n_1 \leq n_2$. According to the induction hypothesis the sum over all nodes in these two subtrees, i.e.

the sum over all nodes in the tree except the root, is bounded by $(n_1 - 1)a + bn_1 \log n_1 + (n_2 - 1)a + bn_2 \log n_2$. The entire sum is thus bounded by

$$\begin{aligned} a + bn_1 \log((n_1 + n_2)/n_1) + (n_1 - 1)a + bn_1 \log n_1 + (n_2 - 1)a + bn_2 \log n_2 \\ &= (n - 1)a + bn_1 \log n + bn_2 \log n_2 \\ &< (n - 1)a + bn_1 \log n + bn_2 \log n \\ &= (n - 1)a + bn \log n \end{aligned}$$

which proves the lemma. □

7.3.2 Algorithms

We first describe an algorithm that finds all right-maximal pairs in S with bounded gap using AVL trees to keep track of the elements in the leaf-lists during a traversal of the binary suffix tree $T_B(S)$. We then extend it to find all maximal pairs in S with bounded gap using an additional AVL tree to filter out efficiently all right-maximal pairs that are not left-maximal. Both algorithms run in time $O(n \log n + z)$ and space $O(n)$, where z is the number of reported pairs. In the following we assume, unless stated otherwise, that v is a node in the binary suffix tree $T_B(S)$ with path-label α and children w_1 and w_2 named such that $|LL(w_1)| \leq |LL(w_2)|$. We say that w_1 is the small child of v and that w_2 is the big child of v.

Right-Maximal Pairs with Upper and Lower Bounded Gap

To find all right-maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ we consider every node v in the binary suffix tree $T_B(S)$ in a bottom-up fashion, e.g. during a depth-first traversal. At every node v we use AVL trees storing the leaf-lists $LL(w_1)$ and $LL(w_2)$ at the two children of v to report the proper right-maximal pairs of the path-label α of v. The details are given in Algorithm 7.1 and explained next.

At every node v in $T_B(S)$ we construct an AVL tree, a *leaf-list tree* T, that stores the elements in $LL(v)$. If v is a leaf then we construct T directly in Step 1. If v is an internal node then $LL(v)$ is the union of the disjoint leaf-lists $LL(w_1)$ and $LL(w_2)$. By assumption $LL(w_1)$ and $LL(w_2)$ are stored in the already constructed T_1 and T_2. We construct T by merging T_1 and T_2 using Lemma 7.2, where $|T_1| \leq |T_2|$. Before constructing T in Step 2c we use T_1 and T_2 to report right-maximal pairs from node v by reporting every p in $LL(w_1)$ against every q in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$, where $L(p, |\alpha|)$ and $R(p, |\alpha|)$ are the intervals defined by (7.1) and (7.2). This is done in two steps. In Step 2a we find for every p in $LL(w_1)$ the minimum element $q_r(p)$ in $LL(w_2) \cap R(p, |\alpha|)$ and the minimum element $q_\ell(p)$ in $LL(w_2) \cap L(p, |\alpha|)$ by searching in T_2 using Lemma 7.3. In Step 2b we report pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ for every p in $LL(w_1)$ and increasing q's in $LL(w_2)$, starting with $q_r(p)$ and $q_\ell(p)$ respectively, until the gap violates the upper or lower bound.

To argue that Algorithm 7.1 finds all right-maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ by Lemma 7.1 it is sufficient to show that we for every p

in $LL(w_1)$ report all right-maximal pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. The rest follows because we at every node v in $T_B(S)$ consider every p in $LL(w_1)$. Consider the call $\mathsf{Report}(q_r(p), p + |\alpha| + g_2(|\alpha|))$ in Step 2b. The implementation of Report implies that p is reported against every q in $LL(w_2) \cap [q_r(p) \mathinner{..} p + |\alpha| + g_2(|\alpha|)]$. The construction of $q_r(p)$ and the definition of $R(p, |\alpha|)$ implies that the set $LL(w_2) \cap [q_r(p) \mathinner{..} p + |\alpha| + g_2(|\alpha|)]$ is equal to $LL(w_2) \cap R(p, |\alpha|)$. Hence, the call to Report reports all right-maximal pairs $(p, q, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Similarly the call $\mathsf{Report}(q_\ell(p), p - |\alpha| - g_1(|\alpha|))$ reports all right-maximal pairs $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

Now consider the running time of Algorithm 7.1. Building the binary suffix tree $T_B(S)$ takes time $O(n)$ [203, 137, 189, 53], and creating an AVL tree of size one at each leaf in Step 1 also takes time $O(n)$. At every internal node in $T_B(S)$ we perform Step 2. Since $|T_1| \leq |T_2|$, the searching in Step 2a and the merging in Step 2c take time $O(|T_1| \log(|T_2|/|T_1|))$ by Lemmas 7.3 and 7.2 respectively. The reporting of pairs in Step 2b takes time proportional to $|T_1|$, because we consider every p in $LL(w_1)$, plus the number of reported pairs. Summing this over all nodes gives by Lemma 7.4 that the total running time is $O(n \log n + z)$, where z is the number of reported pairs. Constructing and keeping $T_B(S)$ requires space $O(n)$. Since no element at any time is stored in more than one leaf-list tree, Algorithm 7.1 requires space $O(n)$ in total.

Theorem 7.1 *Algorithm 7.1 finds all right-maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ in time $O(n \log n + z)$ and space $O(n)$, where z is the number of reported pairs.*

Maximal Pairs with Upper and Lower Bounded Gap

We now turn our attention towards finding all maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Our approach is to extend Algorithm 7.1 to filter out all right-maximal pairs that are not left-maximal. A simple solution is to extend the procedure Report to check if $S[p-1] \neq S[q-1]$ before reporting the pair $(p, q, |\alpha|)$ or $(q, p, |\alpha|)$ in Step 2b. This solution takes time proportional to the number of inspected right-maximal pairs, and not time proportional to the number of reported maximal pairs. Even though the maximum number of right-maximal pairs and maximal pairs in strings of a given length are asymptotically equal, many strings contain significantly fewer maximal pairs than right-maximal pairs. We therefore want to filter out all right-maximal pairs that are not left-maximal *without* inspecting all right-maximal pairs. In the remainder of this section we describe one approach to achieve this.

Consider the reporting step in Algorithm 7.1. Assume that we are about to report from a node v with children w_1 and w_2. At this point the leaf-list trees T_1 and T_2, where $|T_1| \leq |T_2|$, are available and they make it possible to access the elements in $LL(w_1) = \{p_1, p_2, \ldots, p_s\}$ and $LL(w_2) = \{q_1, q_2, \ldots, q_t\}$ in sorted order. Our approach is to divide the sorted leaf-list $LL(w_2)$ into blocks of contiguous elements, such that the elements q_{i-1} and q_i are in the same block if and only if $S[q_{i-1} - 1] = S[q_i - 1]$. We say that we divide the sorted leaf-list

Algorithm 7.1 Find all right-maximal pairs in string S with bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$ and create at each leaf an AVL tree of size one that stores the index at the leaf.

2. *Reporting and merging:* When the AVL trees T_1 and T_2, where $|T_1| \leq |T_2|$, at the two children w_1 and w_2 of a node v with path-label α are available, we do the following:

   (a) Let $\{p_1, p_2, \ldots, p_s\}$ be the elements in T_1 in sorted order. For each element p in T_1 we find

   $$\begin{aligned} q_r(p) &= \min\bigl\{x \in T_2 \bigm| x \geq p + |\alpha| + g_1(|\alpha|)\bigr\} \\ q_\ell(p) &= \min\bigl\{x \in T_2 \bigm| x \geq p - |\alpha| - g_2(|\alpha|)\bigr\} \end{aligned}$$

   by searching in T_2 with the two sorted lists $\{p_i + |\alpha| + g_1(|\alpha|) \mid i = 1, 2, \ldots, s\}$ and $\{p_i - |\alpha| - g_2(|\alpha|) \mid i = 1, 2, \ldots, s\}$ using Lemma 7.3.

   (b) For each element p in T_1 we call $\mathsf{Report}(q_r(p), p + |\alpha| + g_2(|\alpha|))$ and $\mathsf{Report}(q_\ell(p), p - |\alpha| - g_1(|\alpha|))$ where Report is the following procedure.

   ```
   Report(from, to)
       $q = from$
       while $q \leq to$ do
           report pair $(p, q, |\alpha|)$ if $p < q$, and $(q, p, |\alpha|)$ otherwise
           $q = next(q)$
   ```

   (c) Build the leaf-list tree T at node v by merging T_1 and T_2 applying Lemma 7.2.

into blocks of elements with equal left-characters. To filter out all right-maximal pairs that are not left-maximal we must avoid to report p in $LL(w_1)$ against any element q in $LL(w_2)$ in a block of elements with left-character $S[p-1]$. This gives the overall idea of the extended algorithm; we extend the reporting step in Algorithm 7.1 such that whenever we are about to report p in $LL(w_1)$ against q in $LL(w_2)$ where $S[p-1] = S[q-1]$, we skip all elements in the current block containing q and continue reporting p against the first element q' in the following block, which by the definition of blocks satisfies that $S[p-1] \neq S[q'-1]$.

To implement this extended reporting step efficiently we must be able to skip all elements in a block without inspecting each of them. We achieve this by constructing an additional AVL tree, the *block-start tree*, that keeps track of the blocks in the leaf-list. At each node v during the traversal of $T_B(S)$ we thus construct two AVL trees; the leaf-list tree T that stores the elements in $LL(v)$, and the block-start tree B that keeps track of the blocks in the sorted leaf-list by storing all the elements in $LL(v)$ that start a block. We keep links from the block-start tree to the leaf-list tree such that we in constant time can

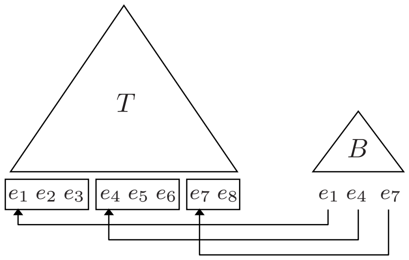


Figure 7.3: The data structure constructed at each node v in $T_B(S)$. The leaf-list tree T stores all elements in $LL(v)$. The block-start tree B stores all elements in $LL(v)$ that start a block in the sorted leaf-list. We keep links from elements in the block-start tree to corresponding elements in the leaf-list tree.

go from an element in the block-start tree to the corresponding element in the leaf-list tree. Figure 7.3 illustrates the leaf-list tree, the block-start tree and the links between them. Before we present the extended algorithm and explain how to use the block-start tree to efficiently skip all elements in a block. We first describe how to construct the leaf-list tree T and the block-start tree B at node v from the leaf-list trees, T_1 and T_2, and the block-start trees, B_1 and B_2, at its two children w_1 and w_2.

Since the leaf-list $LL(v)$ is the union of the disjoint leaf-lists $LL(w_1)$ and $LL(w_2)$ stored in T_1 and T_2 respectively, we can construct the leaf-list tree T by merging T_1 and T_2 using Lemma 7.2. It is more involved to construct the block-start tree B. The reason is that an element p_i that starts a block in $LL(w_1)$ or an element q_j that starts a block in $LL(w_2)$ does not necessarily start a block in $LL(v)$ and vice versa, so we cannot construct B by merging B_1 and B_2. Let $\{e_1, e_2, \ldots, e_{s+t}\}$ be the elements in $LL(v)$ in sorted order. By definition the block-start tree B contains all elements e_k in $LL(v)$ where $S[e_{k-1}-1] \neq S[e_k-1]$. We construct B by modifying B_2. We choose to modify B_2, and not B_1, because $|LL(w_1)| \leq |LL(w_2)|$, which by the "smaller-half trick" allows us to consider all elements in $LL(w_1)$ without spending too much time in total. To modify B_2 to become B we must identify all the elements that are in B but not in B_2 and vice versa.

Lemma 7.5 *If e_k is in B but not in B_2 then $e_k \in LL(w_1)$ or $e_{k-1} \in LL(w_1)$.*

Proof. Assume that e_k is in B and that both e_k and e_{k-1} are in $LL(w_2)$. In $LL(w_2)$ the elements e_k and e_{k-1} are neighboring elements. Let these neighboring elements in $LL(w_2)$ be denoted q_j and q_{j-1}. Since e_k is in B and therefore starts a block in $LL(v)$ then $S[q_j - 1] = S[e_k - 1] \neq S[e_{k-1} - 1] = S[q_{j-1} - 1]$. This shows that $q_j = e_k$ is in B_2 and the lemma follows. □

In the following *NEW* denotes the set of elements e_k in B where either e_k or e_{k-1} is in $LL(w_1)$. It follows from Lemma 7.5 that *NEW* contains at least all elements in B that are not in B_2. We can construct *NEW* in sorted order while merging T_1 and T_2 as follows. When an element e_k from T_1, i.e. from

$LL(w_1)$, is placed in T, i.e. in $LL(v)$, we include it in the set NEW if it starts a block in $LL(v)$. Similarly the next element e_{k+1} in $LL(v)$ is included in NEW if it starts a block in $LL(v)$.

Constructing the set NEW is the first step in modifying B_2 to become B. The next step is to identify the elements that should be removed from B_2, that is, to identify the elements that are in B_2 but not in B.

Lemma 7.6 *An element q_j in B_2 is not in B if and only if the largest element e_k in NEW smaller than q_j in B_2 has the same left-character as q_j.*

Proof. If q_j is in B_2 but does not start a block in $LL(v)$, then it must be in a block started by some element e_k with the same left-character as q_j. This block cannot contain q_{j-1} because q_j being in B_2 implies that $S[q_j - 1] \neq S[q_{j-1} - 1]$. We thus have the ordering $q_{j-1} < e_k < q_j$. This implies that e_k is the largest element in NEW smaller than q_j. If e_k is the largest element in NEW smaller than q_j, then no block starts in $LL(v)$ between e_k and q_j, i.e. all elements e in $LL(v)$ where $e_k < e < q_j$ satisfy that $S[e-1] = S[e_k-1]$, so if $S[e_k-1] = S[q_j-1]$ then q_j does not start a block in $LL(v)$. □

To identify the elements that should be removed from B_2, we search B_2 with the sorted list NEW using Lemma 7.3 to find all pairs of elements (e_k, q_j), where e_k is the largest element in NEW smaller than q_j in B_2. If the left-characters of e_k and q_j in such a pair are equal, i.e. $S[e_k - 1] = S[q_j - 1]$, then by Lemma 7.6 the element q_j is not in B and must therefore be removed from B_2. It follows from the proof of Lemma 7.6 that if this is the case then $q_{j-1} < e_k < q_j$, so we can, without destroying the order among the nodes in B_2, remove q_j from B_2 and insert e_k instead, simply by replacing the element q_j with the element e_k at the node storing q_j in B_2.

We can now summarize the three steps it takes to modify B_2 to become B. In Step 1 we construct the sorted set NEW that contains all elements in B that are not in B_2. This is done while merging T_1 and T_2 using Lemma 7.2. In Step 2 we remove the elements from B_2 that are not in B. The elements in B_2 being removed and the elements from NEW replacing them are identified using Lemmas 7.3 and 7.6. In Step 3 we merge the remaining elements in NEW into the modified B_2 using Lemma 7.2. Adding links from the new elements in B to the corresponding elements in T can be done while replacing and merging in Steps 2 and 3. Since $|NEW| \leq 2\,|T_1|$ and $|B_2| \leq |T_2|$, the time it takes to construct B is dominated by the the time it takes to merge a sorted list of size $2\,|T_1|$ into an AVL tree of size $|T_2|$. By Lemma 7.2 this is within a constant factor of the time it takes to merge T_1 and T_2, so the time is takes to construct B is dominated by the time it takes to construct the leaf-list tree T.

Now that we know how to construct the leaf-list tree T and block-start tree B at node v from the leaf-list trees, T_1 and T_2, and block-start trees, B_1 and B_2, at its two children w_1 and w_2, we can proceed with the implementation of the extended reporting step. The details are shown in Algorithm 7.2. This algorithm is similar to Algorithm 7.1 except that we at every node v in $T_B(S)$ construct two AVL trees; the leaf-list tree T that stores the elements in $LL(v)$, and the block-start tree B that keeps track of the blocks in $LL(v)$ by storing

Algorithm 7.2 Find all maximal pairs in string S with bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$ and create at each leaf two AVL trees of size one, the leaf-list and the block-start tree, storing the index at the leaf.

2. *Reporting and merging:* When the leaf-list trees T_1 and T_2, where $|T_1| \leq |T_2|$, and the block-start trees B_1 and B_2 at the two children w_1 and w_2 of node v with path-label α are available, we do the following:

   (a) Let $\{p_1, p_2, \ldots, p_s\}$ be the elements in T_1 in sorted order. For each element p in T_1 we find

   $$\begin{aligned} q_r(p) &= \min\{x \in T_2 \mid x \geq p + |\alpha| + g_1(|\alpha|)\} \\ q_\ell(p) &= \min\{x \in T_2 \mid x \geq p - |\alpha| - g_2(|\alpha|)\} \\ b_r(p) &= \min\{x \in B_2 \mid x \geq p + |\alpha| + g_1(|\alpha|)\} \\ b_\ell(p) &= \min\{x \in B_2 \mid x \geq p - |\alpha| - g_2(|\alpha|)\} \end{aligned}$$

   by searching in T_2 and B_2 with the sorted lists $\{p_i + |\alpha| + g_1(|\alpha|) \mid i = 1, 2, \ldots, s\}$ and $\{p_i - |\alpha| - g_2(|\alpha|) \mid i = 1, 2, \ldots, s\}$ using Lemma 7.3.

   (b) For each element p in T_1 we call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ and $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$, where ReportMax is the following procedure.

```
ReportMax($from\_in\_T$, $from\_in\_B$, $to$)
    $q = from\_in\_T$
    $b = from\_in\_B$
    while $q \leq to$ do
        if $S[q-1] \neq S[p-1]$ then
            report pair $(p, q, |\alpha|)$ if $p < q$, and $(q, p, |\alpha|)$ otherwise
            $q = next(q)$
        else
            while $b \leq q$ do $b = next(b)$
            $q = b$
```

   (c) Build the leaf-list tree T at node v by merging T_1 and T_2 using Lemma 7.2. Build the block-start tree B at node v by modifying B_2 as described in the text.

the subset of elements that start a block. If v is a leaf, we construct T and B directly. If v is an internal node, we construct T by merging the leaf-list trees T_1 and T_2 at its two children w_1 and w_2, and we construct B by modifying the block-start tree B_2 as explained above.

Before constructing T and B we report all maximal pairs from node v with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, by reporting every p in $LL(w_1)$ against every q in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$ where $S[p-1] \neq S[q-1]$. This is done in two steps. In Step 2a we find for every p in $LL(w_1)$ the minimum elements $q_\ell(p)$ and $q_r(p)$, as well as the minimum elements $b_\ell(p)$ and $b_r(p)$ that start a block, in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$ respectively. This is done by searching in T_2 and B_2 using Lemma 7.3. In Step 2b we report pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ for every p in $LL(w_1)$ and increasing q's in $LL(w_2)$ starting with $q_r(p)$ and $q_\ell(p)$ respectively, until the gap violates the upper or lower bound. Whenever we are about to report p against q where $S[p-1] = S[q-1]$, we instead use the block-start tree B_2 to skip all elements in the block containing q and continue with reporting p against the first element in the following block.

To argue that Algorithm 7.2 finds all the maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ it is sufficient to show that we for every p in $LL(w_1)$ report all maximal pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. The rest follows because we at every node in $T_B(S)$ consider every p in $LL(w_1)$. Consider the call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ in Step 2b. The implementation of ReportMax implies that unless we skip elements by increasing b, we consider every q in $LL(w_2) \cap R(p, |\alpha|)$ exactly as in Algorithm 7.1. The test $S[q-1] \neq S[p-1]$ ensures that we only report maximal pairs. Whenever $S[q-1] = S[p-1]$ we increase b until $b = \min\{x \in B_2 \mid x > q\}$, which by construction of B_2 and $b_r(p)$ is the element that starts the block following the block containing q. Hence, all the elements q', where $q < q' < b$, we skip by setting q to b thus satisfy that $S[p-1] = S[q-1] = S[q'-1]$. We conclude that $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ reports p against exactly those q in $LL(w_2) \cap R(p, |\alpha|)$ where $S[p-1] \neq S[q-1]$, i.e. it reports all maximal pairs $(p, q, |\alpha|)$ at node v with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Similarly, the call $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$ reports all maximal pairs $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

We now consider the running time of Algorithm 7.2. We first argue that the call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ takes constant time plus time proportional to the number of reported pairs $(p, q, |\alpha|)$. To do this all we have to show is that the time used to skip blocks, i.e. the number of times we increase b, is proportional to the number of reported pairs. By construction $b_r(p) \geq q_r(p)$, so the number of times we increase b is bounded by the number of blocks in $LL(w_2) \cap R(p, |\alpha|)$. Since neighboring blocks contain elements with different left-characters, we report p against an element from at least every second block in $LL(w_2) \cap R(p, |\alpha|)$. The number of times we increase b is thus proportional to the number of reported pairs. Similarly the call $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$ also takes constant time plus time proportional to the number of reported pairs $(q, p, |\alpha|)$. We thus have that Step 2b takes time proportional to $|T_1|$ plus the number of reported pairs. Everything else we do at node v, i.e. searching in T_2 and B_2 and constructing the leaf-list tree T and block-start tree B, takes time $O(|T_1| \log(|T_2|/|T_1|))$. Summing this over all nodes gives by Lemma 7.4 that the total running time of the algorithm is $O(n \log n + z)$, where z is the number of reported pairs. Since constructing and keeping $T_B(S)$ requires space $O(n)$, and since no element at any time is in more than one leaf-list tree, and maybe one block-start tree, Algorithm 7.2 requires space $O(n)$.

Theorem 7.2 *Algorithm 7.2 finds all maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ in time $O(n \log n + z)$ and space $O(n)$, where z is the number of reported pairs.*

As a closing remark we can observe that Algorithm 7.2 never uses the block-start tree B_1 at the small child w_1. This observation can be used to ensure that only one block-start tree exists during the execution of the algorithm. If we implement the traversal of $T_B(S)$ as a depth-first traversal in which we at each node v first recursively traverse the subtree rooted at the small child w_1, then we do not need to store the block-start tree returned by this recursive traversal while recursively traversing the subtree rooted at the big child w_2. This implies that only one block-start tree exists at all times during the recursive traversal of $T_B(S)$. The drawback is that we at each node v need to know in advance which child is the small child, but this knowledge can be obtained in linear time by annotating each node of $T_B(S)$ with the size of the subtree it roots.

7.4 Pairs with Lower Bounded Gap

If we relax the constraint on the gap and only want to find all maximal pairs in S with gap at least $g(|\alpha|)$, where g is a function that can be computed in constant time, then a straightforward solution is to use Algorithm 7.2 with $g_1(|\alpha|) = g(|\alpha|)$ and $g_2(|\alpha|) = n$. This obviously finds all maximal pairs with gap at least $g(|\alpha|)$ in time $O(n \log n + z)$. However, the missing upper bound on the gap makes it possible to reduce the running time to $O(n+z)$ since reporting from each node during the traversal of the binary suffix tree is simplified.

The reporting of pairs from node v with children w_1 and w_2 is simplified, because the lack of an upper bound on the gap implies that we do not have to search $LL(w_2)$ for the first element to report against the current element in $LL(w_1)$. Instead we can start by reporting the current element in $LL(w_1)$ against the biggest (and smallest) element in $LL(w_2)$, and then continue reporting it against decreasing (and increasing) elements from $LL(w_2)$ until the gap becomes smaller than $g(|\alpha|)$. Unfortunately this simplification alone does not reduce the asymptotic running time because inspecting every element in $LL(w_1)$ and keeping track of the leaf-lists in AVL trees alone requires time $\Theta(n \log n)$. To reduce the running time we must thus avoid to inspect every element in $LL(w_1)$ and find another way to store the leaf-lists. We achieve this by using the priority-queue like data structures presented in the next section to store the leaf-lists during the traversal of the binary suffix tree.

7.4.1 Data Structures

A heap-ordered tree is a tree in which each node stores an element and has a key. Every node other than the root satisfies that its key is greater than or equal to the key at its parent. Heap-ordered trees have been widely studied and are the basic structure of many priority queues [204, 56, 195, 62]. In this section we utilize heap-ordered trees to construct two data structures, *the heap-tree* and *the colored heap-tree*, that are useful in our application of finding pairs with lower bounded gap but might also have applications elsewhere.

A heap-tree stores a collection of elements with comparable keys and supports the following operations.

Init(e, k): Return a heap-tree of size one that stores element e with key k.

Find(H, x): Return all elements e stored in the heap-tree H with key $k \leq x$.

Min(H): Return the element e stored in H with minimum key.

Meld(H, H'): Return a heap-tree that stores all elements in H and H' with unchanged keys and colors.

A colored heap-tree stores a collection of colored elements with comparable keys. We use $\mathit{color}(e)$ to denote the color of element e. A colored heap-tree supports the same operations as a heap-tree except that it allows us to find all elements not having a particular color. The operations are as follows.

ColorInit(e, k): Return a colored heap-tree of size one that stores element e with key k.

ColorFind(H, x, c): Return all elements e stored in the colored heap-tree H with key $k \leq x$ and $\mathit{color}(e) \neq c$.

ColorMin(H): Return the element e stored in H with minimum key.

ColorSec(H): Return the element e stored in H with minimum key such that $\mathit{color}(e) \neq \mathit{color}(\mathsf{ColorMin}(H))$.

ColorMeld(H, H'): Return a colored heap-tree that stores all elements in H and H' with unchanged keys.

In the following we will describe how to implement heap-trees and colored heap-trees using heap-ordered trees such that Init, Min, ColorInit, ColorMin and ColorSec take constant time, Find and ColorFind take time proportional to the number of returned elements, and Meld and ColorMeld take amortized constant time. This means that we can meld n (colored) heap-trees of size one into a single (colored) heap-tree of size n by an arbitrary sequence of $n - 1$ meld operations in time $O(n)$ in the worst case.

Heap-Trees

We implement heap-trees as binary heap-ordered trees as illustrated in Figure 7.4. At every node in the heap-ordered tree we store an element from the collection of elements we want to store. The key of a node is the key of the element it stores. We use $v.\mathit{elm}$ to refer to the element stored at node v, $v.\mathit{key}$ to refer to the key of node v, and $v.\mathit{right}$ and $v.\mathit{left}$ to refer to the two children

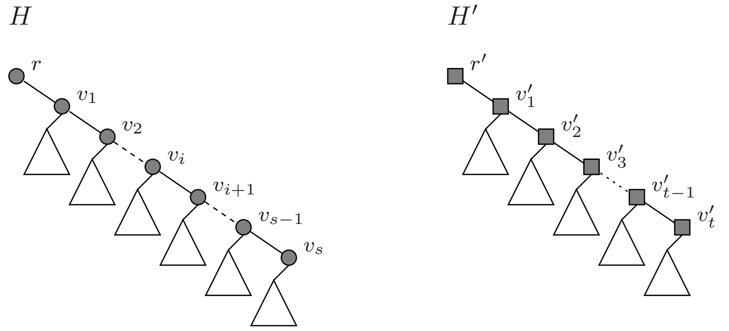


Figure 7.4: Heap-trees are binary heap-ordered trees. The figure shows two heap-trees H and H'. The nodes on the backbone of the heap-trees are shaded.

of node v. Besides the heap-order we maintain the invariant that the root of the heap-ordered tree has no left-child.

We define the *backbone* of a heap-tree as the path in the heap-ordered tree that starts at the root and continues via nodes reachable from the root via a sequence of right-children. We define the length of the backbone as the number of edges on the path it describes. Consider the heap-trees H and H' in Figure 7.4; the backbone of H is the path $r, v_1, \ldots, v_s$ of length s and the backbone of H' is the path $r', v'_1, \ldots, v'_t$ of length t. We say that the node on the backbone farthest from the root is at the bottom of the backbone. We keep track of the nodes on the backbone of a heap-tree using a stack, *the backbone-stack*, in which the root is at the bottom and the node farthest from the root is at the top. The backbone-stack makes it easy to access the nodes on the backbone from the bottom and up towards the root.

We now turn to the implementation of Init, Min, Find and Meld. The implementation of Init(e, k) is straightforward. We construct a single node v where $v.elm = e$, $v.key = k$ and $v.right = v.left = null$ and a backbone-stack of size one that contains node v. The implementation of Min(H) is also straightforward. The heap-order implies that root r of H stores the element with minimum key, i.e. Min$(H) = r.elm$.

The implementation of Find(H, x) is based on a recursive traversal of H starting at the root. At each node v we compare $v.key$ to x. If $v.key \leq x$, we report $v.elm$ and continue recursively with the two children of v. If $v.key > x$, then by the heap-order all keys at nodes in the subtree rooted at v are greater than x, so we return from v without reporting. Clearly this traversal reports all elements stored at nodes v with $v.key \leq x$, i.e. all elements stored with key $k \leq x$. Since each node has at most two children, we make for each reported element at most two additional comparisons against x corresponding to the at most two recursive calls from which we return without reporting. The running time of Find(H, x) is thus proportional to the number of reported elements.

The implementation of Meld(H, H') is done in two steps. Figure 7.5 illustrates the melding of the heap-trees H and H' from Figure 7.4. We assume that $r.key \leq r'.key$. In Step 1 we merge the backbones of H and H' together such

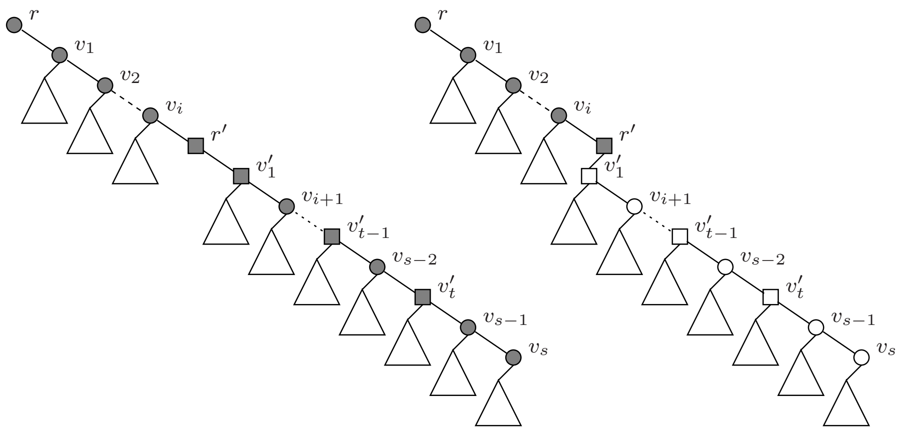


Figure 7.5: The two steps of melding the heap-trees H and H' shown in Figure 7.4. The heap-tree to the left is the result of merging the backbones. The heap-tree to the right is the result of shortening the backbone by moving the right-child of r' in the merged backbone to the left-child. The nodes on the backbones are marked.

that the heap-order is satisfied in the resulting tree. The merged backbone is constructed from the bottom and up towards the root by popping nodes from the backbone-stacks of H and H'. Step 1 results in a heap-tree with a backbone of length $s+t+1$. Since $r.key \leq r'.key$, a prefix of the merged backbone consists of nodes $r, v_1, v_2, \ldots, v_i$ solely from the backbone of H. In Step 2 we shorten the merged backbone. Since the root r' of H' has no left-child, the node r' on the merged backbone has no left-child either, so by moving the right-child of r' to this empty spot, making it the left-child of r', we shorten the length of the merged backbone to $i+1$.

The two steps of $\mathsf{Meld}(H, H')$ clearly construct a heap-ordered tree that stores all elements in H and H' with unchanged keys. Since $r.key \leq r'.key$, the root of the constructed heap-ordered tree is the root of H and therefore has no left-child. The constructed heap-ordered tree is thus a heap-tree as wanted. The backbone of the new heap-tree is the path $r, v_1, \ldots, v_i, r'$. We observe that the backbone-stack of H after Step 1 contains exactly the nodes $r, v_1, \ldots v_i$. We can thus construct the backbone-stack of the new heap-tree by pushing r' onto what remains of the backbone-stack of H after Step 1.

Now consider the running time of $\mathsf{Meld}(H, H')$. Step 1 takes time proportional to the total number of nodes popped from the two backbone-stacks. Since $i+1$ nodes remains on the backbone-stack of H, Step 1 takes time $(s+1)+(t+1)-(i+1) = s+t-i+1$. Step 2 and construction of the new backbone-stack takes constant time, so, except for a constant factor, melding two heap-trees with backbones of length s and t takes time $T(s,t) = s+t-i+1$. In our application of finding pairs we are more interested in bounding the total time required to do a sequence of melds rather than bounding the time of each individual meld. We therefore turn to amortized analysis [179].

On a forest F of heap-trees we define the potential function $\Phi(F)$ to be the sum of the lengths of the backbones of the heap-trees in the forest. Melding two heap-trees with backbones of length s and t, as illustrated in Figure 7.5, changes the potential of the forest with $\Delta\Phi = i+1-(s+t)$. The amortized running time of melding the two heap-trees is thus $T(s,t)+\Delta\Phi = (s+t-i+1)+(i-s-t+1) = 2$, so starting with n heap-trees of size one, i.e. a forest F_0 with potential $\Phi(F_0) = 0$, and doing a sequence of $n-1$ meld operations until the forest F_{n-1} consists of a single heap-tree, takes time $O(n)$ in the worst case.

Colored Heap-Trees

We implement colored heap-trees as colored heap-ordered trees in much tqhe same way as we implemented heap-trees as uncolored heap-ordered trees. The implementation only differs in two ways. First, a node in the colored heap-ordered tree stores a set of elements instead of just a single element. Secondly, a node, including the root, can have several left-children. The elements stored at a node, and the references to the left-children of a node, are kept in uncolored heap-trees. More precisely, a node v in the colored heap-ordered tree has the following attributes.

$v.elms$: A heap-tree that stores the elements at node v. $\mathsf{Find}(v.elms, x)$ returns all elements stored at node v with key less than or equal to x. All elements stored at node v have identical colors. We say that this color is the color of node v and denote it by $color(v)$.

$v.key$: The key of node v. We set the key of a node to be the minimum key of an element stored at the node, i.e. the key of node v is the key of the element stored at the root of the heap-tree $v.elms$.

$v.right$: A reference to the right-child of node v.

$v.lefts$: A heap-tree that stores the references to the left-children of node v. A reference is stored with a key equal to the key of the referenced left-child, so $\mathsf{Find}(v.lefts, x)$ returns the references to all left-children of node v with key less than or equal to x.

As for the heap-tree we define the backbone of a colored heap-tree as the path that starts at the root and continues via nodes reachable from the root via a sequence of right-children. We use a stack, the backbone-stack, to keep track of the nodes on the backbone. In addition to the heap-order, saying that the key of every node other than the root is greater than or equal to the key of its parent, we maintain the following three invariants about the color of the nodes and the relation between the elements stored at a node and its left-children.

I_1: Every node v other than the root r has a color different from its parent.

I_2: Every node v satisfies that $|\mathsf{Find}(v.elms, x)| \geq |\mathsf{Find}(v.lefts, x)|$ for any x.

I_3: The root r satisfies that $|\mathsf{Find}(r.\mathit{elms}, x)| \geq |\mathsf{Find}(r.\mathit{lefts}, x)| + 1$ for any $x \geq \mathsf{Min}(r.\mathit{elms})$.

We now turn our attention towards the implementation of the operations on colored heap-trees. $\mathsf{ColorInit}(e, k)$ is straightforward. We simply construct a single node v where $v.\mathit{key} = k$, $v.\mathit{elms} = \mathsf{Init}(e, k)$ and $v.\mathit{right} = v.\mathit{lefts} = \mathit{null}$ and a backbone-stack that contains node v. $\mathsf{ColorMin}(H)$ is also straightforward. The heap-order implies that the element with minimum key is stored in the heap-tree $r.\mathit{elms}$ at the root r of H, so $\mathsf{ColorMin}(H) = \mathsf{Min}(r.\mathit{elms})$. The heap-order and I_1 imply that $\mathsf{ColorSec}(H)$ is the element stored with minimum key at a child of r. The element stored with minimum key at the right-child is $\mathsf{Min}(r.\mathit{right})$ and the element stored with minimum key at a left-child must by the heap-order of $r.\mathit{lefts}$ be the element stored with minimum key at the left-child referenced by the root of $r.\mathit{lefts}$, i.e. $\mathsf{Min}(\mathsf{Root}(r.\mathit{lefts}).\mathit{elm})$. Both $\mathsf{ColorMin}(H)$ and $\mathsf{ColorSec}(H)$ can thus be found in constant time.

We implement $\mathsf{ColorFind}(H, x, c)$ as a recursive traversal of H starting at the root. More precisely, we implement $\mathsf{ColorFind}(H, x, c)$ as $\mathsf{ReportFrom}(r)$ where r is the root of H and $\mathsf{ReportFrom}$ is the following recursive procedure.

```
ReportFrom($v$)
    if $key(v) \leq x$ then
        if $color(v) \neq c$ then
            $E = \mathsf{Find}(v.elms, x)$
            for $e$ in $E$ do
                report $e$
        ReportFrom($v.right$)
        $W = \mathsf{Find}(v.lefts, x)$
        for $w$ in $W$ do
            ReportFrom($w$)
```

The correctness of this implementation is established as follows. The heap-order ensures that all nodes v with $v.\mathit{key} \leq x$ are visited during the traversal. The definition of $v.\mathit{key}$ implies that any element e with key $k \leq x$ is stored at a node v with $v.\mathit{key} \leq x$, i.e. among the elements returned by $\mathsf{Find}(v.\mathit{elms}, x)$ for some v visited during the traversal. Together with the test $\mathit{color}(v) \neq c$ this implies that all elements e with key $k \leq x$ and color different from c are reported by $\mathsf{ColorFind}(H, x, c)$.

Now consider the running time of $\mathsf{ColorFind}(H, x, c)$. Since $\mathsf{Find}(v.\mathit{elms}, x)$ and $\mathsf{Find}(v.\mathit{lefts}, x)$ both take time proportional to the number of returned elements, it follows that the running time is dominated by the number of recursive calls plus the number of reported elements. To argue that the running time of $\mathsf{ColorFind}(H, x, c)$ is proportional to the number of reported elements we therefore argue that the number of reported elements dominates the number of recursive calls. We only make recursive calls from a node v if $v.\mathit{key} \leq x$. Let v be such a node and consider two cases.

If $\mathit{color}(v) \neq c$ then we report at least one element, namely the element with key $v.\mathit{key}$, and by the invariants I_2 and I_3 we report at least as many elements as the number of left-children we call when reporting from v. Hence, except for a constant term that we can charge for visiting node v, the number of reported elements at v accounts for the call to v and all the recursive calls from v.

If $\mathit{color}(v) = c$ then we do not report any elements at v, but the invariant I_1 ensures that we have reported elements at its parent (unless v is the root) and that we will be reporting elements at all left-children we call from v. The call to v is thus already accounted for by the elements reported at its parent, and except for a constant term that we can charge for visiting node v, all calls from v will be accounted for by elements reported at the children of v. We conclude that the number of reported elements dominates the number of recursive calls, so $\mathsf{ColorFind}(H, x, c)$ takes time proportional to the number of reported elements.

We implement $\mathsf{ColorMeld}(H, H')$ similar to $\mathsf{Meld}(H, H')$ except that we must ensure that the constructed colored heap-tree obeys the three invariants. Let H and H' be colored heap-trees with roots r and r' named such that $r.\mathit{key} \leq r'.\mathit{key}$. We implement $\mathsf{ColorMeld}(H, H')$ as the following three steps.

1. *Merge*. We merge the backbones of H and H' together such that the resulting heap-ordered tree stores all elements in H and H' with unchanged keys. The merging is done by popping nodes from the backbone-stacks of H and H' until the backbone-stack of H' is empty

2. *Solve conflicts*. A node w on the merged backbone with the same color as its parent v is a violation of invariant I_1. We solve conflicts between neighboring nodes v and w of equal color by melding the elements and left-children of the two nodes and removing node w. We say that parent v swallows the child w.

   $v.\mathit{elms} = \mathsf{Meld}(v.\mathit{elms}, w.\mathit{elms})$
   $v.\mathit{lefts} = \mathsf{Meld}(v.\mathit{lefts}, w.\mathit{lefts})$
   $v.\mathit{right} = w.\mathit{right}$

3. *Shorten backbone*. Let v be the node on the merged backbone corresponding to r' or the node that swallowed r' in Step 2. We shorten the backbone by moving the right-child of v to the set of left-children of v.

   $v.\mathit{lefts} = \mathsf{Meld}(v.\mathit{lefts}, \mathsf{Init}(v.\mathit{right}, v.\mathit{right}.\mathit{key}))$
   $v.\mathit{right} = \mathit{null}$

The main difference from $\mathsf{Meld}(H, H')$ is Step 2 where the invariant I_1 is restored along the merged backbone. To establish the correctness of the implementation of $\mathsf{ColorMeld}(H, H')$ we consider each of the three steps in more details.

In Step 1 we merge the backbones of H and H' together such that the resulting tree is a heap-ordered tree that stores all elements in H and H' with unchanged keys. Since the merging does not change the left-children or the elements of any node and since H and H' both obey I_2 and I_3, the constructed heap-ordered tree also obeys I_2 and I_3. The merged backbone can however contain neighboring nodes of equal color. These conflicts are a violation of I_1.

In Step 2 we restore I_1. We solve all conflicts on the merged backbone between neighboring nodes v and w of equal color by letting the parent v swallow the child w as illustrated in Figure 7.6. We observe that since H and H' both obey I_1 a conflict must involve a node from both of them. This implies that a conflict can only occur in the part of the merged backbone made of nodes popped off the backbone-stacks in Step 1. We also observe that solving a conflict does not induce a new conflict. Combined with the previous observation this implies that the number of conflicts is bounded by the number of nodes popped off the backbone-stacks in Step 1. Finally, we observe that solving a conflict does not induce violations of I_2 and I_3, so after solving all conflicts on the merged backbone we have a colored heap-tree that stores all elements in H and H' with unchanged keys.

In Step 3 we shorten the merged backbone. It is done by moving the right-child of r' to its left-children, or in case r' has been swallowed by a node v in Step 2, by moving the right-child of v to its left-children. The subtree rooted by the right-child moved follows along, and thus becomes a subtree rooted by the new left-child of r' (or v). To argue that shortening the backbone does not induce violations of I_2 and I_3 we start by making two observations. First, we observe that moving the right-child of a node that obeys I_3 to its set of left-children results in a node that obeys I_2. Secondly, we observe that if a node that obeys I_2 (or I_3) swallows a node that obeys I_2 it results in a node that still obeys I_2 (or I_3).

Since r' is the root of H', it obeys I_3 before Step 2. We consider two cases. First, if r' is not swallowed in Step 2, the first observation immediately implies that it obeys I_2 after Step 3. Secondly, if r' is swallowed by a node v in Step 2, we might as well think of Steps 2 and 3 as occurring in opposite order as this does not affect the resulting tree. Hence, first we move the right-child of r' to its set of left-children, which by the first observation results in a node that obeys I_2, then we let node v swallow this node, which by the second observation does not affect the invariants obeyed by v.

We conclude that the implementation of $\mathsf{ColorMeld}(H, H')$ constructs a colored heap-tree that obeys all three invariants and stores all elements in H and H' with unchanged keys and colors. The backbone-stack of the colored heap-tree constructed by $\mathsf{ColorMeld}(H, H')$ is what remains on the backbone-stack of H after popping nodes in Step 1 with the node r' pushed onto it, unless the node r' is swallowed in Step 2.

Now consider the time it takes to meld n colored heap-trees of size one together by a sequence of $n - 1$ melds. If we ignore the time it takes to meld the heap-trees storing elements and references to left-children when solving conflicts in Step 2 and shortening the backbone in Step 3, then we can bound the time it takes to do the sequence of melds by $O(n)$ exactly as we did in the previous section. Melding n colored heap-trees of size one involves melding at most n heap-trees of size one storing elements, and at most n heap-trees of size one storing references to left-children. Since melding n heap-trees of size one takes time $O(n)$, we have that melding the heap-trees storing elements and references to left-children also takes time $O(n)$, so melding n colored heap-trees of size one takes time $O(n)$ in the worst case.

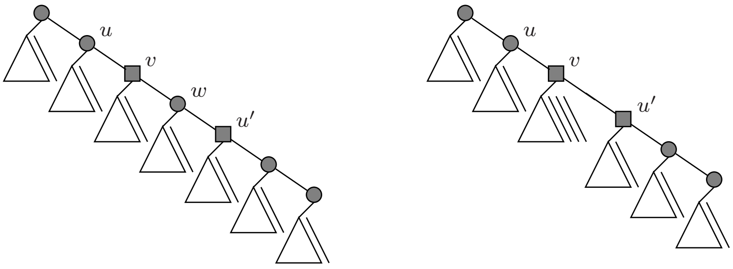


Figure 7.6: This figure illustrates how a conflict on the merged backbone is solved. If $color(v) = color(w)$ then I_1 is violated. The invariant is restored by letting node v swallow node w, i.e. melding the elements and left-children at the two nodes and removing node w. Since $color(u) \neq color(w) = color(v)$ and $color(u') \neq color(v)$, solving a conflict does not induce another conflict.

7.4.2 Algorithms

In the following we present two algorithms to find pairs with lower bounded gap. First we describe a simple algorithm to find all right-maximal pairs with lower bounded gap using heap-trees, then we extend it to find all maximal pairs with lower bounded gap using colored heap-trees. Both algorithms run in time $O(n + z)$ where z is the number of reported pairs.

Right-Maximal Pairs with Lower Bounded Gap

We find all right-maximal pairs in S with gap at least $g(|\alpha|)$, by for each node v in the binary suffix tree $T_B(S)$ considering the leaf-lists at its two children w_1 and w_2. The pair $(p, q, |\alpha|)$, for $p \in LL(w_1)$ and $q \in LL(w_2)$, is right-maximal and has gap at least $g(|\alpha|)$ if and only if $q \geq p + |\alpha| + g(|\alpha|)$. If we let p_{min} denote the minimum element in $LL(w_1)$ this implies that every q in

$$Q = \{q \in LL(w_2) \mid q \geq p_{min} + |\alpha| + g(|\alpha|)\}$$

forms a right-maximal pair $(p, q, |\alpha)$ with gap at least $g(|\alpha|)$ with every p in

$$P_q = \{p \in LL(w_1) \mid p \leq q - g(|\alpha|) - |\alpha|\} .$$

By construction P_q contains p_{min} and we have that $(p, q, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $q \in Q$ and $p \in P_q$. We can construct Q and P_q using heap-trees. Let H_i and $\bar{H}_i$ be heap-trees that store the elements in $LL(w_i)$ ordered by "$\leq$" and "$\geq$" respectively. By definition of the operations Min and Find we have that $p_{min} = \mathsf{Min}(H_1)$, $Q = \mathsf{Find}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|)$ and $P_q = \mathsf{Find}(H_1, q - g(|\alpha|) - |\alpha|)$.

This leads to the formulation of Algorithm 7.3 in which we at every node v in $T_B(S)$ construct two heap-trees, H and $\bar{H}$, that store the elements in $LL(v)$ ordered by "$\leq$" and "$\geq$" respectively. If v is a leaf, we construct H and $\bar{H}$ directly by creating two heap-trees of size one each storing the index at the leaf. If v is an internal node, we construct H and $\bar{H}$ by melding the corresponding

Algorithm 7.3 Find all right-maximal pairs in S with lower bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$. Create at each leaf two heap-trees of size one, H ordered by "$\leq$" and $\bar{H}$ ordered by "$\geq$", that both store the index at the leaf.

2. *Reporting and melding:* When the heap-trees H_1 and $\bar{H}_1$ at the left-child of node v, and the heap-trees H_2 and $\bar{H}_2$ at the right-child of node v are available we report pairs of α, the path-label of v, and construct the heap-trees H and $\bar{H}$ as follows

```
 1  $Q = \mathsf{Find}(\bar{H}_2, \mathsf{Min}(H_1) + |\alpha| + g(|\alpha|))$
 2  for $q$ in $Q$ do
 3      $P_q = \mathsf{Find}(H_1, q - g(|\alpha|) - |\alpha|)$
 4      for $p$ in $P_q$ do
 5          report pair $(p, q, |\alpha|)$

 6  $P = \mathsf{Find}(\bar{H}_1, \mathsf{Min}(H_2) + |\alpha| + g(|\alpha|))$
 7  for $p$ in $P$ do
 8      $Q_p = \mathsf{Find}(H_2, p - g(|\alpha|) - |\alpha|)$
 9      for $q$ in $Q_p$ do
10          report pair $(q, p, |\alpha|)$

11  $H = \mathsf{Meld}(H_1, H_2)$
12  $\bar{H} = \mathsf{Meld}(\bar{H}_1, \bar{H}_2)$
```

heap-trees at the two children (lines 11–12). Before constructing H and $\bar{H}$ at node v, we report right-maximal pairs of its path-label (lines 1–10).

To argue that Algorithm 7.3 finds all right-maximal pairs in S with gap at least $g(|\alpha|)$ it is sufficient to show that we at each node v in $T_B(S)$ report all pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$, where $p \in LL(w_1)$ and $q \in LL(w_2)$, with gap at least $g(|\alpha|)$. The rest follows because we consider every node in $T_B(S)$. Let v be a node in $T_B(S)$ at which the heap-trees H_1, $\bar{H}_1$, H_2, and $\bar{H}_2$ at its two children are available. As explained above $(p, q, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $q \in Q$ and $p \in P_q$, which are exactly the pairs reported in lines 1–5. Symmetrically we can argue that $(q, p, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $p \in P$ and $q \in Q_p$, which are exactly the pairs reported in lines 6–10.

Now consider the running time of the algorithm. We first note that constructing two heap-trees of size one at each of the n leaves in $T_B(S)$ and melding them together according to the structure of $T_B(S)$ takes time $O(n)$ because each of the $n - 1$ meld operation takes amortized constant time. We then note that the reporting of pairs at each node, lines 1–10, takes time proportional to the number of reported pairs because the find operation takes time proportional to the number of returned elements and the set P_q (and Q_p) is non-empty for every element q in Q (and p in P). Finally we recall that constructing the binary suffix tree $T_B(S)$ takes time $O(n)$. Now consider the space needed by the algorithm. The binary suffix tree requires space $O(n)$. The heap-trees also requires space $O(n)$ because no element at any time is stored in more than one heap-tree. Finally, since no leaf-list contains more than n elements, storing the elements returned by the find operations during the reporting requires no more than space $O(n)$. In summary we formulate the following theorem.

Theorem 7.3 *Algorithm 7.3 finds all right-maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap at least $g(|\alpha|)$ in time $O(n + z)$ and space $O(n)$, where z is the number of reported pairs.*

Maximal Pairs with Lower Bounded Gap

Essential to the above algorithm is that we in time proportional to its size can construct the set Q that contains all elements q in $LL(w_2)$ that form a right-maximal pair $(p_{min}, q, |\alpha|)$ with gap at least $g(|\alpha|)$. Unfortunately the left-characters $S[q-1]$ and $S[p_{min}-1]$ can be equal, so Q can contain elements that do not form a maximal pair with any element in $LL(w_1)$. Since we aim for the reporting of pairs to take time proportional to the number of reported pairs, this implies that we cannot afford to consider every element in Q if we only want to report maximal pairs.

Fortunately we can efficiently construct the subset of $LL(w_2)$ that contains all the elements that form at least one maximal pair. An element q in $LL(w_2)$ forms a maximal pair if and only if there is an element p in $LL(w_1)$ such that $q \geq p + |\alpha| + g(|\alpha|)$ and $S[q-1] \neq S[p-1]$. We can construct this subset of $LL(w_2)$ using colored heap-trees. We define the color of an element to be its left-character, i.e. the color of p in $LL(w_1)$ and q in $LL(w_2)$ is $S[p-1]$ and $S[q-1]$ respectively. Let H_i and $\bar{H}_i$ be colored heap-trees that store the elements in $LL(w_i)$ ordered by "$\leq$" and "$\geq$" respectively. Using $p_{min} = \mathsf{ColorMin}(H_1)$ and $p_{sec} = \mathsf{ColorSec}(H_1)$ we can characterize the elements in $LL(w_2)$ that form at least one maximal pair with gap at least $g(|\alpha|)$ by considering two cases.

First, if $q \geq p_{sec} + |\alpha| + g(|\alpha|)$ then $(p_{min}, q, |\alpha|)$ and $(p_{sec}, q, |\alpha|)$ both have gap at least $g(|\alpha|)$ and since $S[p_{min}-1] \neq S[p_{sec}-1]$ at least one of them is maximal, so every $q \geq p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair with gap at least $g(|\alpha|)$. If # is a character not appearing anywhere in S, i.e. no element in $LL(w_2)$ has color #, this is the same as saying that every q in $Q' = \mathsf{ColorFind}(\bar{H}_2, p_{sec} + |\alpha| + g(|\alpha|), \#)$ forms a maximal pair with gap at least $g(|\alpha|)$. Secondly, if $q < p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair $(p, q, |\alpha|)$ with gap at least $g(|\alpha|)$ then $p_{min} \leq p < p_{sec}$. This implies that $S[p-1] = S[p_{min}-1]$, so $(p_{min}, q, |\alpha|)$ is also maximal and has gap at least $g(|\alpha|)$. We thus have that $q < p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair with gap at least $g(|\alpha|)$ if and only if $(p_{min}, q, |\alpha|)$ is maximal and has gap at least $g(|\alpha|)$, i.e. if and only if $S[q-1] \neq S[p_{min}-1]$ and $q \geq p_{min} + |\alpha| + g(|\alpha|)$. This implies that the set $Q'' = \mathsf{ColorFind}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|), S[p_{min}-1])$ contains every $q < p_{sec} + |\alpha| + g(|\alpha|)$ that forms a maximal pair with gap at least $g(|\alpha|)$.

By construction of Q' and Q'' the set $Q' \cup Q''$ contains all elements in $LL(w_2)$ that form a maximal pair with gap at least $g(|\alpha|)$. More precisely,

Algorithm 7.4 Find all maximal pairs in S with lower bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$. Create at each leaf two colored heap-trees of size one, H ordered by "$\leq$" and $\bar{H}$ ordered by "$\geq$", that both store the index at the leaf with color corresponding to its left-character.

2. *Reporting and melding:* When the colored heap-trees H_1 and $\bar{H}_1$ at the left-child of node v, and the colored heap-trees H_2 and $\bar{H}_2$ at the right-child of node v are available we report pairs of α, the path-label of v, and construct the colored heap-trees H and $\bar{H}$ as follows, where # is a character not appearing anywhere in S.

```
 1  $p_{min}, p_{sec} = \mathsf{ColorMin}(H_1), \mathsf{ColorSec}(H_1)$
 2  $Q' = \mathsf{ColorFind}(\bar{H}_2, p_{sec} + |\alpha| + g(|\alpha|), \#)$
 3  $Q'' = \mathsf{ColorFind}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|), S[p_{min} - 1])$
 4  for $q$ in $Q' \cup Q''$ do
 5      $P_q = \mathsf{ColorFind}(H_1, q - g(|\alpha|) - |\alpha|, S[q-1])$
 6      for $p$ in $P_q$ do
 7          report pair $(p, q, |\alpha|)$

 8  $q_{min}, q_{sec} = \mathsf{ColorMin}(H_2), \mathsf{ColorSec}(H_2)$
 9  $P' = \mathsf{ColorFind}(\bar{H}_1, q_{sec} + |\alpha| + g(|\alpha|), \#)$
10  $P'' = \mathsf{ColorFind}(\bar{H}_1, q_{min} + |\alpha| + g(|\alpha|), S[q_{min} - 1])$
11  for $p$ in $P' \cup P''$ do
12      $Q_p = \mathsf{ColorFind}(H_2, p - g(|\alpha|) - |\alpha|, S[p-1])$
13      for $q$ in $Q_p$ do
14          report pair $(q, p, |\alpha|)$

15  $H = \mathsf{ColorMeld}(H_1, H_2)$
16  $\bar{H} = \mathsf{ColorMeld}(\bar{H}_1, \bar{H}_2)$
```

every q in the set $Q' \cup Q''$ forms a maximal pair $(p, q, |\alpha|)$ with gap at least $g(|\alpha|)$ with every $p \leq q - g(|\alpha|) - |\alpha|$ in $LL(w_1)$ where $S[p-1] \neq S[q-1]$, i.e. with every p in the set $P_q = \mathsf{ColorFind}(H_1, q - g(|\alpha|) - |\alpha|, S[q-1])$ which by construction is non-empty. We can construct the set $Q' \cup Q''$ efficiently as follows. Every element in Q'' greater than $p_{sec} + |\alpha| + g(|\alpha|)$ is also in Q', so we can construct $Q' \cup Q''$ by concatenating Q' and what remains of Q'' after removing all elements greater than $p_{sec} + |\alpha| + g(|\alpha|)$ from it. Combined with the complexity of ColorFind this implies that we can construct the set $Q' \cup Q''$ in time proportional to $|Q'| + |Q''| \leq 2|Q' \cup Q''|$.

This leads to the formulation of Algorithm 7.4. The algorithm is similar to Algorithm 7.3 except that we maintain colored heap-trees during the traversal of the binary suffix tree. At every node we report maximal pairs of its path-label. In lines 1–7 we report all maximal pairs $(p, q, |\alpha|)$ by constructing and considering the elements in P_q for every q in $Q' \cup Q''$. In lines 8–15 we analogously report all maximal pairs $(q, p, |\alpha|)$. The correctness of the algorithm follows immediately from the above discussion. Since the operations on colored heap-trees have the same complexities as the corresponding operations on heap-tress, the running time and space requirement of the algorithm is exactly as analyzed for Algorithm 7.3. In summary we can formulate the following theorem.

Theorem 7.4 *Algorithm 7.4 finds all maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap at least $g(|\alpha|)$ in time $O(n + z)$ and space $O(n)$, where z is the number of reported pairs.*

7.5 Conclusion

We have presented efficient and flexible methods to find all maximal pairs $(i, j, |\alpha|)$ in a string under various constraints on the gap $j - i - |\alpha|$. If the gap is required to be between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, the running time is $O(n \log n + z)$ where n is the length of the string and z is the number of reported pairs. If the gap is only required to be at least $g_1(|\alpha|)$, the running time reduces to $O(n+z)$. In both cases we use space $O(n)$.

In some cases it might be interesting only to find maximal pairs $(i, j, |\alpha|)$ fulfilling additional requirements on $|\alpha|$, e.g. to filter out pairs of short substrings. This is straightforward to do using our methods by only reporting from the nodes in the binary suffix tree whose path-label α fulfills the requirements on $|\alpha|$. In other cases it might be of interest just to find the vocabulary of substrings that occur in maximal pairs. This is also straightforward to do using our methods by just reporting the path-label α of a node if we can report one or more maximal pairs from the node.

Instead of just looking for maximal pairs, it could be interesting to look for an array of occurrences of the same substring in which the gap between consecutive occurrences is bounded by some constants. This problem requires a suitable definition of a maximal array. One definition and approach is presented in [169]. Another definition inspired by the definition of a maximal pair could be to require that every pair of occurrences in the array is a maximal pair. This definition seems very restrictive. A more relaxed definition could be to only require that we cannot extend all the occurrences in the array to the left or to the right without destroying at least one pair of occurrences in the array.

Acknowledgments

This work was initiated while Christian Nørgaard Storm Pedersen and Jens Stoye were visiting Dan Gusfield at UC Davis. We would like to thank Dan Gusfield and Rob Irwing for listening to some early results.

Chapter 8

Finding Maximal Quasiperiodicities in Strings

The paper *Finding Maximal Quasiperiodicities in Strings* presented in this chapter has been published as a technical report [32] and a conference paper [33].

[32] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. Technical Report RS-99-25, BRICS, September 1999.

[33] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411, 2000.

Except for minor typographical changes the content of this chapter is equal to the technical report [32].

Finding Maximal Quasiperiodicities in Strings

Gerth Stølting Brodal[*] Christian N. S. Pedersen[*]

Abstract

Apostolico and Ehrenfeucht defined the notion of a maximal quasiperiodic substring and gave an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log^2 n)$. In this paper we give an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Our algorithm uses the suffix tree as the fundamental data structure combined with efficient methods for merging and performing multiple searches in search trees. Besides finding all maximal quasiperiodic substrings, our algorithm also marks the nodes in the suffix tree that have a superprimitive path-label.

8.1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory regularities are important in the analysis of stochastic processes. In computer science repetitive elements in strings are important in e.g. data compression, speech recognition, coding, automata and formal language theory.

A widely studied regularity in strings are consecutive occurrences of the same substring. Two consecutive occurrences of the same substring is called an occurrence of a *square* or a *tandem repeat*. In the beginning of the last century, Thue [183, 184] showed how to construct arbitrary long strings over any alphabet of more than two characters that contain no squares. Since then a lot of work have focused on developing efficient methods for counting or detecting squares in strings. Several methods that determine if a string of length n contains a square in time $O(n)$ have been presented, e.g. [132, 165, 42]. Several methods that find occurrences of squares in a string of length n in time $O(n \log n)$ plus the time it takes to output the detected squares have been presented, e.g. [41, 11, 131, 177]. Recently two methods [107, 75] have been presented that find a compact representation of all squares in a string of length n in time $O(n)$.

A way to describe the regularity of an entire string in terms of repetitive substrings is the notion of a *periodic* string. Gusfield [74, page 40] defines

[*]Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: {gerth,cstorm}@brics.dk.

string S as periodic if it can be constructed by concatenations of a shorter string α. The shortest string from which S can be generated by concatenations is the *period* of S. A string that is not periodic is *primitive*. Some regularities in strings cannot be characterized efficiently using periods or squares. To remedy this, Ehrenfeucht, as referred in [8], suggested the notation of a *quasiperiodic* string. A string S is quasiperiodic if it can be constructed by concatenations and superpositions of a shorter string α. We say that α covers S. Several strings might cover S. The shortest string that covers S is the *quasiperiod* of S. A covering of S implies that S contains a square, so by the result of Thue not all strings are quasiperiodic. A string that is not quasiperiodic is *superprimitive*. Apostolico, Farach and Iliopoulos [10] presented an algorithm that finds the quasiperiod of a given string of length n in time $O(n)$. This algorithm was simplified and made on-line by Breslauer [28]. Moore and Smyth [141] presented an algorithm that finds all substrings that covers a given string of length n in time $O(n)$.

Similar to the period of a string, the quasiperiod of a string describes a global property of the string, but quasiperiods can also be used to characterize substrings. Apostolico and Ehrenfeucht [9] introduced the notion of maximal quasiperiodic substrings of a string. Informally, a quasiperiodic substring γ of S with quasiperiod α is maximal if no extension of γ can be covered by α or αa, where a is the character following γ in S. Apostolico and Ehrenfeucht showed that the maximal quasiperiodic substrings of S correspond to path-labels of certain nodes in the suffix tree of S, and gave an algorithm that finds all maximal quasiperiodic substrings of a string of length n in time $O(n \log^2 n)$ and space $O(n \log n)$. The algorithm is based on a bottom-up traversal of the suffix tree in which maximal quasiperiodic substrings are detected at the nodes in the suffix tree by maintaining various data structures during the traversal. The general structure of the algorithm resembles the structure of the algorithm by Apostolico and Preparata [11] for finding tandem repeats.

In this paper we present an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Similar to the algorithm by Apostolico and Ehrenfeucht, our algorithm finds the maximal quasiperiodic substrings in a bottom-up traversal of the suffix tree. The improved time and space bound is a result of using efficient methods for merging and performing multiple searches in search trees, combined with observing that some of the work done, and data stored, by the Apostolico and Ehrenfeucht algorithm is avoidable. The analysis of our algorithm is based on a stronger version of the well known "smaller-half trick" used in the algorithms in [41, 11, 177] for finding tandem repeats. The stronger version of the "smaller-half trick" is hinted at in [138, Exercise 35] and stated in Lemma 8.6. In [139, Chapter 5] it is used in the analysis of finger searches. In [30] it is used in the analysis and formulation of an algorithm to find all maximal pairs with bounded gap.

Recently, and independent of our work, Iliopoulos and Mouchard in [96] report an algorithm with running time $O(n \log n)$ for finding all maximal quasiperiodic substrings in a string of length n. Their algorithm differs from our algorithm as it does not use the suffix tree as the fundamental data structure, but uses the partitioning technique used by Crochemore [41] combined with several other data structures. Finding maximal quasiperiodic substrings can thus be done in two different ways similar to the difference between the algorithms by Crochemore [41] and Apostolico and Preparata [11] for finding tandem repeats.

The rest of this paper is organized as follows. In Section 8.2 we define the preliminaries used in the rest of the paper. In Section 8.3 we state and prove properties of quasiperiodic substrings and suffix trees. In Section 8.4 we state and prove results about efficient merging of and searching in height-balanced trees. In Section 8.5 we stated our algorithm to find all maximal quasiperiodic substrings in a string. In Section 8.6 we analyze the running time of our algorithm and in Section 8.7 we show how the algorithm can be implemented to use linear space.

8.2 Definitions

In the following we let $S, \alpha, \beta, \gamma \in \Sigma^*$ denote strings over some finite alphabet Σ. We let $|s|$ denote the length of S, $S[i]$ the ith character in S for $1 \leq i \leq |S|$, and $S[i \mathinner{..} j] = S[i]S[i+1] \cdots S[j]$ a substring of S. A string α *occurs* in a string γ at position i if $\alpha = \gamma[i \mathinner{..} i+|\alpha|-1]$. We say that $\gamma[j]$, for all $i \leq j \leq i+|\alpha|-1$, is covered by the occurrence of α at position i.

A string α *covers* a string γ if every position in γ is covered by an occurrence of α. Figure 8.1 shows that $\gamma = abaabaabaabaab$ is covered by $\alpha = abaab$. Note that if α covers γ then α is both a prefix and a suffix of γ. A string is *quasiperiodic* if it can be covered by a shorter string. A string is *superprimitive* if it is not quasiperiodic, that is, if it cannot be covered by a shorter string. A superprimitive string α is a quasiperiod of a string γ if α covers γ. In Lemma 8.1 we show that if α is unique, and α is therefore denoted the *quasiperiod* of γ.

The *suffix tree* $T(S)$ of the string S is the compressed trie of all suffixes of the string $S\$$, where $\$ \notin \Sigma$. Each leaf in $T(S)$ represents a suffix $S[i \mathinner{..} n]$ of S and is annotated with the index i. We refer to the set of indices stored at the leaves in the subtree rooted at node v as the *leaf-list* of v and denote it $LL(v)$. Each edge in $T(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i \mathinner{..} n]$. We refer to the substring of S spelled by the path from the root to node v as the *path-label* of v and denote it $L(v)$. Figure 8.2 shows a suffix tree.

For a node v in $T(S)$ we partition $LL(v) = (i_1, i_2, \ldots, i_k)$, where $i_j < i_{j+1}$ for $1 \leq j < k$, into a sequence of disjoint subsequences $R_1, R_2, \ldots, R_r$, such that each R_ℓ is a maximal subsequence $i_a, i_{a+1}, \ldots, i_b$, where $i_{j+1} - i_j \leq |L(v)|$ for $a \leq j < b$. Each R_ℓ is denoted a *run* at v and represents a maximal substring of S that can be covered by $L(v)$, i.e. $L(v)$ covers $S[\min R_\ell \mathinner{..} |L(v)|-1+\max R_\ell]$, and we say that R_ℓ is a run from $\min R_\ell$ to $|L(v)|-1+\max R_\ell$. A run R_ℓ at v is said to *coalesce* at v if R_ℓ contains indices from at least two children of v, i.e. if for no child w of v we have $R_\ell \subseteq LL(w)$. We use $C(v)$ to denote the set of coalescing runs at v.

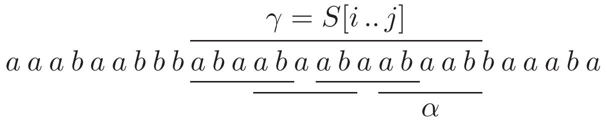


Figure 8.1: The substring $\gamma =$ *abaabaabaabaab* is a maximal quasiperiodic substring with quasiperiod $\alpha =$ *abaab*. The quasiperiod α covers the substring γ.

8.3 Maximal Quasiperiodic Substrings

If S is a string and $\gamma = S[i\,..\,j]$ a substring covered by a shorter string $\alpha = S[i\,..\,i+|\alpha|-1]$, then γ is quasiperiodic and we describe it by the triple $(i, j, |\alpha|)$. A triple $(i, j, |\alpha|)$ describes a *maximal quasiperiodic substring* of S, in the following abbreviated *MQS*, if the following requirements are satisfied.

1. $\gamma = S[i\,..\,j]$ is quasiperiodic with quasiperiod α.
2. If α covers $S[i'\,..\,j']$, where $i' \leq i \leq j \leq j'$, then $i' = i$ and $j' = j$.
3. $\alpha S[j+1]$ does not cover $S[i\,..\,j+1]$.

Figure 8.1 shows a maximal quasiperiodic substring. The problem we consider in this paper is for a string S to generate all triples $(i, j, |\alpha|)$ that describe MQSs. This problem was first studied by Apostolico and Ehrenfeucht in [9]. In the following we state important properties of quasiperiodic substrings which are essential to the algorithm to be presented.

Lemma 8.1 *Every quasiperiodic string γ has a unique quasiperiod α.*

Proof. Assume that γ is covered by two distinct superprimitive strings α and β. Since α and β are prefixes of γ we can without loss of generality assume that α is a proper prefix of β. Since α and β are suffixes of γ, then α is also a proper suffix of β. Since α and β cover γ, and α is a prefix and suffix of β it follows that α covers β, implying the contradiction that β is not superprimitive. □

Lemma 8.2 *If γ occurs at position i and j in S, and $1 \leq j-i \leq |\gamma|/2$, then γ is quasiperiodic.*

Proof. Let α be the prefix of γ of length $|\gamma|-(j-i)$, i.e. $\alpha = S[i\,..\,i+|\gamma|-(j-i)-1] = S[j\,..\,i+|\gamma|-1]$. Since $j-i \leq |\gamma|/2$ implies that $i-1+|\gamma|-(j-i) \geq j-1$, we conclude that α covers γ. □

Lemma 8.3 *If the triple $(i, j, |\alpha|)$ describes a MQS in S, then there exists a non-leaf node in the suffix tree $T(S)$ with path-label α.*

Proof. Assume that α covers the quasiperiodic substring $S[i\,..\,j]$ and that no node in $T(S)$ has path-label α. Since all occurrences of α in S are followed by the same character $a = S[i+|\alpha|]$, αa must cover $S[i\,..\,j+1]$, contradicting the maximality requirement 3. □

Lemma 8.4 *If γ is a quasiperiodic substring in S with quasiperiod α and u is a non-leaf node in the suffix tree $T(S)$ with path-label γ, then there exists an ancestor node v of u in $T(S)$ with path-label α.*

Proof. Since u is a non-leaf node in $T(S)$ of degree at least two, there exist characters a and b such that both γa and γb occur in S. Since α is a suffix of γ we then have that both αa and αb occur in S, i.e. there exist two suffixes of S having respectively prefix αa and αb, implying that there exists a node v in $T(S)$ with $L(v) = \alpha$. Since α is also a prefix of γ, v is an ancestor of u in $T(S)$. □

Lemma 8.5 *If v is a node in the suffix tree $T(S)$ with a superprimitive path-label α, then the triple $(i, j, |\alpha|)$ describes a MQS in S if and only if there is a run R from i to j that coalesces at v.*

Proof. Let $(i, j, |\alpha|)$ describe a MQS in S and assume that the run $R \in C(v)$ from i and j does not coalesce at v. Then there exists a child v' of v in $T(S)$ such that $R \subseteq LL(v')$. The first symbol along the edge from v to v' is $a = S[i + |\alpha|]$. Every occurrence of α in R is thus followed by a, i.e. αa covers $S[i \mathinner{..} j + 1]$. This contradicts the maximality requirement 3 and shows the "if" part of the lemma.

Let R be a coalescing run from i to j at node v, i.e. $L(v) = \alpha$ covers $S[i \mathinner{..} j]$, and let $a = S[j + 1]$. To show that $(i, j, |\alpha|)$ describes a MQS in S it is sufficient to show that αa does not cover $S[i \mathinner{..} j + 1]$. Since R coalesces at v, there exists a minimal $i'' \in R$ such that αa does not occur in S at position i''. If $i'' = i = \min R$ then αa cannot cover S at position i'' since it by the definition of R cannot occur at any position ℓ in S satisfying $i - |\alpha| \leq \ell \leq i$. If $i'' \neq i = \min R$ then αa occurs at $\min R$ and $\max R$, i.e. there exists $i', i''' \in R$, such that $i' < i'' < i'''$, αa occurs at i' and i''' in S, and αa does not occour at any position ℓ in S satisfying $i' < \ell < i'''$. To conclude that $(i, j, |\alpha|)$ describes a MQS we only have to show that $S[i''' - 1]$ is not covered by the occourence of αa at position i', i.e. $i''' - i' > |\alpha| + 1$. By Lemma 8.2 follows that $i'' - i' > |\alpha|/2$ and $i''' - i'' > |\alpha|/2$, so $i''' - i' \geq |\alpha| + 1$. Now assume that $i''' - i' = |\alpha| + 1$. This implies that $|\alpha|$ is odd and that $i'' - i' = i''' - i'' = (|\alpha| + 1)/2$. Using this we get

$$a = S[i' + |\alpha|] = S[i'' + (|\alpha| - 1)/2] = S[i''' + (|\alpha| - 1)/2] = S[i'' + |\alpha|] \neq a \,.$$

This contradiction shows that $(i, j, |\alpha|)$ describes a MQS and shows the "only if" part of the theorem. □

Theorem 8.1 *Let v be a non-leaf node in $T(S)$ with path-label α. Since v is a non-leaf node in $T(S)$ there exists $i_1, i_2 \in LL(v)$ such that $S[i_1 + |\alpha|] \neq S[i_2 + |\alpha|]$. The path-label α is quasiperiodic if and only if there exists an ancestor node $u \neq v$ of v in $T(S)$ with path-label β that for $\ell = 1$ or $\ell = 2$ satisfies the following two conditions.*

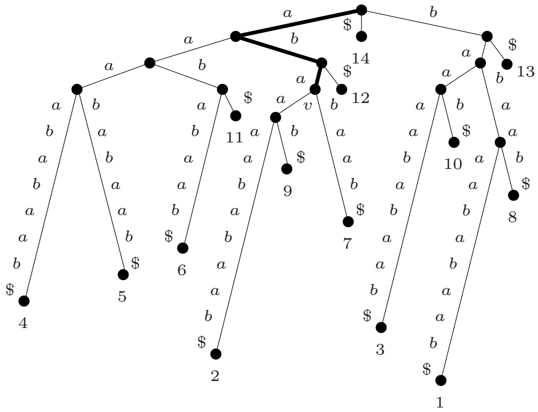


Figure 8.2: The suffix tree of the string *babaaaababaab*. Node v has a superprimitive path-label *aba*. There is a coalescing run at v from 7 to 11. Hence the substring *ababa* occurring at position 7 in *babaaaababaab* is a maximal quasiperiodic substring.

1. *Both i_ℓ and $i_\ell + |\alpha| - |\beta|$ belong to a coalescing run R at u, and*
2. *for all $i', i'' \in LL(u)$, $|i' - i''| > |\beta|/2$.*

Proof. If α is superprimitive, then no string β covers α, i.e. there exists no node u in $T(S)$ where $C(u)$ includes a run containing both i_ℓ and $i_\ell + |\alpha| - |\beta|$ for $\ell = 1$ or $\ell = 2$. If α is quasiperiodic, then we argue that the quasiperiod β of α satisfies conditions 1 and 2. Since β is superprimitive, condition 2 is satisfied by Lemma 8.2. Since β is the quasiperiod of α, we by Lemma 8.4 have that β is the path-label of a node u in $T(S)$. Since $\beta = S[i_1 \mathinner{..} i_1 + |\beta| - 1] = S[i_2 \mathinner{..} i_2 + |\beta| - 1] = S[i_1 + |\alpha| - |\beta| \mathinner{..} i_1 + |\alpha| - 1] = S[i_2 + |\alpha| - |\beta| \mathinner{..} i_2 + |\alpha| - 1]$ and $S[i_1 + |\alpha|] \neq S[i_2 + |\alpha|]$ then either $S[i_1 + |\alpha|] \neq S[i_1 + |\beta|]$ or $S[i_2 + |\alpha|] \neq S[i_2 + |\beta|]$, which implies that either i_1 and $i_1 + |\alpha| - |\beta|$ are in a coalescing run at u, or i_2 and $i_2 + |\alpha| - |\beta|$ are in a coalescing run at u. Hence, condition 1 is satisfied. □

Theorem 8.2 *A triple $(i, j, |\alpha|)$ describes a MQS in S if and only if the following three requirements are satisfied*

1. *There exists a non-leaf node v in $T(S)$ with path-label α.*
2. *The path-label α is superprimitive.*
3. *There exists a coalescing run R from i to j at v.*

Proof. The theorem follows directly from the definition of MQS, Lemma 8.3 and Lemma 8.5. □

Figure 8.2 illustrates the properties described by Theorem 8.2.

8.4 Searching and Merging Height-Balanced Trees

In this section we consider various operations on height-balanced binary trees, e.g. AVL-trees [1], and present an extension of the well-known "smaller-half trick" which implies a non-trivial bound on the time it takes to perform a sequence of operations on height-balanced binary trees. This bound is essential to the running time of our algorithm for finding maximal quasiperiodic substrings to be presented in the next section.

A height-balanced tree is a binary search tree where each node stores an element from a sorted list, such that for each node v, the elements in the left subtree of v are smaller than the element at v, and the elements in the right subtree of v are larger than the element at v. A height-balanced tree satisfies that for each node v, the heights of the left and right subtree of v differ by at most one. Figure 8.3 shows a height-balanced tree with 15 elements. A height-balanced tree with n elements has height $O(\log n)$, and element insertions, deletions, and membership queries can be performed in time $O(\log n)$, where updates are based on performing left and right rotations in the tree. We refer to [2] for further details.

For a sorted list $L = (x_1, \ldots, x_n)$ of n distinct elements, and an element x and a value δ, we define the following functions which capture the notation of *predecessors* and *successors* of an element, and the notation of Δ-*predecessors* and Δ-*successors* which in Section 8.5 will be used to compute the head and the tail of a coalescing run:

$$\begin{aligned}
\text{pred}(L, x) &= \max\{y \in L \mid y \leq x\} , \\
\text{succ}(L, x) &= \min\{y \in L \mid y \geq x\} , \\
\text{max-gap}(L) &= \max\{0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\} , \\
\Delta\text{-pred}(L, \delta, x) &= \min\{y \in L \mid y \leq x \ \wedge \ \text{max-gap}(L \cap [y, x]) \leq \delta\} , \\
\Delta\text{-succ}(L, \delta, x) &= \max\{y \in L \mid y \geq x \ \wedge \ \text{max-gap}(L \cap [x, y]) \leq \delta\} .
\end{aligned}$$

For an example, consider the list $L = (5, 7, 13, 14, 17, 21, 25, 30, 31)$. In this list $\text{pred}(L, 20) = 17$, $\text{succ}(L, 20) = 21$, $\text{max-gap}(L) = 13 - 7 = 6$, $\Delta\text{-pred}(L, 4, 20) = 13$, and $\Delta\text{-succ}(L, 4, 20) = 25$. Note that by definition $\text{pred}(L, x) = \Delta\text{-pred}(L, 0, x)$ and $\text{succ}(L, x) = \Delta\text{-succ}(L, 0, x)$.

We consider an extension of hight-balanced trees where each node v in addition to $\mathsf{key}(v)$, $\mathsf{height}(v)$, $\mathsf{left}(v)$, $\mathsf{right}(v)$, and $\mathsf{parent}(v)$, which respectively stores the element at v, the height of the subtree T_v rooted at v, pointers to the left and right children of v and a pointer to the parent node of v, also stores the following information: $\mathsf{previous}(v)$ and $\mathsf{next}(v)$ are pointers to the nodes which store the immediate predecessor and successor elements of $\mathsf{key}(v)$ in the sorted list, $\mathsf{min}(v)$ and $\mathsf{max}(v)$ are pointers to the nodes storing the smallest and largest elements in the subtree rooted at v, and $\mathsf{max\text{-}gap}(v)$ is the value of max-gap applied to the list of all elements in the subtree T_v rooted at v. Figure 8.3 shows a height-balanced tree and the corresponding extended height-balanced tree (previous and next pointers are omitted in the figure).

If v has a left child v_1, $\mathsf{min}(v)$ points to $\mathsf{min}(v_1)$. Otherwise $\mathsf{min}(v)$ points to v. Symmetrically, if v has a right child v_2, $\mathsf{max}(v)$ points to $\mathsf{max}(v_2)$. Oth-

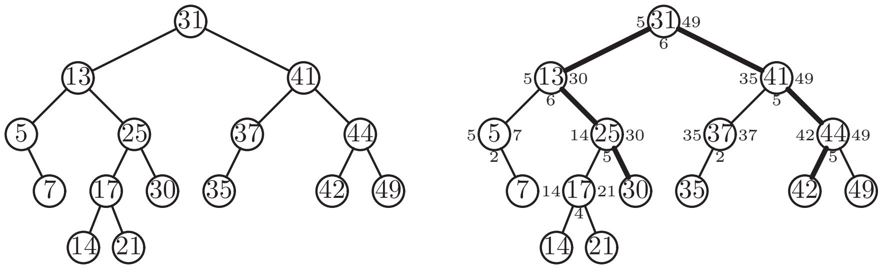


Figure 8.3: A height-balanced tree with 15 elements, and the corresponding extended height-balanced tree. Each node in the extended height-balanced tree with at least one child is annotated with min (left), max (right) and $\mathsf{max\text{-}gap}$ (bottom). The emphasized path is the search path for $\Delta\text{-Pred}(T, 4, 42)$

erwise $\mathsf{max}(v)$ points to v. If v stores element e and has a left child v_1 and a right child v_2, then $\mathsf{max\text{-}gap}(v)$ can be computed as

$$\begin{aligned}\mathsf{max\text{-}gap}(v) = \max\{0, \mathsf{max\text{-}gap}(v_1), \mathsf{max\text{-}gap}(v_2),\\ \mathsf{key}(v) - \mathsf{key}(\mathsf{max}(v_1)), \mathsf{key}(\mathsf{min}(v_2)) - \mathsf{key}(v)\}\,. \quad (8.1)\end{aligned}$$

If v_1 and/or v_2 do not exist, then the expression is reduced by removing the parts of the expression involving the missing nodes/node. The equation can be used to recompute the information at nodes being rotated when rebalancing a height-balanced search tree. Similar to the function max-gap(L) and the operation $\mathsf{max\text{-}gap}(v)$, we can define and support the function min-gap(L) and the operation $\mathsf{min\text{-}gap}(v)$. The operations we consider supported for an extended height-balanced tree T are the following, where $e_1, \ldots, e_k$ denotes a sorted list of k distinct elements. The output of the four last operations is a list of k pointers to nodes in T containing the answer to each search key e_i.

- $\mathsf{MultiInsert}(T, e_1, \ldots, e_k)$ inserts (or merges) the k elements into T.
- $\mathsf{MultiPred}(T, e_1, \ldots, e_k)$ for each e_i finds $\text{pred}(T, e_i)$.
- $\mathsf{MultiSucc}(T, e_1, \ldots, e_k)$ for each e_i finds $\text{succ}(T, e_i)$.
- $\mathsf{Multi\text{-}\Delta\text{-}Pred}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds $\Delta\text{-pred}(T, \delta, e_i)$.
- $\mathsf{Multi\text{-}\Delta\text{-}Succ}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds $\Delta\text{-succ}(T, \delta, e_i)$.

We merge two height-balanced trees T and T', $|T| \geq |T'|$, by inserting the elements in T' into T, i.e. $\mathsf{MultiInsert}(T, e_1, e_2, \ldots, e_k)$ where $e_1, e_2, \ldots, e_k$ are the elements in T' in sorted order. The following theorem states the running time of the operations.

Theorem 8.3 *Each of the operations* MultiInsert, MultiPred, MultiSucc, Multi-Δ-Pred, *and* Multi-Δ-Succ *can be performed in time $O(k \cdot \max\{1, \log(n/k)\})$, where n is the size of the tree and k is the number elements to be inserted or searched for.*

Proof. If $k \geq n$, the theorem follows immediately. In the following we therefore assume $k \leq n$. Brown and Tarjan in [34] show how to merge two (standard) height-balanced trees in time $O(k \cdot \max\{1, \log(n/k)\})$, especially their algorithm performs k top-down searches in time $O(k \cdot \max\{1, \log(n/k)\})$. Since a search for an element e either finds the element e or the predecessor/successor of e it follows that $\mathsf{MultiPred}$ and $\mathsf{MultiSucc}$ can be computed in time $O(k \cdot \max\{1, \log(n/k)\})$ using the $\mathsf{previous}$ and next pointers. The implementation of $\mathsf{MultiInsert}$ follows from the algorithm of [34] by observing that only the $O(k \cdot \max\{1, \log(n/k)\})$ nodes visited by the merging need to have their associated min, max and $\mathsf{max\text{-}gap}$ information recomputed due to the inserted elements, and the recomputation can be done in a traversal of these nodes in time $O(k \cdot \max\{1, \log(n/k)\})$ using Equation 8.1.

We now consider the $\mathsf{Multi\text{-}\Delta\text{-}Pred}$ operation. The $\mathsf{Multi\text{-}\Delta\text{-}Succ}$ operation is implemented symmetrically to the $\mathsf{Multi\text{-}\Delta\text{-}Pred}$ operation, and the details of $\mathsf{Multi\text{-}\Delta\text{-}Succ}$ are therefore omitted. The first step of $\mathsf{Multi\text{-}\Delta\text{-}Pred}$ is to apply $\mathsf{MultiPred}$, such that for each e_i we find the node v_i with $\mathsf{key}(v_i) = \mathrm{pred}(T, e_i)$. By definition $\Delta\text{-pred}(T, \delta, e_i) = \Delta\text{-pred}(T, \delta, \mathsf{key}(v_i))$. Figure 8.4 contains code for computing $\Delta\text{-pred}(T, \delta, \mathsf{key}(v_i))$. The procedure $\mathsf{\Delta\text{-}pred}(v, \delta)$ finds for a node v in T the node v' with $\mathsf{key}(v') = \Delta\text{-pred}(T, \delta, \mathsf{key}(v))$. The procedure uses the two recursive procedures $\mathsf{\Delta\text{-}pred\text{-}max}(v, \delta)$ and $\mathsf{\Delta\text{-}pred\text{-}min}(v, \delta)$ which find nodes v' and v'' satisfying respectively $\mathsf{key}(v') = \Delta\text{-pred}(T_v, \delta, \mathsf{key}(\mathsf{max}(v)))$ and $\mathsf{key}(v'') = \Delta\text{-pred}(T, \delta, \mathsf{key}(\mathsf{min}(v)))$. Note that $\mathsf{\Delta\text{-}pred\text{-}max}$ only has search domain T_v. The search $\mathsf{\Delta\text{-}pred}(v, \delta)$ basically proceeds in two steps: in the first step a path from v is followed upwards to some ancestor w of v using $\mathsf{\Delta\text{-}pred\text{-}min}$, and in the second step a path is followed from w to the descended of w with key $\Delta\text{-pred}(T, \delta, \mathsf{key}(v))$ using $\mathsf{\Delta\text{-}pred\text{-}max}$. See Figure 8.3 for a possible search path.

A Δ-predecessor search can be done in time $O(\log n)$, implying that we can find k Δ-predecessors in time $O(k \log n)$. To improve this time bound we apply dynamic programming. Observe that each call to $\mathsf{\Delta\text{-}pred\text{-}min}$ corresponds to following a child-parent edge and each call to $\mathsf{\Delta\text{-}pred\text{-}max}$ corresponds to following a parent-child edge. By memorizing the results of the calls to $\mathsf{\Delta\text{-}pred\text{-}min}$ and $\mathsf{\Delta\text{-}pred\text{-}max}$ it follows that each edge is "traversed" in each direction at most once, that all calls to $\mathsf{\Delta\text{-}pred\text{-}min}$ and textsfΔ-pred-max correspond to edges in at most k leaf-to-root paths.

From [34, Lemma 6] we have the statement: If T is a height-balanced tree with n nodes, and T' is a subtree of T with at most k leaves, then T' contains $O(k \cdot \max\{1, \log(n/k)\})$ nodes and edges. We conclude that $\mathsf{Multi\text{-}\Delta\text{-}Pred}$ takes time $O(k \cdot \max\{1, \log(n/k)\})$, since the time required for the k calls to $\mathsf{\Delta\text{-}Pred}$ is $O(k)$ plus the number of non-memorized recursive calls. □

If each node in a binary tree supplies a term $O(k)$, where k is the number of leaves in the smallest subtree rooted at a child of the node, then the sum over all terms is $O(N \log N)$. In the literature, this bound is often referred to as the "smaller-half trick". It is essential to the running time of several methods for finding tandem repeats [41, 11, 177]. Our method for finding maximal quasiperiodic substrings uses a stronger version of the "smaller-half

```
proc $\Delta$-pred($v, \delta$)
  if left($v$) $\neq$ nil and key($v$) $-$ key(max(left($v$))) $> \delta$
    return $v$
  if left($v$) $=$ nil or max-gap(left($v$)) $\leq \delta$
    return $\Delta$-pred-min($v, \delta$)
  return $\Delta$-pred-max(left($v$), $\delta$)

proc $\Delta$-pred-max($v, \delta$)
  if right($v$) $\neq$ nil and (max-gap(right($v$)) $> \delta$ or key(min(right($v$))) $-$ key($v$) $> \delta$)
    return $\Delta$-pred-max(right($v$), $\delta$))
  if left($v$) $=$ nil or (key($v$) $-$ key(max(left($v$))) $> \delta$
    return $v$
  return $\Delta$-pred-max(left($v$), $\delta$))

proc $\Delta$-pred-min($v, \delta$)
  if parent($v$) $=$ nil
    return min($v$)
  if $v =$ left(parent($v$))
    return $\Delta$-pred-min(parent($v$), $\delta$)
  if key(min($v$)) $-$ key(parent($v$)) $> \delta$
    return min($v$)
  if left(parent($v$)) $\neq$ nil and key(parent($v$)) $-$ key(max(left(parent($v$)))) $> \delta$
    return parent($v$)
  if left(parent($v$)) $\neq$ nil and max-gap(left(parent($v$))) $> \delta$
    return $\Delta$-pred-max(left(parent($v$)), $\delta$))
  return $\Delta$-pred-min(parent($v$), $\delta$)
```

Figure 8.4: Code for computing the Δ-predecessor of a node in an extended height-balanced tree.

trick" hinted at in [138, Exercise 35] and stated in Lemma 8.6. It implies that we at every node in a binary tree with N leaves can perform a fixed number of the operations stated in Theorem 8.3, with n and k as stated in the lemma, in total time $O(N \log N)$.

Lemma 8.6 *If each internal node v in a binary tree with N leaves supplies a term $O(k \log(n/k))$, where n is the number of leaves in the subtree rooted at v and $k \leq n/2$ is the number of leaves in the smallest subtree rooted at a child of v, then the sum over all terms is $O(N \log N)$.*

Proof. As the terms are $O(k \log(n/k))$ we can find constants, a and b, such that the terms are upper bounded by $a + bk \log(n/k)$. We will by induction in the number of leaves of the binary tree prove that the sum is upper bounded by $(N-1)a + bN \log N = O(N \log N)$. If the tree is a leaf then the upper bound holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $N-1$ leaves. Consider a tree with N leaves where the number of leaves in the subtrees rooted at the two children of the root are k and $N-k$ where $0 < k \leq N/2$. According to the induction hypothesis the sum over all nodes in these two subtrees, i.e. the sum over all nodes of in the tree except the root, is bounded by $(k-1)a + bk \log k + ((N-k)-1)a + b(N-k)\log(N-k)$.

The the entire sum is thus bounded by

$$\begin{aligned} a + bk\log(N/k) + (k-1)a + bk\log k + ((N-k)-1)a + b(N-k)\log(N-k) \\ &= (N-1)a + bk\log N + b(N-k)\log(N-k) \\ &< (N-1)a + bk\log N + b(N-k)\log N \\ &= (N-1)a + bN\log N \end{aligned}$$

which proves the lemma. □

8.5 Algorithm

The algorithm to find all maximal quasiperiodic substrings in a string S of length n first constructs the suffix tree $T(S)$ of S in time $O(n)$ using any existing suffix tree construction algorithm, e.g. [203, 137, 189], and then processes $T(S)$ in two phases. Each phase involves one or more traversals of $T(S)$. In the first phase the algorithm identifies all nodes of $T(S)$ with a superprimitive path-label. In the second phase the algorithm reports the maximal quasiperiodic substrings in S. This is done by reporting the coalescing runs at the nodes which in the first phase were identified to have superprimitive path-labels.

To identify nodes with superprimitive path-labels we apply the concepts of *questions*, *characteristic occurrences* of a path-label, and *sentinels* of a node. Let v be a non-leaf node in $T(S)$ and $u \neq v$ an ancestor node of v in $T(S)$. Let v_1 and v_2 be the two leftmost children of v, and $i_1 = \min(LL(v_1))$ and $i_2 = \min(LL(v_2))$. A *question posed to* u is a triple (i, j, v) where $i \in LL(v) \subset LL(u)$ and $j = i + |L(v)| - |L(u)| \in LL(u)$, and the answer to the question is true if and only if i and j are in the same coalescing run at u.

We define the two occurrences of $L(v)$ at positions i_1 and i_2 to be the *characteristic occurrences* of $L(v)$, and define the *sentinels* $\hat{v}_1$ and $\hat{v}_2$ of v as the positions immediately after the two characteristic occurrences of $L(v)$, i.e. $\hat{v}_1 = i_1 + |L(v)|$ and $\hat{v}_2 = i_2 + |L(v)|$. Since i_1 and i_2 are indices in leaf-lists of two distinct children of v, we have $S[\hat{v}_1] \neq S[\hat{v}_2]$. In the following we let $SL(v)$ be the list of the sentinels of the nodes in the subtree rooted at v in $T(S)$. Since there are two sentinels for each non-leaf node, $|SL(v)| \leq 2|LL(v)| - 2$.

Theorem 8.1 implies the following technical lemma which forms the basis for detecting nodes with superprimitive path-labels in $T(S)$.

Lemma 8.7 *The path-label $L(v)$ is quasiperiodic if and only if there exists a sentinel $\hat{v}$ of v, and an ancestor w of v (possibly $w = v$) for which there exists $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w))\, ;\, \hat{v}[$ such that $(\hat{v} - |L(v)|, j, v)$ is a question that can be posed and answered successfully at an ancestor node $u \neq v$ of w (possibly $u = w$) with $|L(u)| = \hat{v} - j$ and $\text{min-gap}(LL(u)) > |L(u)|/2$.*

Proof. If there exists a question $(\hat{v} - |L(v)|, \hat{v} - |L(u)|, v)$ that can be answered successfully at u, then $\hat{v} - |L(v)|$ and $\hat{v} - |L(u)|$ are in the same run at u, i.e. $L(u)$ covers $L(v)$ and $L(v)$ is quasiperiodic. If $L(v)$ is quasiperiodic, we have from Theorem 8.1 that there for $i_\ell = \hat{v}_\ell - |L(v)|$, where $\ell = 1$ or $\ell = 2$, exists an ancestor node $u \neq v$ of v where both i_ℓ and $i_\ell + |L(v)| - |L(u)|$ belong to a coalescing run at u and min-gap$(LL(u)) > |L(u)|/2$. The lemma follows by letting $w = u$ and $j = \hat{v}_\ell - |L(u)|$. □

Since the position j and the sentinel $\hat{v}$ uniquely determine the question $(\hat{v} - |L(v)|, j, v)$, it follows that to decide the superprimitivity of all nodes it is sufficient for each node w to find all pairs $(\hat{v}, j)$ where $\hat{v} \in SL(w)$ and $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w))\,;\, \hat{v}[$, or equivalently $j \in LL(w)$ and $\hat{v} \in SL(w) \cap]j\,;\, j + 2 \cdot \text{min-gap}(LL(w))[$. Furthermore, if $\hat{v}$ and j result in a question at w, but $j \in LL(w')$ and $\hat{v} \in SL(w')$ for some child w' of w, then $\hat{v}$ and j result in the same question at w' since min-gap$(LL(w')) \geq$ min-gap$(LL(w))$, i.e. we only need to find all pairs $(\hat{v}, j)$ at w where $\hat{v}$ and j come from two distinct children of w. We can now state the details of the algorithm.

Phase I – Marking Nodes with Quasiperiodic Path-Labels

In Phase I we mark all nodes in $T(S)$ that have a quasiperiodic path-label by performing three traversals of $T(S)$. We first make a depth-first traversal of $T(S)$ where we for each node v compute min-gap$(LL(v))$. We do this by constructing for each node v a search tree $T_{LL}(v)$ that stores $LL(v)$ and supports the operations in Section 8.4. In particular the root of $T_{LL}(v)$ should store the value $\mathsf{min\text{-}gap}(T_{LL}(v))$ to be assigned to v. If v is a leaf, $T_{LL}(v)$ only contains the index annotated to v. If v is an internal node, we construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right when these have been computed. If the children of v are $v_1, \ldots, v_k$ we merge $T_{LL}(v_1), \ldots, T_{LL}(v_{i+1})$ by performing a binary merge of $T_{LL}(v_{i+1})$ with the result of merging $T_{LL}(v_1), \ldots, T_{LL}(v_i)$. As a side effect of computing $T_{LL}(v)$ the T_{LL} trees of the children of v are destroyed.

We pose and answer questions in two traversals of $T(S)$ explained below as Step 1 and Step 2. For each node v we let $Q(v)$ contain the list of questions posed at v. Initially $Q(v)$ is empty.

Step 1 (Generating Questions) In this step we perform a depth-first traversal of $T(S)$. At each node v we construct search trees $T_{LL}(v)$ and $T_{SL}(v)$ which store respectively $LL(v)$ and $SL(v)$ and support the operations mentioned in Section 8.4. For a non-leaf node v with leftmost children v_1 and v_2, we compute the sentinels of v as $\hat{v}_1 = \mathsf{min}(T_{LL}(v_1)) + |LL(v_1)|$ and $\hat{v}_2 = \mathsf{min}(T_{LL}(v_2)) + |LL(v_1)|$. The T_{LL} trees need to be recomputed since these are destroyed in the first traversal of $T(S)$. The computation of $T_{SL}(v)$ is done similarly to the computation of $T_{LL}(v)$ by merging the T_{SL} lists of the children of v from left-to-right, except that after the merging the T_{SL} trees of the children we also need to insert the two sentinels $\hat{v}_1$ and $\hat{v}_2$ in $T_{SL}(v)$.

We visit node v, and call it the *current node*, when the T_{LL} and T_{SL} trees at the children of v are available. During the traversal we maintain an array depth such that $\mathsf{depth}(k)$ refers to the node u on the path from the current node to the root with $|L(u)| = k$ if such a node exists. Otherwise $\mathsf{depth}(k)$ is undef. We maintain depth by setting $\mathsf{depth}(|L(u)|)$ to u when we arrive at node u from its parent, and by setting $\mathsf{depth}(|L(u)|)$ to undef when we return from node u to its parent.

When v is the current node we have from Lemma 8.7 that it is sufficient to generate questions for pairs $(\hat{w}, j)$ where $\hat{w}$ and j come from two different children of v. We do this while merging the T_{LL} and T_{SL} trees of the children. Let the children of v be $v_1, \dots, v_k$. Assume $LL_i = LL(v_1) \cup \cdots \cup LL(v_i)$ and $SL_i = SL(v_1) \cup \cdots \cup SL(v_i)$ has been computed as T_{LL_i} and T_{SL_i} and we are in the process of computing LL_{i+1} and SL_{i+1}. The questions we need to generate while computing LL_{i+1} and SL_{i+1} are those where $j \in LL_i$ and $\hat{w} \in SL(v_{i+1})$ or $j \in LL(v_{i+1})$ and $\hat{w} \in SL_i$. Assume $j \in T_{LL}$ and $\hat{w} \in T_{SL}$, where either $T_{LL} = T_{LL_i}$ and $T_{SL} = T_{SL}(v_{i+1})$ or $T_{LL} = T_{LL}(v_{i+1})$ and $T_{SL} = T_{SL_i}$. There are two cases. If $|T_{LL}| \leq |T_{SL}|$ we locate each $j \in T_{LL}$ in T_{SL} by performing a $\mathsf{MultiSucc}$ operation. Using the next pointers we can then for each j report those $\hat{w} \in T_{SL}$ where $\hat{w} \in \,]j\,;\, j + 2 \cdot \text{min-gap}(LL(v))[$. If $|T_{LL}| > |T_{SL}|$ we locate each $\hat{w} \in T_{SL}$ in T_{LL} by performing a $\mathsf{MultiPred}$ operation. Using the $\mathsf{previous}$ pointers we can then for each $\hat{w}$ report those $j \in T_{SL}$ where $j \in \,]\hat{w} - 2 \cdot \text{min-gap}(LL(v))\,;\, \hat{w}[$. The two sentinels $\hat{v}_1$ and $\hat{v}_2$ of v are handled similarly to the later case by performing two searches in $T_{LL}(v)$ and using the $\mathsf{previous}$ pointers to generate the required pairs involving the sentinels $\hat{v}_1$ and $\hat{v}_2$ of v.

For a pair $(\hat{w}, j)$ that is generated at the current node v, we generate a question $(\hat{w} - |L(w)|, j, w)$ about descendent w of v with sentinel $\hat{w}$, and pose the question at ancestor $u = \mathsf{depth}(\hat{w} - j)$ by inserting $(\hat{w} - |L(w)|, j, w)$ into $Q(u)$. If such an ancestor u does not exists, i.e. $\mathsf{depth}(\hat{w} - j)$ is undef, or $\mathsf{min\text{-}gap}(u) \leq |L(u)|/2$ then no question is posed.

Step 2 (Answering Questions) Let $Q(v)$ be the set of questions posed at node v in Step 1. If there is a coalescing run R in $C(v)$ and a question (i, j, w) in $Q(v)$ such that $\min R \leq i < j \leq \max R$, then i and j are in the same coalescing run at v and we mark node w as having a quasiperiodic path-label.

We identify each coalescing run R in $C(v)$ by the tuple $(\min R, \max R)$. We *answer* question (i, j, w) in $Q(v)$ by deciding if there is a run $(\min R, \max R)$ in $C(v)$ such that $\min R \leq i < j \leq \max R$. If the questions (i, j, w) in $Q(v)$ and runs $(\min R, \max R)$ in $C(v)$ are sorted lexicographically, we can answer all questions by a linear scan through $Q(v)$ and $C(v)$. In the following we describe how to generate $C(v)$ in sorted order and how to sort $Q(v)$.

Constructing Coalescing Runs The coalescing runs are generated in a traversal of $T(S)$. At each node v we construct $T_{LL}(v)$ storing $LL(v)$. We construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right. A coalescing run R in $LL(v)$ contains an index from at least two distinct children of v, i.e. there are indices $i' \in LL(v_1)$ and $i'' \in LL(v_2)$ in R for two distinct children v_1 and v_2 of v such that $i' < i''$ are neighbors in $LL(v)$ and $i'' - i' \leq |L(v)|$. We say that i' is a *seed* of R. We identify R by the tuple $(\min R, \max R)$. We have $\min R = \Delta\text{-pred}(LL(v), |L(v)|, i')$ and $\max R = \Delta\text{-succ}(LL(v), |L(v)|, i')$.

To construct $C(v)$ we collect seeds $i_{r_1}, i_{r_2}, \dots, i_{r_k}$ of every coalescing run in $LL(v)$ in sorted order. This done by checking while merging the T_{LL} trees of the children of v if an index gets a new neighbor in which case the index can be identified as a seed. Since each insertion at most generates two seeds we can collect all seeds into a sorted list while performing the merging. From the seeds we can compute the first and last index of the coalescing runs by doing $\mathsf{Multi\text{-}\Delta\text{-}Pred}(T_{LL}(v), |L(v)|, i_{r_1}, i_{r_2}, \ldots, i_{r_k})$ and $\mathsf{Multi\text{-}\Delta\text{-}Succ}(T_{LL}(v),$ $|L(v)|,$ $i_{r_1}, i_{r_2}, \ldots, i_{r_k})$. Since we might have collected several seeds of the same run, the list of coalescing runs $R_1, R_2, \ldots, R_k$ might contain doublets which can be removed by reading through the list once. Since the seeds are collected in sorted order, the resulting list of runs is also sorted.

Sorting the Questions We collect the elements in $Q(v)$ for every node v in $T(S)$ into a single list Q that contains all question (i, j, w) posed at nodes in $T(S)$. We annotate every element in Q with the node v it was collected from. By construction every question (i, j, w) posed at a node in $T(S)$ satisfies that $0 \leq i < j < n$. We can thus sort the elements in Q lexicographically with respect to i and j using radix sort. After sorting the elements in Q we distribute the questions back to the proper nodes in sorted order by a linear scan through Q.

Phase II – Reporting Maximal Quasiperiodic Substrings

After Phase I all nodes that have a quasiperiodic path-label are marked, i.e. all unmarked nodes are nodes that have a superprimitive path-label. By Theorem 8.2 we report all maximal quasiperiodic substrings by reporting the coalescing runs at every node that has a superprimitive path-label. In a traversal of the marked suffix tree we as in Phase I construct $C(v)$ at every unmarked node and report for every R in $C(v)$ the triple $(\min R, \max R, |L(v)|)$ that identifies the corresponding maximal quasiperiodic substring.

8.6 Running Time

In every phase of the algorithm we traverse the suffix tree and construct at each node v search trees that stores $LL(v)$ and/or $SL(v)$. At every node v we construct various lists by considering the children of v from left-to-right and perform a constant number of the operations in Theorem 8.3. Since the overall merging of information in $T(S)$ is done by binary merging, we by Lemma 8.6 have that this amounts to time $O(n \log n)$ in total. To generate and answer questions we use time proportional to the total number of questions generated. Lemma 8.8 states that the number of questions is bounded by $O(n \log n)$. We conclude that the running time of the algorithm is $O(n \log n)$.

Lemma 8.8 *At most $O(n \log n)$ questions are generated.*

Proof. We will prove that each of the $2n$ sentinels can at most result in the generation of $O(\log n)$ questions. Consider a sentinel $\hat{w}$ of node w and assume that it generates a question $(\hat{w} - |L(w)|, j, w)$ at node v. Since $\hat{w} - j < 2 \cdot$

min-gap($LL(v)$), j is either pred($LL(v), \hat{w} - 1$) (a question of Type A) or the left neighbor of pred($LL(v), \hat{w} - 1$) in $LL(v)$ (a question of Type B). For $\hat{w}$ we first consider all indices resulting in questions of Type A along the path from w to the root. Note that this is an increasing sequence of indices. We now show that the distance of $\hat{w}$ to the indices is geometrically decreasing, i.e. there are at most $O(\log n)$ questions generated of Type A. Let j and j' be two consecutive indices resulting in questions of Type A at node v and at an ancestor node u of v. Since $j < j' < \hat{w}$ and $j' - j \geq$ min-gap($LL(u)$) and $\hat{w} - j' < 2 \cdot$ min-gap($LL(u)$), we have that $\hat{w} - j' < \frac{2}{3}(\hat{w} - j)$. Similarly we can bound the number of questions of Type B generated for a sentinel $\hat{w}$ by $O(\log n)$. □

8.7 Achieving Linear Space

Storing the suffix tree $T(S)$ uses space $O(n)$. During a traversal of the suffix tree we construct search trees as explained. Since no element, index or sentinel, at any time is stored in more than a constant number of search trees, storing the search trees uses space $O(n)$. Unfortunately, storing the sets $C(v)$ and $Q(v)$ of coalescing runs and questions at every node v in the suffix tree uses space $O(n \log n)$. To reduce the space consumption we must thus avoid to store $C(v)$ and $Q(v)$ at all nodes simultaneously. The trick is to modify Phase I to alternate between generating and answering questions.

We observe that generating questions and coalescing runs (Step 1 and the first part of Step 2) can be done in a single traversal of the suffix tree. This traversal is Part 1 of Phase I. Answering questions (the last part of Step 1) is Part 2 of Phase I. To reduce the space used by the algorithm to $O(n)$ we modify Phase I to alternate in rounds between Part 1 (generating questions and coalescing runs) and Part 2 (answering questions).

We say that node v is *ready* if $C(v)$ is available and all questions from it has been generated, i.e. Part 1 has been performed on it. If node v is ready then all nodes in its subtree are ready. Since all questions to node v are generated at nodes in its subtree, this implies that $Q(v)$ is also available. By definition no coalescing runs are stored at non-ready nodes and Lemma 8.9 states that only $O(n)$ questions are stored at non-ready nodes. In a round we produce ready nodes (perform Part 1) until the number of questions plus coalescing runs stored at nodes readied in the round exceed n, we then answer the questions (perform Part 2) at nodes readied in the round. After a round we dispose questions and coalescing runs stored at nodes readied in the round. We continue until all nodes in the suffix tree have been visited.

Lemma 8.9 *There are at most $O(n)$ questions stored at non-ready nodes.*

Proof. Let v be a node in $T(S)$ such that all nodes on the path from v to the root are non-ready. Consider a sentinel $\hat{w}$ corresponding to a node in the subtree rooted at v. Assume that this sentinel has induced three questions $(\hat{w} - |L(w)|, j', w)$, $(\hat{w} - |L(w)|, j'', w)$ and $(\hat{w} - |L(w)|, j''', w)$, where $j' < j'' < j'''$, that are posed at ancestors of v. By choice of v, these ancestors are non-ready nodes. One of the ancestors is node $u = \mathsf{depth}(\hat{w} - j')$. Since question

$(\hat{w} - |L(w)|, j', w)$ is posed at u, min-gap$(LL(u)) > |L(u)|/2$. Since $j', j'', j''' \in LL(u)$ and $j''' - j' \leq \hat{w} - j' = |L(u)|$, it follows that min-gap$(LL(u)) \leq \min\{j'' - j', j''' - j''\} \leq |L(u)|/2$. This contradicts that min-gap$(LL(u)) > |L(u)|/2$ and shows that each sentinel has generated at most two questions to non-ready nodes. The lemma follows because there are at most $2n$ sentinels in total. □

Alternating between Part 1 and Part 2 clearly results in generating and answering the same questions as if Part 1 and Part 2 were performed without alternation. The correctness of the algorithm is thus unaffected by the modification of Phase I. Now consider the running time. The running time of a round can be divided into time spent on readying nodes (Part 1) and time spent on answering questions (Part 2). The total time spent on readying nodes is clearly unaffected by the alternation. To conclude the same for the total time spent on answering questions, we must argue that the time spent on sorting the posed questions in each round is proportional to the time otherwise spent in the round.

The crucial observation is that each round takes time $\Omega(n)$ for posing questions and identifying coalescing runs, implying that the $O(n)$ term in each radix sorting is neglectable. We conclude that the running time is unaffected by the modification of Phase I. Finally consider the space used by the modified algorithm. Besides storing the suffix tree and the search trees which uses space $O(n)$, it only stores $O(n)$ questions and coalescing runs at nodes readied in the current round (by construction of a round) and $O(n)$ questions at non-ready nodes (by Lemma 8.9). In summary we have the following theorem.

Theorem 8.4 *All maximal quasiperiodic substrings of a string of length n can be found in time $O(n \log n)$ and space $O(n)$.*

8.8 Conclusion

We have presented an algorithm that finds all maximal quasiperiodic substrings of a string of length n in time $O(n \log n)$ and space $O(n)$. Besides improving on a previous algorithm by Apostolico and Ehrenfeucht, the algorithm demonstrates the usefulness of suffix trees combined with efficient methods for merging and performing multiple searches in search trees. We believe that the techniques presented in this paper could also be useful in improving the running time of the algorithm for the string statistic problem presented by Apostolico and Preparata [12] to $O(n \log n)$.

Chapter 9

Prediction of RNA Secondary Structure

The paper *An Improved Algorithm for RNA Secondary Structure prediction* presented in this chapter has been published in part as a technical report [129], a conference paper [130] and a journal paper [128]

[129] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. An improved algorithm for RNA secondary structure prediction. Technical Report RS-99-15, BRICS, May 1999.

[130] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. Internal loops in RNA secondary structure prediction. In *Proceedings of the 3th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 260–267, 1999.

[128] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999.

The technical report presents the work described in the conference and journal paper in a unified way. Except for minor typographical changes the content of this chapter is equal to the technical report [129]. An implementation of the method for RNA secondary structure prediction presented in this chapter is available at `www.daimi.au.dk/∼rlyngsoe/zuker`.

Prediction of RNA Secondary Structure

Rune B. Lyngsø[*] Michael Zuker[†] Christian N. S. Pedersen[‡]

Abstract

Though not as abundant in known biological processes as proteins, RNA molecules serve as more than mere intermediaries between DNA and proteins, e.g. as catalytic molecules. Furthermore, RNA secondary structure prediction based on free energy rules for stacking and loop formation remains one of the few major breakthroughs in the field of structure prediction. We present a new method to evaluate all possible internal loops of size at most k in an RNA sequence, s, in time $\mathrm{O}(k|s|^2)$; this is an improvement from the previously used method that uses time $\mathrm{O}(k^2|s|^2)$. For unlimited loop size this improves the overall complexity of evaluating RNA secondary structures from $\mathrm{O}(|s|^4)$ to $\mathrm{O}(|s|^3)$ and the method applies equally well to finding the optimal structure and calculating the equilibrium partition function. We use our method to examine the soundness of setting $k = 30$, a commonly used heuristic.

9.1 Introduction

Structure prediction remains one of the most compelling, yet elusive areas of computational biology. Not yielding to overwhelming numbers and resources this area still poses a lot of interesting questions for future research. For RNA, if one restricts attention to the prediction of unknotted secondary structures, much progress has been achieved. Dynamic programming algorithms combined with the nearest neighbor model and experimentally determined free energy parameters give rigorous solutions to the problems of computing minimum free energy structures, structures that are usually close to real world optimal foldings, and partition functions that yield exact base pair probabilities.

Secondary structure in RNA is the list of base pairs that occur in a three dimensional RNA structure. According to the theory of thermodynamics the optimal foldings of an RNA sequence are those of minimum free energy, and thus the native foldings, i.e. the foldings encountered in the real world, should correspond to the optimal foldings. Furthermore, thermodynamics tells us that

[*] Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: rlyngsoe@daimi.au.dk.

[†] Institute for Biomedical Computing, Washington University, St, Louis, USA. E-mail: zuker@ibc.wustl.edu.

[‡] Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: cstorm@brics.dk.

the folding of an RNA sequence in the real world is actually a probability distribution over all possible structures, where the probability of a specific structure is proportional to an exponential of the free energy of the structure. For a set of structures, the partition function is the sum over all structures of the set of the exponentials of the free energies.

Information on the secondary structure of an RNA molecule can be used as a stepping-stone to modeling the full structure of the molecule, which in turn relates to the biological function. As recent experiments have shown that RNA molecules can undertake a wide range of different functions [88], the prediction of RNA secondary structure should continue to be important for biomolecule engineering.

A model was proposed in [186, 185] to calculate the stability (in terms of free energy) of a folded RNA molecule by adding independent contributions from base pair stacking and loop destabilizing terms from the secondary structure. This model has proven a good approximation of the forces governing RNA structure formation, thus allowing fair predictions of real structures by determining the most stable structures in the model of a given sequence.

Based on this model, algorithms for computing the most stable structures have been proposed e.g. in [213, 154]. Zuker [211] proposes a method to determine all base pairs that can participate in structures with a free energy within a specified range from the optimal. McCaskill [136] demonstrates how a related dynamic programming algorithm can be used to calculate equilibrium partition functions, which lead to exact calculations of base pair probabilities in the model.

A major problem for these algorithms is the time required to evaluate possible internal loops. In general, this requires time $O(|s|^4)$ which is often circumvented by assuming that only 'small' loops need to be considered (e.g. [136]). This risks missing some optimal large internal loops, especially when folding at high temperatures, but the time required for evaluating internal loops is reduced to $O(|s|^2)$ thus reducing the overall complexity to $O(|s|^3)$. If the stability of an internal loop can be assumed only to depend on the size of the internal loop, Waterman et. al. [199] describes how to reduce the time requirement to $O(|s|^3)$[1]. This is further improved to $O(|s|^2 \log^2 |s|)$ for convex free energy functions by Eppstein *et al.* [51]. Affine free energy functions (i.e. of the form $a + bn$, where n is the size of the loop) allows for $O(|s|^2)$ computation time by borrowing a simple method used in sequence alignment [67].

Unfortunately the currently used free energy functions for internal loops are not convex, let alone affine. Furthermore, the technique described in [51] hinges on the objective being to find a structure of maximum stability, and thus does not translate to the calculation of the partition function of [136] where a Boltzmann weighted sum of contributions to the partition function is calculated.

In this paper we will describe a method based on a property of current free energy functions for internal loops that allows all internal loops to be evaluated

[1] This method is also referred to by [136] where a combination of the above methods is proposed - a free energy function only dependent on loop size is used for large loops, while small loops are treated specially.

in time $O(|s|^3)$. This method is applicable both to determining the most stable structure and to calculating the partition function.

The rest of this paper is structured as follows. In Section 9.2 we briefly review the basic dynamic programming algorithm for RNA secondary structure prediction and introduce the notation we will be using. In Section 9.3 we present a method yielding cubic time algorithms for evaluating internal loops for certain free energy functions. We argue that this method can be used with currently used free energy functions in Section 9.3.2, and describe how the same technique can be used to calculate the contributions to the partition function from structures with internal loops in Section 9.3.3. In Section 9.4 we compare our method to the previously used method, and in Section 9.5 we present an experiment using the new algorithm to analyze a hitherto commonly used heuristic. In Section 9.6 we discuss some future directions for improvements.

9.2 Basic Dynamic Programming Algorithm

A secondary structure of a sequence s is a set S of base pairs $i \cdot j$ with $1 \leq i < j \leq |s|$, such that $\forall i \cdot j, i' \cdot j' \in S : i = i' \Leftrightarrow j = j'$. Thus, any base can take part in at most one base pair. We will further assume that the structure does not contain pseudo-knots. A pseudo-knot is two "overlapping" base pairs, that is, base pairs $i \cdot j$ and $i' \cdot j'$ with $i < i' < j < j'$.

One can view a pseudo-knot free secondary structure S as a collection of *loops* together with some *external* unpaired bases (see Figure 9.1). Let $i < k < j$ with $i \cdot j \in S$. Then k is said to be *accessible* from $i \cdot j$ if for all $i' \cdot j' \in S$ it is not the case that $i < i' < k < j' < j$. The base pair $i \cdot j$ is said to be the *exterior* base pair of (or *closing*) the loop consisting of $i \cdot j$ and all bases accessible from it. If i' and j' are accessible from $i \cdot j$ and $i' \cdot j' \in S$ – observe that for a structure without pseudo-knots either both or none of i' and j' will be accessible from $i \cdot j$ if $i' \cdot j' \in S$ – then $i' \cdot j'$ is called an *interior* base pair of the loop and is said to be accessible from $i \cdot j$. If there are no interior base pairs the loop is called a *hairpin* loop. With one interior base pair it is called a *stacked pair* if $i' = i + 1$ and $j' = j - 1$, and otherwise it is called an *internal* loop (*bulges* are a special kind of internal loops with either $i' = i + 1$ or $j' = j - 1$). Loops with more than one interior base pair are called *multibranched* loops. Unpaired bases and base pairs not accessible from any base pair are called external.

RNA secondary structure prediction is the problem of determining the most stable structure for a given sequence. We measure stability in terms of the free energy of the structure. Thus we want to find a structure of minimal free energy which we will also call an optimal structure. The energy of a secondary structure is assumed to be the sum of the energies of the loops of the structure and furthermore the loops are assumed to be independent, that is, the energy of a loop only depends on the loop and not on the rest of the structure [185].

Based on these assumptions one can specify a recursion to calculate the energy of the optimal structure for a sequence s [213, 154]. Before presenting our improvement to the part of the algorithm dealing with internal loops, we will briefly review the hitherto used method. We use the same notation as

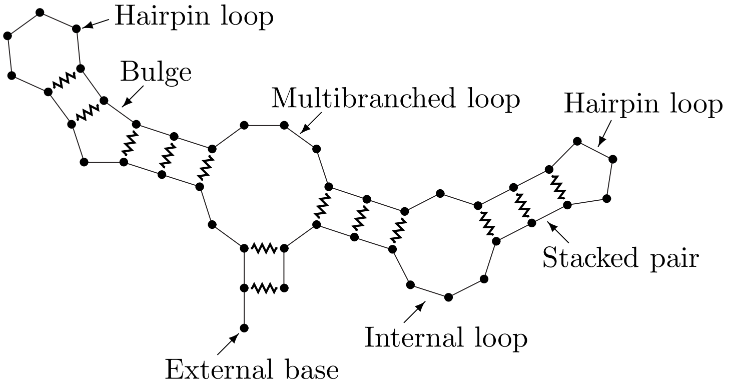


Figure 9.1: An example RNA structure. Bases are depicted by circles, the RNA backbone by straight lines and base pairings by zigzagged lines.

in [188]. Four arrays[2] – W, V, VBI and VM – are used to hold the minimal free energy of certain restricted structures of subsequences of s. The entries of these arrays are interdependent and can be calculated recursively using prespecified free energy functions – eS, eH, eL and eM – for the contributions from the various types of loops as follows.

- The energy of an optimal structure of the subsequence from 1 through i:

$$W(i) = \min\{W(i-1), \min_{1<j\leq i}\{W(j-1) + V(j,i)\}\}. \tag{9.1}$$

- The energy of an optimal structure of the subsequence from i through j closed by $i \cdot j$:

$$V(i,j) = \min\{eH(i,j), eS(i,j) + V(i+1,j-1), \\ VBI(i,j), VM(i,j)\} \tag{9.2}$$

  where $eH(i,j)$ is the energy of a hairpin loop closed by $i \cdot j$ and $eS(i,j)$ is the energy of stacking base pair $i \cdot j$ with $i+1 \cdot j-1$.

- The energy of an optimal structure of the subsequence from i through j where $i \cdot j$ closes a bulge or an internal loop:

$$VBI(i,j) = \min_{\substack{i<i'<j'<j \\ i'-i+j-j'>2}} \{eL(i,j,i',j') + V(i',j')\} \tag{9.3}$$

  where $eL(i,j,i',j')$ is the energy of a bulge or internal loop with exterior base pair $i \cdot j$ and interior base pair $i' \cdot j'$.

[2] Actually two arrays – V and W – suffices, but we will use four arrays to simplify the description. Below we will introduce a fifth array WM that will also be needed in an efficient implementation.

- The energy of an optimal structure of the subsequence from i through j where $i \cdot j$ closes a multibranched loop:

  $$VM(i,j) = \min_{\substack{i<i_1<j_1< \\ \cdots \\ <i_k<j_k<j}} \{eM(i,j,i_1,j_1,\ldots,i_k,j_k) + \sum_{l=1}^{k} V(i_l,j_l)\} \tag{9.4}$$

  where $k > 1$ and $eM(i,j,i_1,j_1,\ldots,i_k,j_k)$ is the energy of a multibranched loop with exterior base pair $i \cdot j$ and interior base pairs $i_1 \cdot j_1, \ldots, i_k \cdot j_k$.

When all entries of these arrays have been filled out, $W(|s|)$ contains the free energy for optimal structures and an optimal structure can be determined by backtracking the calculations that led to this free energy.

To make the problem of determining the optimal secondary structure tractable the following simplifying assumption is often made. The energy of multibranched loops can be decomposed into linear contributions from the number of unpaired bases in the loop, the number of branches in the loop and a constant [212][3], that is

$$\begin{aligned} &eM(i,j,i_1,j_1,\ldots,i_k,j_k) = \\ &\qquad a + bk + c\Big(i_1 - i - 1 + j - j_k - 1 + \sum_{l=1}^{k-1}(i_{l+1} - j_l - 1) \Big). \end{aligned} \tag{9.5}$$

We introduce an extra array

- The energy of an optimal structure of the subsequence from i through j that constitutes part of a multibranched loop structure, that is, where unpaired bases and external base pairs are penalized according to Equation 9.5:

  $$\begin{aligned} WM(i,j) = \min\{&V(i,j) + b, WM(i,j-1) + c, WM(i+1,j) + c, \\ &\min_{i<k\leq j}\{WM(i,k-1) + WM(k,j)\}\} \end{aligned} \tag{9.6}$$

which enables us to restate the calculation of the energy of the optimal multibranched loop as

$$VM(i,j) = \min_{i+1<k\leq j-1}\{WM(i+1,k-1) + WM(k,j-1) + a\}. \tag{9.7}$$

Based on these recurrence relations we can by dynamic programming calculate the energy of the optimal structure in time $O(|s|^3)$ – assuming that the free energy functions can be evaluated in constant time – except for the calculation of the entries of VBI which requires $O(|s|^4)$ in total. The bottleneck of finding the optimal structures is thus the evaluation of internal loops. In the following section we will present a method to reduce the time used calculating the entries of VBI from $O(|s|^4)$ to $O(|s|^3)$, thereby improving the time complexity of the overall RNA secondary structure prediction algorithm from $O(|s|^4)$ to $O(|s|^3)$.

[3] It is known that the stability of a multibranched loop also depends on the stacking effects of the base pairs in the loop and their neighboring unpaired bases. These effects can also be handled efficiently, but for simplicity we have omitted the details here.

9.3 Efficient Evaluation of Internal Loops

Examining the recursion for internal loops one observes that two base pairs, $i \cdot j$ and $i' \cdot j'$, may be compared as candidates for the interior base pair for numerous exterior base pairs. If $V(i, j) \ll V(i', j')$, it is evident that we would not have to consider $i' \cdot j'$ as a candidate interior base pair for any entry of VBI where $i \cdot j$ would also be a candidate interior base pair.

Though it would often in practice be the case that we could a priori discard many candidate interior base pairs by the above observation, we can not in general guarantee this to be the case. To get an improvement in the worst case performance of the evaluation of internal loops, we thus have to examine properties of the energy functions for internal loop stability that will allow us to group base pairs and entries of VBI, such that we only have to make one comparison between $i \cdot j$ and $i' \cdot j'$ to determine which one would yield the more stable structure for the entire group of entries. In this section we will exploit such properties of currently used energy functions leading to an algorithm for evaluating internal loops requiring worst case time $O(|s|^3)$.

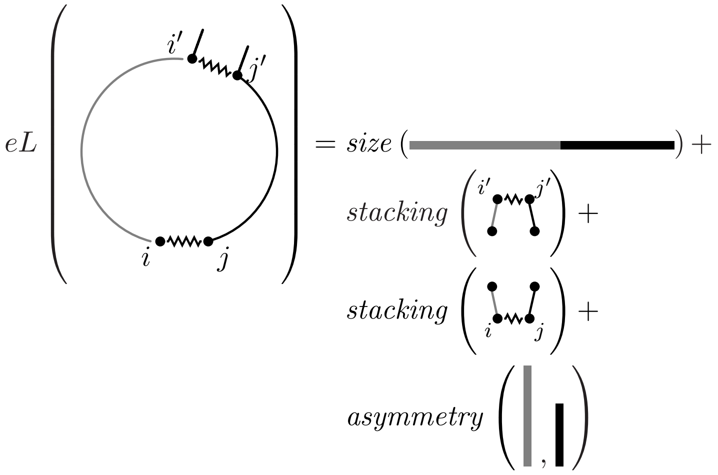


Figure 9.2: The energy function for internal loops can be split into a sum of independent contributions.

Currently used energy rules for internal loop stability (cf. [210]) split the contributions into three parts:

- An entropic term that depends on the size of the loop.

- Stacking energies for the mismatched base pairs adjacent to the enclosing (exterior *and* interior) base pairs.

- An asymmetry penalty for asymmetric loops.

With this separation we can rewrite the internal loop energy function as

$$\begin{aligned} eL(i,j,i',j') = & \operatorname{size}(i'-i+j-j'-2)+ \\ & \operatorname{stacking}(i\cdot j)+\operatorname{stacking}(i'\cdot j')+ \\ & \operatorname{asymmetry}(i'-i-1,j-j'-1). \end{aligned} \tag{9.8}$$

Figure 9.2 gives a graphical representation of these components of the internal loop energy function. In the following we will further assume that the lopsidedness and the size dependence of the asymmetry function can be separated out, or more specifically that

$$\operatorname{asymmetry}(k+1,l+1)=\operatorname{asymmetry}(k,l)+g(k+l) \tag{9.9}$$

holds. The change of the asymmetry function when varying the size while maintaining lopsidedness thus only depends on the size of the loop. This is equivalent to assuming that

$$\operatorname{asymmetry}(k,l)=\operatorname{lopsidedness}(|k-l|)+\operatorname{size}'(k+l), \tag{9.10}$$

where one can observe that the g term in Equation 9.9 corresponds to changes in the size$'$ term in Equation 9.10. This size-dependence of the asymmetry function can be moved to the size-function of the overall internal loop energy function, thus allowing us to restate the assumption of Equation 9.9 as

$$\operatorname{asymmetry}(k+1,l+1)=\operatorname{asymmetry}(k,l). \tag{9.11}$$

In the rest of this paper we will therefore omit the g term, but the formulation of Equation 9.9 might be useful when specifying or recognizing an asymmetry function obeying the assumption.

9.3.1 Finding Optimal Internal Loops

If the assumption of Equation 9.9 holds, we propose Algorithm 9.1 as an efficient alternative to compute the $VBI(i,j)$ entries in the dynamic programming algorithm for predicting RNA secondary structure. The algorithm is an extension of the ideas in [199] where an $O(n^3)$ method for calculating the entries of VBI, assuming that the stability of an internal loop only depends on the size of the loop, was presented. The rationale behind the algorithm is, that when we extend loops while retaining lopsidedness we can reuse comparisons as depicted in Figure 9.3. Thus for a pair of indices, i and j, the algorithm *does not* compute the $VBI(i,j)$ entry. Instead, if we denote all internal loops with a specific size and exterior base pair as a *class* of internal loops, the algorithm evaluates all classes of internal loops where $i\cdot j$ is the middle candidate base pair, that is, choosing $i\cdot j$ as the interior base pair results in a symmetric loop (or almost symmetric – loops of odd size will always have a lopsidedness of at least one).

Proposition 9.1 *Algorithm 9.1 computes VBI correctly under the assumption of Equation 9.9. Furthermore, the time required to compute the entire table is* $O(n^3)$.

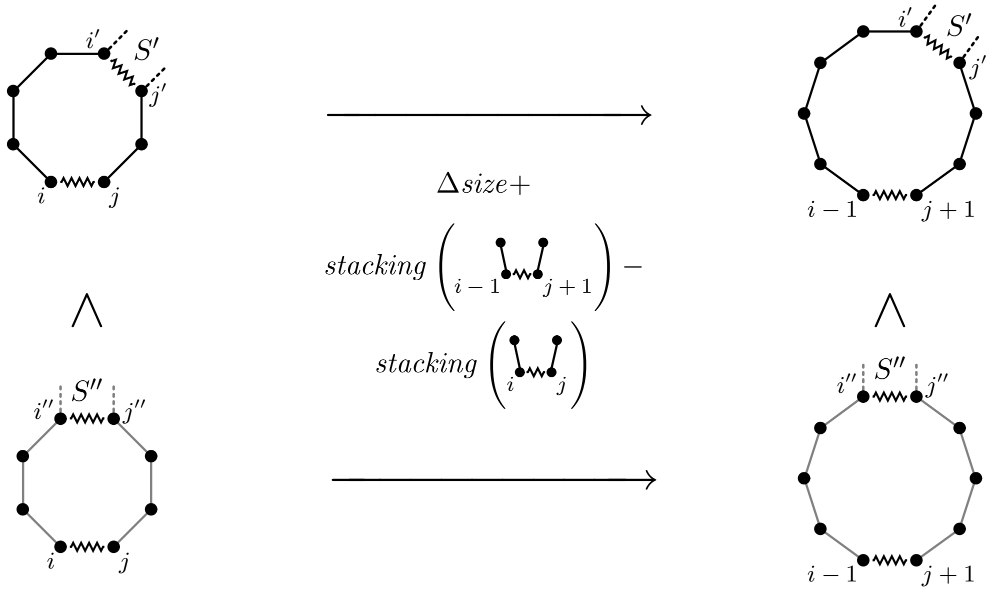


Figure 9.3: The difference in destabilizing energy when extending a loop from being closed by $i \cdot j$ to being closed by $i-1 \cdot j+1$ is determined solely by the size of the loop and the change in stacking stability of the closing base pair. We can thus reuse comparisons between different choices of interior base pairs, e.g. $i' \cdot j'$ and $i'' \cdot j''$.

The time complexity of $O(n^3)$ is easy to see, since the algorithm for each of the $O(n^2)$ pairs of indices, i and j, uses time $O(n)$. To prove the correctness of the algorithm, we will start by sketching a simpler algorithm for which the correctness is obvious, but that has the drawback of using space $O(n^3)$. Then we will argue that Algorithm 9.1 is similar to this algorithm except for the order in which the computations are carried out, that is, the order in which the different candidate interior loops for a specific entry of *VBI* are evaluated. Hence, the correctness of the simpler algorithm implies the correctness of Algorithm 9.1.

We define a new array *VBI'* such that $VBI'(i,j,l)$ is the minimal energy of an internal loop of size l with exterior base pair $i \cdot j$. The following lemma establishes a useful relationship between the entries of *VBI'*.

Lemma 9.1 *If Equation 9.9 holds, then for $l > 2$*

$$VBI'(i,j,l) \; = \; \min \begin{cases} VBI'(i+1,j-1,l-2)+ \\ \qquad \mathrm{size}(l) - \mathrm{size}(l-2)+ \\ \qquad \mathrm{stacking}(i \cdot j) - \mathrm{stacking}(i+1 \cdot j-1) \\ V(i+1,j-l-1) + eL(i,j,i+1,j-l-1) \\ V(i+l+1,j-1) + eL(i,j,i+l+1,j-1). \end{cases} \tag{9.12}$$

Proof. By definition

$$VBI'(i,j,l) = \min_{\substack{i<i'<j'<j \\ i'-i+j-j'-2=l}} \{eL(i,j,i',j') + V(i',j')\}. \tag{9.13}$$

Algorithm 9.1 Evaluation of classes of internal loops with size $2l + a$ and exterior base pair $i - l \cdot j + l + a$.

```
/* When $a = 0$ loops of even size are handled and when $a = 1$ loops of odd
size are handled; this is necessary as we increase the loop size by two in each
iteration. */
for $a = 0$ to $1$ do
  /* $E$ maintains the energy of the optimal loop except for size and external
  stacking contributions. */
  $E = \infty$
  /* Iterate through the exterior base pairs. For even sized loops we skip
  $l = 1$ as this yields a stacked base pair. */
  for $l = 2 - a$ to $\min\{i - 1, |s| - j - a\}$ do
    /* Examine the two new candidate interior base pairs, i.e. the interior
    base pairs next to the currently considered exterior base pair. */
    $E = \min\{E, V(i - l + 1, j - l + 1)+$
                    $\text{asymmetry}(0, 2l + a - 2)+$
                    $\text{stacking}(i - l + 1, j - l + 1),$
                  $V(i + a + l - 1, j + a + l - 1)+$
                    $\text{asymmetry}(2l + a - 2, 0)+$
                    $\text{stacking}(i + a + l - 1, j + a + l - 1)\}$
    /* Update VBI for the currently considered exterior base pair. */
    $VBI(i - l, j + a + l) = \min\{VBI(i - l, j + a + l),$
                    $E + \text{size}(2l + a - 2) + \text{stacking}(i - l, j + a + l)\}$
  end for
end for
```

The last two entries of Equation 9.12 handle the cases where this minimum is obtained by a bulge, that is at $i' = i + 1$ or $j' = j - 1$. Otherwise the minimum is the minimum over

$$\begin{aligned}
&eL(i, j, i', j') + V(i', j') \\
&= \text{size}(l) + \text{asymmetry}(i' - i - 1, j - j' - 1) \\
&\qquad + \text{stacking}(i \cdot j) + \text{stacking}(i' \cdot j') + V(i', j') \\
&= \text{size}(l) + \text{asymmetry}(i' - i - 2, j - j' - 2) \\
&\qquad + \text{stacking}(i \cdot j) + \text{stacking}(i' \cdot j') + V(i', j') \\
&= \text{size}(l - 2) + \text{asymmetry}(i' - i - 2, j - j' - 2) \\
&\qquad + \text{stacking}(i + 1 \cdot j - 1) + \text{stacking}(i' \cdot j') + V(i', j') \\
&\qquad + \text{size}(l) - \text{size}(l - 2) \\
&\qquad + \text{stacking}(i \cdot j) - \text{stacking}(i + 1 \cdot j - 1)
\end{aligned}$$

for all $i' < j'$ with $i' > i + 1$, $j' < j - 1$ and $i' - (i + 1) + (j - 1) - j' - 2 = l - 2$. The last two lines of the last equation are independent of i' and j', and can thus be moved out of the minimum. The minimum of the first two lines over i' and j' satisfying the above constraints is exactly $VBI'(i + 1, j - 1, l - 2)$, thus proving the lemma. □

Lemma 9.1 yields the basic recursion needed to compute each entry of VBI' in constant time[4]. It is easily observed that VBI' contains $O(n^3)$ entries and that VBI can be calculated from VBI' as

$$VBI(i,j) = \min_{l}\{VBI'(i,j,l)\}, \tag{9.14}$$

each of the $O(n^2)$ entries being computable in time $O(n)$. Thus VBI can be computed in time $O(n^3)$ including the time used to compute VBI'. Unfortunately the table VBI' requires space $O(n^3)$, thus rendering this method somewhat impractical. However, it can be observed that we only need $VBI'(i,j,l)$ at most twice, namely when

- determining whether it is a candidate for $VBI(i,j)$.
- calculating the value of $VBI'(i-1,j+1,l+2)$.

This is used in Algorithm 9.1 to avoid maintaining the VBI' table. Instead we use E to hold the value[5] that should otherwise be stored in one of the entries of VBI'. We use this value to check it as a candidate for the relevant entry of VBI, according to Equation 9.14, in the second minimum of the **for**-loop in Algorithm 9.1. After this check we only need the value to calculate the value corresponding to another entry of VBI'; this is done in the first minimum in the next iteration of the **for**-loop. Now the value can safely be discarded as it is no longer needed. It is straightforward to verify that the value that should otherwise have been stored in $VBI'(i',j',l)$ is handled when Algorithm 9.1 is invoked with $i = i' + \lfloor \frac{l}{2} \rfloor$ and $j = j' - \lceil \frac{l}{2} \rceil$. The correctness of the value maintained in E can easily be proved by induction, using Lemma 9.1.

9.3.2 The Asymmetry Function Assumption

The assumption of Equation 9.9 might seem somewhat unrealistic as, for one thing, we treat bulges just as if they were normal internal loops. If Equation 9.9 only holds for $\min(k,l) \geq c-1$ we can however modify the algorithm to handle this situation, a modification that does lead to an increase in time complexity by a factor of c, for a total time complexity of $O(cn^3)$.

This is done simply by examining all the $O(cn^3)$ loops with a stem of unpaired bases shorter than c separately, and then applying the technique of extending loops while retaining lopsidedness to the rest of the loops, starting the iteration at $l = c$ and adding or subtracting $c-1$ from the indices of the interior base pairs considered, including where they partake in the parameters of the asymmetry function. Thus bulges can be treated specially while only doubling the time complexity.

[4] This is of course assuming that entries of V are ready at hand when we need them. The cost of computing the entries of V can however be charged to V, and thus we don't have to consider it here.

[5] To avoid having to keep adding and subtracting the size and external stacking terms in Algorithm 9.1 we defer adding these terms until the value is considered as a candidate for one of the VBI entries.

Papanicolaou et. al. [156] propose an asymmetry penalty function on the form

$$\text{asymmetry}(k,l) = \min\{K, N_{k,l} f(M_{k,l})\}, \tag{9.15}$$

usually called Ninio type asymmetry penalty functions, with $N_{k,l} = |k-l|$ and $M_{k,l} = \min\{k,l,c\}$. The constants K and c and the function f are parameters of the penalty function. We observe that $N_{k+1,l+1} = N_{k,l}$ and that $M_{k+1,l+1} = M_{k,l}$ if $\min\{k,l\} \geq c$. For $\min\{k,l\} \geq c$ it thus follows that $\text{asymmetry}(k+1,l+1) = \text{asymmetry}(k,l)$, and thus asymmetry functions on this form adheres to the above relaxed assumption, allowing us to solve the RNA secondary structure prediction problem using Ninio type asymmetry penalty functions in time $\mathrm{O}(cn^3)$. In [156] an asymmetry function with $c = 5$ was proposed. A modification of the parameters based on thermodynamic studies was proposed in [162]. With these parameters $c = 1$ thus allowing us to treat only bulges specially[6].

9.3.3 Computing the Partition Function

In [136] it is described how to compute the full equilibrium partition functions and thus the probabilities of all base pairs. The method used closely mimics the free energy calculation described above, and thus it should be of no surprise that the method presented in this paper also applies to the calculation of the partition functions. In this section we will briefly sketch how to compute the internal loops' contribution to the partition functions. The reader is referred to [136] for the full details on how to calculate the partition functions.

In [136] $Q_{i,j}$ denotes the partition function on the segment from base i through base j, while $Q^b_{i,j}$ denotes the *restricted* partition function for the same sequence segment with the added constraint that bases i and j form a base pair[7]. We will specify how to calculate the contributions from structures with an internal loop closed by $i \cdot j$.

From [136, equations 4 and 7] it is seen that the contributions from these structures – if we consider a stacked pair to be an internal loop of size 0 – are

$$\sum_{i<h<l<j} e^{-eL(i,j,h,l)/kT} Q^b_{h,l}\,, \tag{9.16}$$

where [136, equation 7] uses $F_2(i,j,h,l)$ to gather the energies of all structures with an internal loop with base pairs $i \cdot j$ and $h \cdot l$, thus reducing the terms of the sum to $e^{-F_2(i,j,h,l)/kT}$.

Similar to the approach in Section 9.3.1 we define $Q^{il}_{i,j,l}$ to be the partition function for all structures with an internal loop of size l closed by $i \cdot j$, thus corresponding to $VBI'(i,j,l)$ in the energy calculations in Section 9.3.1. Now it can be proved that

[6] Sequence dependent destabilizing energies are available for internal loops of size three. These – and similar specific energy functions for small loops – can be handled as a special case without affecting the general method for calculating internal loop stability though.

[7] Thus $Q^b_{i,j}$ corresponds to $V(i,j)$ in energy calculations.

Algorithm 9.2 Evaluation of classes of internal loops with size $2l + a$ and exterior base pair $i - l \cdot j + l + a$.

```
/* Make sure to handle both even sized and odd sized loops. */
for $a = 0$ to $1$ do
  /* $Q$ maintains the partition function contribution for the current class of
  internal loops except for size and external stacking factors. */
  $Q = 0$
  /* Iterate through the exterior base pairs. For even sized loops we skip
  $l = 1$ as this yields a stacked base pair. */
  for $l = 2 - a$ to $\min\{i - 1, |s| - j - a\}$ do
    /* Add contributions from the two new interior base pairs, i.e. the inte-
    rior base pairs next to the currently considered exterior base pair. */
    $Q = Q + Q^b_{i-l+1,j-l+1} e^{-(\text{asymmetry}(0,2l+a-2)+\text{stacking}(i-l+1\cdot j-l+1))/kT}$
      $+ Q^b_{i+a+l-1,j+a+l-1} e^{-(\text{asymmetry}(2l+a-2,0)+\text{stacking}(i+a+l-1\cdot j+a+l-1))/kT}$
    /* Update $Q^b$ with contributions from the currently considered class of
    internal loops. */
    $Q^b_{i-l,j+a+l} = Q^b_{i-l,j+a+l} + Q e^{-(\text{size}(2l+a-2)+\text{stacking}(i-l\cdot j+a+l))/kT}$
  end for
end for
```

$$
\begin{aligned}
Q^{il}_{i,j,l} &= Q^{il}_{i+1,j-1,l-2} e^{(\text{size}(l-2)-\text{size}(l)+\text{stacking}(i+1\cdot j-1)-\text{stacking}(i\cdot j))/kT} \\
&\quad + Q^b_{i+1,j-l-1} e^{-eL(i,j,i+1,j-l-1)/kT} + Q^b_{i+l+1,j-1} e^{-eL(i,j,i+l+1,j-1)/kT}
\end{aligned}
\tag{9.17}
$$

by similar arguments as in the proof of Lemma 9.1. There is a slight problem if $\text{stacking}(i \cdot j) = \infty$ or $\text{stacking}(i+1 \cdot j-1) = \infty$ – that is, if bases i and j or bases $i + 1$ and $j - 1$ does not form a base pair – but in the proof of Equation 9.17 this can be handled by assuming that all stacking energies are finite. In the algorithm we handle it by postponing the multiplication with the exponential of the stacking energies until adding the contribution of $Q^{il}_{i,j,l}$ to $Q^b_{i,j}$. We can now rewrite Equation 9.16 as

$$
\sum_{l=0}^{j-i-2} Q^{il}_{i,j,l}\,, \tag{9.18}
$$

and based on Equations 9.17 and 9.18 we can now proceed to present Algorithm 9.2 to handle internal loop contributions to the partition function; the observant reader will notice the close similarity between Algorithms 9.1 and 9.2. Again it is an easy observation that the time complexity is $O(n^3)$, and the correctness of Algorithm 9.2 can be proven by arguments similar to the proof of the correctness of Algorithm 9.1.

9.4 Implementation

The method described in this paper has been implemented in *ZUKER*[8], a C program to find the optimal structure of an RNA sequence based on energy rules. To be able to compare the performance of this method to previously used methods, compiler directives determines whether the compiled code will use complete enumeration of all internal loops or the method described here, and whether only to consider loops smaller than a specified size. By this we hope to have eliminated most of the noise due to differences in implementations so as to get a comparison of the underlying methods.

We decided to test our method against the complete enumeration method, both when using a cutoff size of 30 for internal loops (a commonly used cutoff size) and when allowing loops of any size. All four methods were tested with random sequences of length 500 and 1000, respectively, and the results are summarized in Table 9.1. As expected a huge increase in performance is obtained when allowing internal loops of any size, but even when limiting internal loops to size at most 30, our method obtains a speedup of 30 – 40 % compared to the complete enumeration method.

Sequence length	500	1000
Complete enumeration, unlimited loop size	2,119 s	35,988 s
Our method, unlimited loop size	127 s	1,123 s
Complete enumeration, loop size ≤ 30	48 s	264 s
Our method, loop size ≤ 30	30 s	182 s

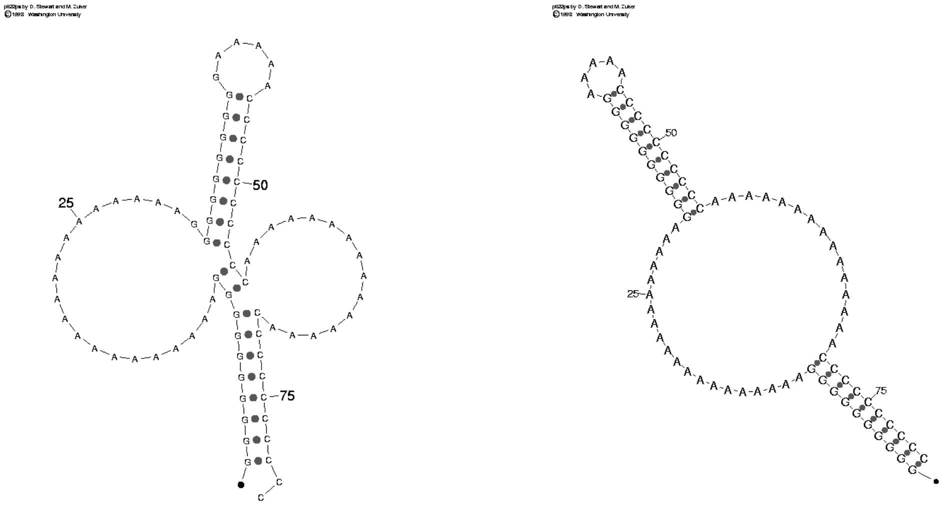
Table 9.1: Comparison of different methods to evaluate internal loops. The running times are as reported by the Unix `time` command on a Silicon Graphics Indigo 2.

The current implementation encompasses the method for calculating the optimal substructure on the parts of the sequence *excluding* the substring from i through j, thus allowing the prediction of suboptimal structures as described in [211] and calculation of base pair probabilities based on partition functions as described in [136]. We are currently working on adding coaxial stacking modifications to the multibranched loop evaluations, and on extending the program to take other parameters, e.g. mutual information or base pair confidences obtained from alignments, into account.

9.5 Experiments

To make the problem of determining the optimal secondary structure for an RNA sequence more tractable it has hitherto been common practice to limit the size of internal loops. The `mfold` server has a built-in limit of 30 and in [87] a limit of 30 is also hinted at. With the ability to make a rigorous search for the optimal structure, we decided to see whether this limit has been reasonable.

[8] ZUKER – Unlimited Ken Energy-based RNA-folding, the name reflecting that no limit is imposed on how far to look for the closing base pair of an internal loop.


plt22ps by D. Stewart and M. Zuker
© 1992 Washington University
25
50
75
plt22ps by D. Stewart and M. Zuker
© 1992 Washington University
50
25
75

(a) Maximum loop size 30; Energy: −29.6 kcal/mol

(b) No maximum loop size; Energy: −42.9 kcal/mol

Figure 9.4: Foldings of the sequence GGGGGGGGGGAAAAAAAAAAAAAAAAAAAA GGGGGGGGGGAAAAACCCCCCCCCCAAAAAAAAAAAAAAACCCCCCCCCC

9.5.1 A Constructed "Mean" Sequence

The easiest way to find a loop of size larger than 30 is of course to construct it yourself. We constructed a sequence of length 80 consisting only of C's, G's and A's (but no U's), designed to fold into two stems of 10 base pairing C's and G's separated by an internal loop of 35 unpaired A's, and with a hairpin loop consisting of 5 A's. The result of folding this sequence at 37 °C with and without a size limit of 30, respectively, is shown in Figure 9.4

One can observe that the prediction with a cutoff size of 30 does in fact pair most of the C's with G's – but instead of having the A's in one big internal loop they are folded out as two bulges. A further observation is that there can indeed be a major increase in stability by choosing one large internal loop instead of two smaller bulges.

Though this example may be cute, the interesting question of course is whether RNA sequences for which the optimal structure contains a large internal loop occur naturally. The reason that a cutoff size of 30 has been deemed reasonable is of course that no internal loops even close to this size are observed in a standard structure prediction at 37 °C. But when the temperature is increased, base pairs become less stable which may cause short stems of stacking base pairs to break up. We thus decided to look at a couple of sequences for which structure prediction at higher temperatures would be interesting.

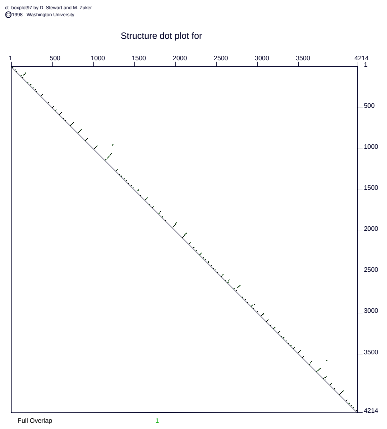


Figure 9.5: Dot-plot of the prediction of the Qβ structure at 65 °C. The absence of long range base pairings (dots far away from the diagonal) is apparent.

9.5.2 Qβ

Jacobson [98] reported on some experiments on determining structural features in Qβ denatured to various extents. It is believed that denaturing effects relates to temperature effects, and we thus chose to fold this sequence at nine different temperatures in the range from 45 °C to 100 °C to see whether we would find any of the structural features reported by Jacobson.

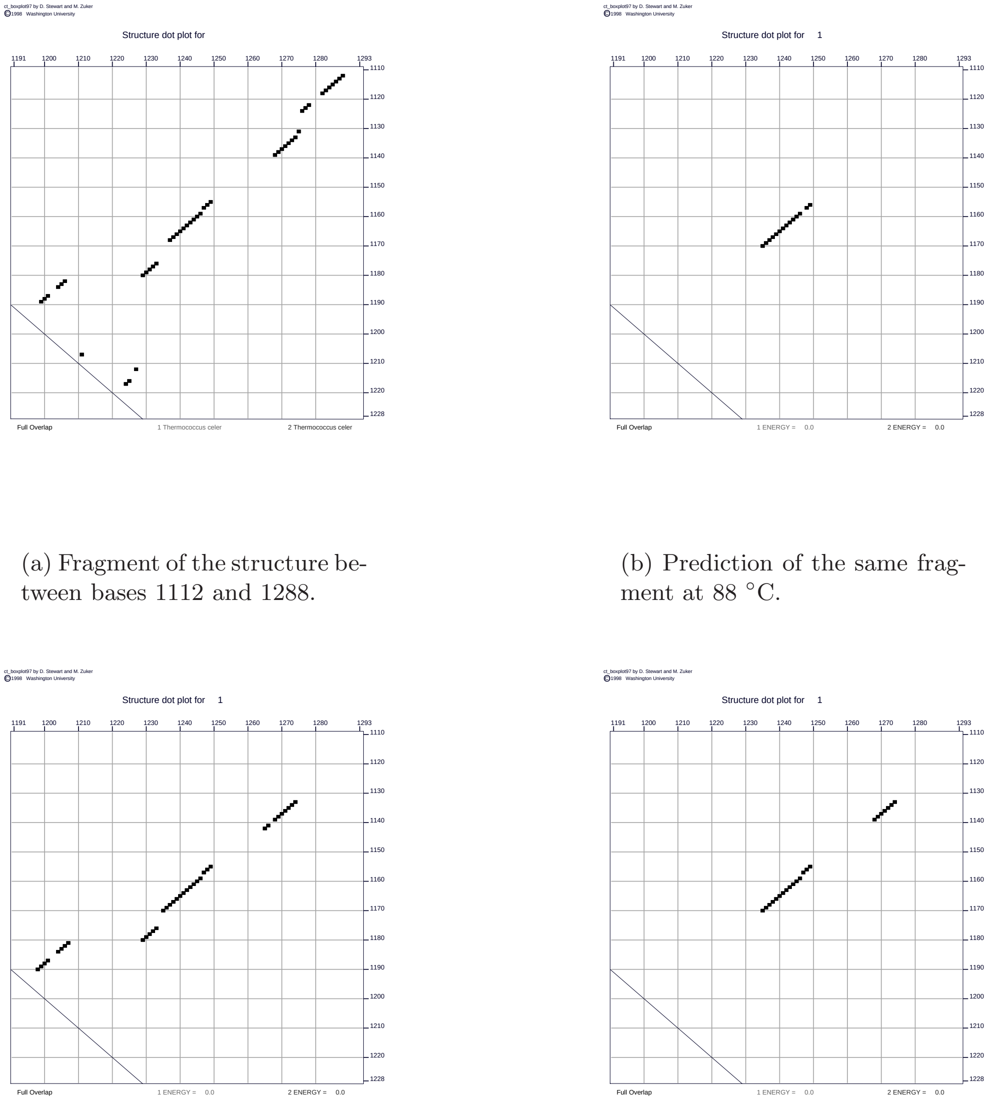
None of these predicted foldings showed any signs of the features Jacobson reported – at higher temperatures the structure simply came apart as small structural fragments, usually covering less than 100 nucleotides. Furthermore we did not observe any internal loops larger than size 25. An example prediction is shown in Figure 9.5.

9.5.3 *Thermococcus Celer*

Thermococcus celer is an organism that lives in solfataric marine water holes of Vulcano, Italy, at temperatures around 90 °C; its optimal growth temperature is reported to be around 88 °C [209]. Furthermore, the structure of the 23S subunit exhibits an internal loop of size 33 closed by base pairs $1139 \cdot 1268$ and $1155 \cdot 1249$, cf. [77, 76].

Folding this sequence at 88 °C we did (almost) get the inner stem of this internal loop but the outer stem came apart as two single strands (cf. Figure 9.6(b)). When lowering the temperature to 75 °C we did get both stems, but the internal loop was split into two loops of size 2 and 27, respectively, by a short stem consisting of the base pairs $1141 \cdot 1266$ and $1142 \cdot 1265$ (cf. Figure 9.6(c)).

We then tried to search the range of temperatures between 75 °C and 88 °C, and at 82 °C we did in fact correctly predict the internal loop of size 33 (cf. Figure 9.6(d)). At this temperature we on the other hand missed the structure



(a) Fragment of the structure between bases 1112 and 1288.

(b) Prediction of the same fragment at 88 °C.

(c) Prediction of the same fragment at 75 °C.

(d) Prediction of the same fragment at 82 °C.

Figure 9.6: Known and Predicted structures for *thermococcus celer*.

inside the inner stem, a structure that is quite well predicted at 75 °C; no temperature thus seemed decisively best for predicting this structural fragment. Generally, as with the Qβ predictions, these predictions missed long-range base pairings and predicted structures consisting of fragments covering less than 300 bases.

It should however be mentioned that a prediction at 82 °C with a cutoff size of 30 completely misses the outer stem and thus makes a prediction of this fragment identical to the prediction at 88 °C. Thus we get a decisively better prediction at this temperature when examining internal loops of all sizes than when using a cutoff size of 30.

9.6 Conclusion

It is well known that heuristics may speed up the evaluation of internal loops in practice. One way to do this, is for all subsequences to keep track of the most stable structure of any of its subsequences. This is then used to cut off the evaluation of large loops closed by a specific base pair, when it is evident that they can not be more stable than the most stable structure closed by that base pair found so far.

As the method described in Section 9.3 actually evaluates the internal loops closed by a specific base pair in order of decreasing size, the above heuristic can not be combined with our method. We have instead implemented a heuristic based on determining upper bounds for the free energy of the optimal multibranched loop closed by some base pair. This heuristic unfortunately does not seem to have a positive effect for sequences shorter than 1000 nucleotides, as, for all but very long sequences, the time spent determining when to stop further evaluation exceeds the time that would have been spent evaluating the rest of the loops.

It would of course be more interesting to obtain further improvements on the worst-case behavior of the algorithm, possibly by applying some advanced search techniques similar to those described in [51]. This is not a straightforward task though, as our method has shifted the focus from the exterior (closing) base pair to the interior base pair of an internal loop. The same interior base pair might be optimal for several choices of exterior base pairs. Furthermore, the exterior base pair that yields the most stable substructure with a specific interior base pair might not even be one of them. Thus it is of no use just to search for the exterior base pair yielding the most stable substructure.

Our studies of structure predictions at high temperatures did not show an abundance of internal loops larger than the hitherto used cutoff size. There is thus no reason to suspect that predictions using this cutoff size are generally erroneous. We were however able to predict one internal loop that exceeds this size limit. Furthermore we predicted a number of internal loops with size larger than 20. This indicates that the cutoff size of 30 is probably a little bit to small for safe predictions at high temperatures. Especially if also suboptimal foldings, cf. [211], are sought for, or if calculating the partition functions as in [136], the cutoff size – if used at all – should be set somewhat higher.

Another observation is that the energy parameters estimated for higher temperatures by extrapolation of parameters experimentally determined at lower temperatures do not seem to allow for a prediction of the long range base pairings. One reason for this might be that structures at higher temperatures tend to have more unpaired bases in multibranched loops. The effect of the number of unpaired bases on the stability of multibranched loops should theoretically be logarithmic but are modeled by a linear function for reasons of computational efficiency. This might be acceptable for multibranched loops with only a few unpaired bases but becomes prohibitive as the number of unpaired bases grows.

Finally it should be mentioned that current methods for energy based RNA secondary structure prediction only consider structures that do not contain pseudo knots. Probably *the* open question of RNA secondary structure prediction is to put forth a model including pseudo knots that allows fair predictions within reasonable resources. Currently known methods suffer from either being too time- and space-consuming (time $\mathrm{O}(n^6)$ and space $\mathrm{O}(n^4)$ for the method presented in [167] and time $\mathrm{O}(n^5)$ and space $\mathrm{O}(n^3)$ for a restricted class of pseudo knots presented in [121]) or shifting the focus from stability of loops back to stability of pairs, cf. [178].

Acknowledgements

The authors would like to thank Darrin Stewart for his help with generating the figures for this article. This work was supported, in part, by NIGMS grant GM54250 to MZ.

Chapter 10

Protein Folding in the 2D HP Model

The paper *Protein Folding in the 2D HP Model* presented in this chapter has been published as a technical report [123] and a conference paper [124].

[123] R. B. Lyngsø, and C. N. S. Pedersen. Protein folding in the 2D HP model. Technical Report RS-99-16, BRICS, June 1999.

[124] R. B. Lyngsø and C. N. S. Pedersen. Protein folding in the 2D HP model. In *Proceedings of the 1st Journées Ouvertes: Biologie, Informatique et Mathématiques (JOBIM)*, 2000.

The conference paper and the technical report present work that to a large extend was done as a project for the graduate course *Structural Sequence Analysis* in the Spring of 1996 described in [122]. Except for minor typographical changes the content of this chapter is equal to the conference paper [124].

Protein Folding in the 2D HP Model

Rune B. Lyngsø[*] Christian N. S. Pedersen[†]

Abstract

We study folding algorithms in the two dimensional Hydrophobic-Hydrophilic model for protein structure formation. We consider three generalizations of the best known approximation algorithm. We show that two of the generalizations do not improve the worst case approximation ratio. The third generalization seems to be better. The analysis leads to an interesting combinatorial problem.

10.1 Introduction

Proteins are polymer chains of amino acids. An interesting feature of nature is that even though there are an infinite amount of amino acids, only twenty different amino acids are used in the formation of proteins. The amino acid sequence of a protein can thus be abstracted as a string over an alphabet of size twenty. In nature proteins are of course not one dimensional strings but fold into three dimensional structures. The three dimensional structure of a protein is not static, but vibrates around an equilibrium known as the *native state*. Famous experiments by Anfinsen *et al.* in [6] show that a protein in its natural environment folds into, i.e. vibrates around, a unique three dimensional structure, the *native conformation*, independent of the starting conformation. The native conformation of a protein plays an essential role in the functionality of the protein, and it is widely believed that the native conformation of a protein is determined by the amino acid sequence of the protein. As experimental determination of the native conformation is difficult and time consuming, much work has been done to predict the native conformation computationally.

To predict the structure of a protein computationally it is necessary to model protein structure formation in the real system, i.e. in the proteins natural environment. A model is relevant if it reflects some of the properties of protein structure formation in the real system. One obvious property could be *visual equivalence* between the native conformations in the model and the native conformations in the real system. Another more subtle, but useful property, could be *behavioral equivalence* between protein structure formation in the model and

[*] Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `rlyngsoe@daimi.au.dk`.

[†] Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: `cstorm@brics.dk`.

protein structure formation in the real system. As the laws of thermodynamics state that the native state of a protein is the state of least free energy, the real system is often modeled by a *free energy model* that specifies an energy function that assigns a free energy to every conformation in a set of legal conformations. The native conformation of a protein is then predicted to be a conformation that minimizes the energy function over the set of legal conformations.

The hydrophobic-hydrophilic model proposed by Dill in [44] is a free energy model that models the belief that a major contribution to the free energy of the native conformation of a protein is due to interactions between hydrophobic amino acids that tend to form a core in the spatial structure shielded from the surrounding solvent by hydrophilic amino acids. In the model the amino acid sequence of a protein is abstracted as a binary sequence of hydrophobic and hydrophilic amino acids. Even though some amino acids cannot be classified clearly as being either hydrophobic or hydrophilic, the model disregards this fact to achieve simplicity. The model is usually referred to as the HP model where H stands for hydrophobic and P stands for polar. The HP model is a *lattice model*, so called because the set of legal conformations is embeddings of the abstracted amino acid sequence in a lattice, in this case the two or three dimensional square lattice. In legal conformations amino acids that are adjacent in the sequence occupy adjacent grid points in the lattice, and no grid point in the lattice is occupied by more than one amino acid. Depending on the dimension of the square lattice we refer to the model as the 2D or 3D HP model. The free energy of a conformation depends on the number of non-adjacent hydrophobic amino acids that occupy adjacent grid points in the lattice. Figure 10.1 shows a conformation in the 2D HP model where 9 non-adjacent hydrophobic amino acids occupy adjacent grid points.

Despite the simplicity of the HP model, the folding process in the model have behavioral similarities with the folding process in the real system, and for most properties the 2D HP model has a behavior similar to the 3D HP model [45]. The HP model has been used by chemists to evaluate new hypothesis of protein structure formation [170]. The success of the HP model partly stems from the fact that the discrete set of legal conformations makes it possible to enumerate and consider all conformations of small proteins. Many attempts have been made to predict the native conformation, i.e. the conformation of lowest free energy, of a protein in the HP model [191, 207]. Most interestingly, the HP model was the first relevant model for protein folding for which approximation algorithms for the structure prediction problem, i.e. algorithms that find a conformation with free energy guaranteed close to the free energy of the native conformation, were formulated [80]. For a while it was believed that the structure prediction problem in the HP model would be solvable in polynomial time, but recently it was shown NP-complete [26, 40].

In this paper we describe three attempts to improve the best known approximation algorithm for the structure prediction problem in the 2D HP model [80]. We show that two generalizations of this algorithm, the U-fold algorithm and S-fold algorithm, do not improve on the best known 1/4 worst case approximation ratio, cf. Theorem 10.1, while the approximation ratio of the third generalization, the C-fold algorithm, seems to be better. We prove that the worst case approximation ratio of the C-fold algorithm is at most 1/3, cf. Theorem 10.2, and observe that it is closely related to an interesting combinatorial problem which we examine experimentally. Independently of our work Mauri *et al.* in [135] observe experimentally that the approximation ratio of an algorithm similar to our C-fold algorithm seems to be around 3/8, but they do not develop any worst case upper bound for the approximation ratio.

The rest of this paper is organized as follows. In Section 10.2 we formally describe the 2D HP model and bound the free energy of the native conformation of a protein in the model. In Section 10.3 we describe three attempts to improve the currently best approximation algorithm for the structure prediction problem in the 2D HP model. In Section 10.4 we describe and examine experimentally an interesting problem that is related to the approximation ratio of one of the approximation algorithms described in Section 10.3.

10.2 The 2D HP Model

In the 2D HP model a protein, i.e. an amino acid sequence, is abstracted as a string describing the hydrophobicity of each amino acid in the sequence. Throughout this paper we will use S to denote the abstraction of an amino acid sequence of length n, that is, S is a string of length n over the alphabet $\{0,1\}$ where $S[i]$, for $i = 1, 2, \ldots, n$, is 1 if the ith amino acid in the sequence is hydrophobic and 0 if it is hydrophilic. We will use the term "hydrophobic amino acid" to refer to a 1 at some position in S, and say that the parity of the 1 is even if its position in S is even, and odd if its position in S is odd.

A folding of a protein in the 2D HP model is an embedding of its abstraction S in the 2D square lattice such that adjacent characters in S occupy adjacent grid points in the lattice, and no grid point in the lattice is occupied by more than one character. We say that two 1's in S form a non-local 1-1 bond if they occupy adjacent grid points in the lattice but are not adjacent in S. Figure 10.1 shows a folding of the string 111010100101001001 in the 2D HP model with nine non-local 1-1 bonds. The free energy of a folding of S is the number of non-local 1-1 bonds in the folding multiplied by some constant $\epsilon < 0$. The free energy function models the belief that the driving force of protein structure formation is interactions between hydrophobic amino acids.

We say that the *score* of a folding of S is the number of non-local 1-1 bonds in it, and that the *optimal score* of a folding of S, $\mathrm{OPT}(S)$, is the maximum score of a folding of S. The simple energy function implies that the native conformation of a protein in 2D HP model is a folding of its abstraction with optimal score. The *structure prediction problem* in the 2D HP model is thus to find a folding of S in the 2D square lattice with optimal score. This problem has recently been shown to be NP-complete [26, 40], which makes it interesting to look for approximation algorithms that find a folding of S with score guaranteed to be some fraction of the optimal score of a folding of S. To issue such a guarantee for a folding algorithm, we need an upper bound on $\mathrm{OPT}(S)$. To derive an upper bound on $\mathrm{OPT}(S)$ we make two observations.

The first observation is that a hydrophobic amino acid can form at most two

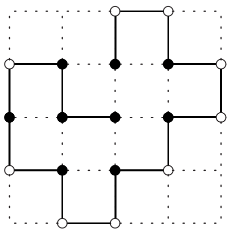


Figure 10.1: A conformation in the 2D HP model with 9 non-local 1-1 bonds.

non-local 1-1 bonds in the 2D square lattice except if it is the first or the last amino acid in the sequence, in which case it can form at most three non-local 1-1 bonds. The second observation is that two hydrophobic amino acids, $S[i]$ and $S[j]$, can occupy adjacent grid points in the 2D square lattice, i.e. form a non-local 1-1 bond, if and only if i is even j is odd or vice versa. If we define EVEN(S) as the set of even positions in S containing a hydrophobic amino acid, i.e. $\{i \mid i \text{ is even and } S[i] = 1\}$, and ODD($S$) as the set of odd positions in S containing a hydrophobic amino acid, i.e. $\{i \mid i \text{ is odd and } S[i] = 1\}$, then the two observations gives

$$\mathrm{OPT}(S) \leq 2 \cdot \min\{|\mathrm{EVEN}(S)|, |\mathrm{ODD}(S)|\} + 2. \tag{10.1}$$

This upper bound was first derived by Hart and Istrail in [80], who used it in the performance analysis of a simple folding algorithm that guarantees a folding with score 1/4 of the optimal score. This algorithm and various attempts to improve it is the topic of the next section.

10.3 The Folding Algorithms

A simple strategy for folding a string in the 2D square lattice is to find a suitable folding point that divides the string into two parts, a prefix and a suffix, that we fold against each other. This creates a “U” structure in which non-local 1-1 bonds can be formed between 1’s on opposite stems of the “U”. Loops protruding from the two stems of the “U” can be used to increase the number of non-local 1-1 bonds between the stems by contracting parts of the stems. We say that a folding created this way is a U-fold. Figure 10.2 shows a schematic U-fold and the left part of Figure 10.3 shows a U-fold of the string 1001001010010101000011 with four non-local 1-1 bonds between the stems and five non-local 1-1 bonds in total.

Hart and Istrail in [80] present a folding algorithm that computes a U-fold of S with a guaranteed number of non-local 1-1 bonds between the stems. By a simple argument they show that the folding point can always be chosen such that at least half of the 1’s with position in EVEN(S) are on one stem and at least half of the 1’s with position in ODD(S) are on the other stem. Since there is an odd number of characters between any two characters in S

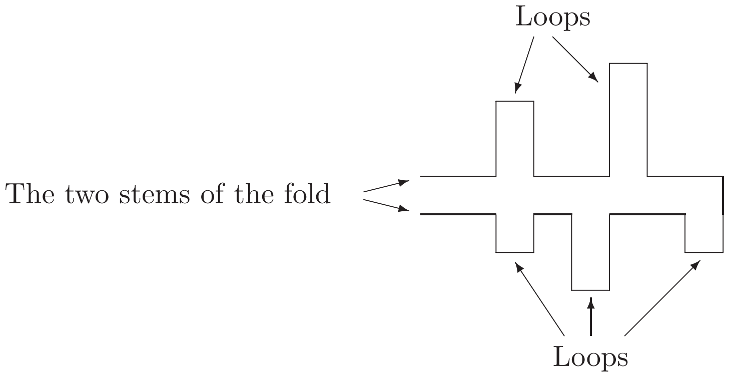


Figure 10.2: A schematic U-fold.

with positions in either EVEN(S) or ODD(S), loops can be used to contract each stem such that every second character on the contracted stem is a 1 with even or odd parity depending on the stem. As each contracted stem contains at least $\min\{|\text{EVEN}(S)|, |\text{ODD}(S)|\}/2$ 1's with equal parity placed in every second position along stem, the number of non-local 1-1 bonds between the stems of the created U-fold is at least $\min\{|\text{EVEN}(S)|, |\text{ODD}(S)|\}/2$, so except for a constant term the created U-fold scores at least 1/4 of the upper bound on OPT(S) given by (10.1). We say that the asymptotic approximation ratio of the algorithm is 1/4. By being a little bit more careful in the choice of folding point Hart and Istrail are able to formulate the folding algorithm such that the create a U-fold, for every string S, scores at least 1/4 of the upper bound on OPT(S). We say that the absolute approximation ratio of the algorithm is 1/4. The folding algorithm runs in time $O(n)$ where n is the length of S.

Our first attempt to improve the approximation ratio of the folding algorithm by Hart and Istrail, is to count all non-local 1-1 bonds between the two stems of the U-fold, and not only those where the 1's on each stem have equal parity. More precisely, we want to compute a U-fold of S with the maximum number of non-local 1-1 bonds between the stems, i.e. a U-fold of S with optimal score between the stems. Computing such a U-fold is not difficult. As illustrated in Figure 10.3, the trick is to observe that a U-fold of S, with folding point k, that maximizes the number of non-local 1-1 bonds between the stems, corresponds to the an alignment of the prefix $S[1 \mathinner{..} k-1]$ with the reversed suffix $S[k+2 \mathinner{..} n]^R$ that maximizes the number of matches between 1's, and allows gaps to be folded as loops.

Such an alignment corresponds to an optimal similarity alignment between $S[1 \mathinner{..} k-1]$ and $S[k+2 \mathinner{..} n]^R$, where a match between two 1's score 1, and all other matches and gaps score 0. To allow gaps to be folded out as loops, all gaps must have even length and between any two gaps in the same string there must be at least two matched characters. These additional rules on gaps can be enforced without increasing the running time of the alignment algorithm, so a U-fold of S with folding point k and optimal score between the stems can be computed in the time required to compute an optimal similarity alignment,

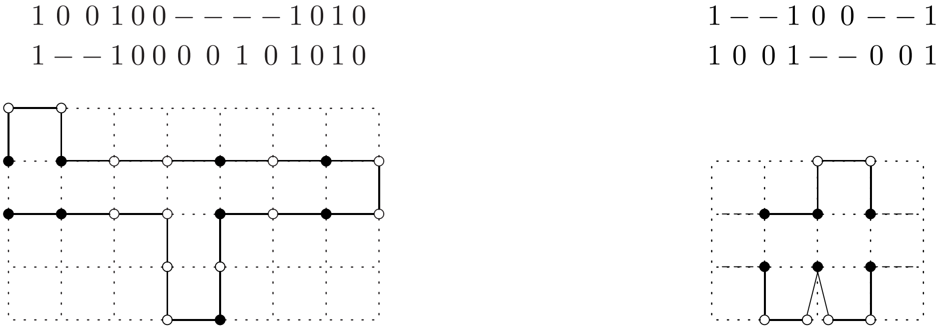


Figure 10.3: Left: Alignment of the prefix 1001001010 of the string 10010010100101010000011 with the rest of the string and the corresponding U-fold. Right: An example of an alignment with illegal gaps. The transformation to a folding implies that two loops protrude from the same element.

i.e. in time $O(n^2)$ where n is the length of S. By considering every folding point this immediately gives an algorithm, the U-fold algorithm, that computes a U-fold with optimal score between the stems in time $O(n^3)$. By observing that the best folding point k corresponds to an entry $(k-1, n-k-1)$ with maximum value in the alignment matrix resulting from an alignment of S and S^R with the above parameters, i.e. matches between 1's score 1, everything else score 0, and gaps have to be expressible as loops, we can reduce the running time of the U-fold algorithm to $O(n^2)$.

As the foldings considered by the folding algorithm by Hart and Istrail are a subset of the foldings considered by our U-fold algorithm, the approximation ratio of the U-fold algorithm is at least 1/4. Unfortunately it is no better in the worst case. As illustrated in Figure 10.4, this follows because any string of the form $(10)^i0(10)^i00(10)^i(01)^i$, $i > 0$, when folded as a U-fold with optimal score between the stems only scores 1/4 of the score of an optimal folding. The 1/4 approximation ratio of our U-fold algorithm and the folding algorithm by Hart and Istrail is thus tight. An obvious way to try to improve the approximation ratio of the U-fold algorithm would be to also count and maximize the number of non-local 1-1 bonds occurring between 1's on the loops. Unfortunately, as above, a set of strings can be constructed such that when folded this way they only score 1/4 of the score of an optimal fold.

Another way to try to improve the approximation ratio of the U-fold algorithm is to consider a larger set of foldings than U-folds. Figure 10.5 illustrates two ways to do this. The first way is to allow multiple bends of the string and loops on the outer stems. This gives rise to what we call S-folds. The second way is to allow two bends of the string that fold the two ends of the string towards each other and loops on the two stems. This gives rise to what we call C-folds. Both the S-fold and the C-fold with optimal score between the stems can be computed in time $O(n^3)$ using dynamic programming. For the C-fold it is easy to see how. A C-fold of S is a U-fold of a prefix, $S[1 \mathinner{..} k]$, and a U-fold of a suffix, $S[k+1 \mathinner{..} n]$, glued together to form a C-fold. As there are less than n

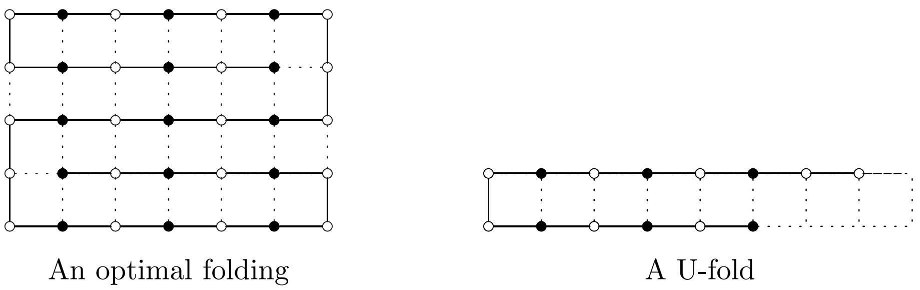
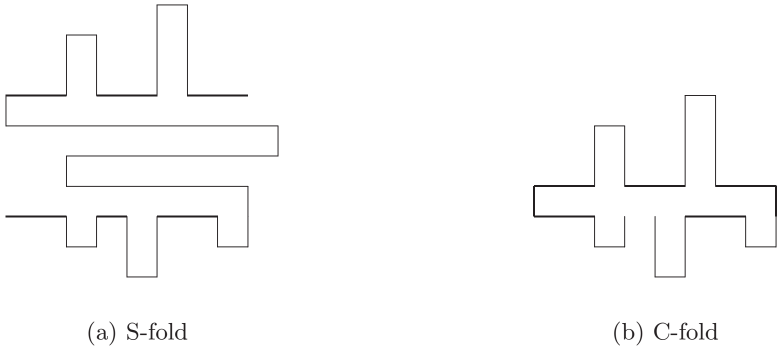


Figure 10.4: A string of the form $(10)^i 0(10)^i 00(10)^i 00(10)^i (01)^i$. For these strings the U-fold with optimal score between the stems is only 1/4 of the score of the optimal folding.


(a) S-fold

(b) C-fold

Figure 10.5: The S- and C-fold are two ways to generalize the U-fold.

ways to divide the string, the best C-fold can be found by computing and gluing together $2n$ U-folds. As each of these U-folds can be computed in time $O(n^2)$, the best C-fold can be computed in time $O(n^3)$. The computation of the best S-fold in time $O(n^3)$ is somewhat more technical. We choose to omit the details of the S-fold algorithm as it, as explained below, unfortunately turns out that its approximation ratio is no better than 1/4.

As S- and C-folds are supersets of U-folds, the approximation ratio of both the S- and C-fold algorithm is at least 1/4. Unfortunately this approximation ratio is tight for the S-fold algorithm because any string of the form $(10)^i(0^{2i+1}1)^{4i}(10)^i$, $i > 0$, when folded as a S-fold with optimal score between the stems only scores 1/4 of the score of an optimal folding. Similar to U-folds, we can show that counting and maximizing the number of non-local 1-1 bonds occurring between 1's on the loops of the S-fold does not improve the worst case approximation ratio of the folding algorithm. In contrast to U- and S-folds, we have not been able to find a set of strings that show that the 1/4 approximation ratio of the C-fold algorithm is tight. In fact experiments indicates, as explained in the next section, that the approximation ratio of the C-fold algorithm is somewhat better than 1/4. This is also observed in [135].

In our analysis of the approximation ratio of the C-fold algorithm we came

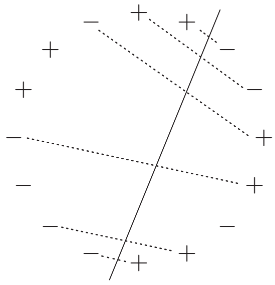


Figure 10.6: An example of a matching in a balanced string

up with a relation to an interesting matching problem. This is the topic of the next section. We end this section by summarizing the presented results.

Theorem 10.1 *The score of the best U- and S-fold of string S is at least, and at most in the worst case,* $1/4$ *of the score of an optimal fold of S. The score of the best C-fold of string S is at least* $1/4$ *of the score of the optimal fold of S.*

10.4 The Circle Problem

Let $P \in \{+,-\}^*$ be a string that contains equally many $+$'s and $-$'s. We say that P is a balanced string of length $n = |P|$. Consider P wrapped around the perimeter of a circle. A matching in P is obtained by dividing the circle by a line and connecting $+$'s with $-$'s using non-crossing lines that all intersect the dividing line. The size of the matching is the number of non-crossing lines connecting $+$'s with $-$'s that intersect the dividing line. Figure 10.6 shows an example of a matching of size 6. A maximum matching in P is a matching in P of maximum size. We use $M(P)$ to denote the size of a maximum matching in P and we use $M(n)$ to denote the minimum of $M(P)$ over all balanced strings P of length n, that is

$$M(n) = \min_{P:|P|=n} M(P).$$

The matching problem in balanced strings, or the circle problem as we call it, is closely related to the approximation ratio of our C-fold algorithm. To see the relation, we introduce the parity labelling of a string. The *parity labelling* of a string $S \in \{0,1\}^*$ is a string $P_S \in \{+,-\}^*$ in which the ith character indicates the parity of the ith 1 in S, e.g. the parity labelling of 100101110101 is $-++-+++$. A *balanced* parity labelling of S is a maximum length subsequence of P_S that contains equally many $+$'s and $-$'s. From the definition of EVEN(S) and ODD(S) follows that P_S contains $|\text{EVEN}(S)|$ $+$'s and $|\text{ODD}(S)|$ $-$'s, so a balanced parity labelling of S is obtained by removing $\big||\text{EVEN}(S)| - |\text{ODD}(S)|\big|$ $+$'s or $-$'s from P_S. The length of a balance parity labelling of S is $2 \cdot \min\{|\text{EVEN}(S)|, |\text{ODD}(S)|\}$, but the labelling is not unique as there can be several ways to choose the $+$'s or $-$'s to remove from P_S, e.g. the parity labelling $-++-+++$ gives $-++-$, $-+-+$ and $--++$ as possible balanced parity labellings.

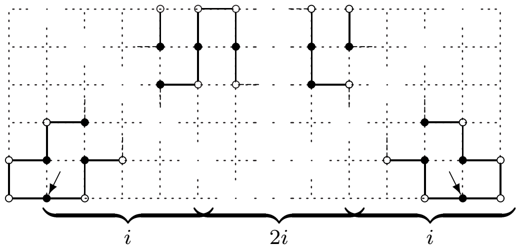


Figure 10.7: A folding of a string of the form $(01)^i000(01)^i(010)^{2i}(10)^i000(10)^i$. Only the two 1's indicated with arrows have less than the optimal two non-local bonds. The total number of non-local bonds in this folding is $2 \cdot \min\{|\mathrm{EVEN}(S)|, |\mathrm{ODD}(S)|\} - 1 = 4i - 1$ and thus, by the balanced parity labelling argument, the optimal score between the stems in a C-fold of this string is approximately 1/3 of the optimal score.

10.4.1 Upper Bounding the C-fold Approximation Ratio

To get the relation to C-folds, we observe that a C-fold of S with k non-local 1-1 bonds between the stems corresponds to a matching of size k in a balanced parity labelling of S. This implies that an upper bound on $M(n)$ is also an upper bound on the score between the stems of C-folds of strings with balanced parity labellings of length n. In other words, if $M(n) \leq \alpha n$ then the score between the stems of a C-fold of S is upper bounded by α multiplied by the length of a balanced parity labelling of S, i.e. upper bounded by $\alpha \cdot 2 \cdot \min\{|\mathrm{EVEN}(S)|, |\mathrm{ODD}(S)|\}$. Since the length of a balanced parity labelling of S is equal to the upper bound on $\mathrm{OPT}(S)$ given by (10.1), $M(n) \leq \alpha n$ implies that the approximation ratio of the C-fold algorithm, with respect to the upper bound on $\mathrm{OPT}(S)$ given by (10.1), is at most α.

It is easy to prove that $M(+^i\,-^i\,(+-)^i\,-^i\,+^i) = 2i + 1$ for any $i > 0$. Hence, $M(n) \leq n/3 + 1$, so the asymptotic approximation ratio of our C-fold algorithm is at most 1/3 if analyzed with respect to the upper bound on $\mathrm{OPT}(S)$ given by (10.1). Fortunately, as illustrated in Figure 10.7, for any $i > 0$ there exists a string $(01)^i000(01)^i(010)^{2i}(10)^i000(10)^i$ with balanced parity labelling $+^i\,-^i\,(+-)^i\,-^i\,+^i$ for which $\mathrm{OPT}(S)$ deviates from the upper bound of (10.1) by at most a constant term. This example proves the following theorem.

Theorem 10.2 *The score of the best C-fold of string S is at most in the worst case, 1/3 of the score of an optimal fold of S.*

10.4.2 Lower Bounding the C-fold Approximation Ratio

To use a matching in a balanced parity labelling of S to improve on the 1/4 approximation ratio of the C-fold algorithm, two requirements must be met. First, we need to be able to transform a matching in a balanced parity labelling of S into a C-fold of S with a number of non-local bonds proportional by some factor β to the size of the matching. Secondly, we need to lower bound the asymptotic ratio of $M(n)/n$ by some constant $\gamma > 1/(4 \cdot \beta)$. This would yield

$$\begin{array}{ccccccccccccc} + & & & - & & & + & & + & & & - \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & \cdots & 1 & 0 & 0 & 1 \\ 1 & - & - & 1 & - & - & 1 & & 1 & - & - & 1 \\ - & & & + & & & - & & - & & & + \end{array}$$

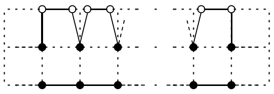


Figure 10.8: An example where the obvious transformation from a matching of a balanced parity labelling of a string to a C-fold, trying to place two 1's with connected labels opposite each other on the stems of a C-fold, fails.

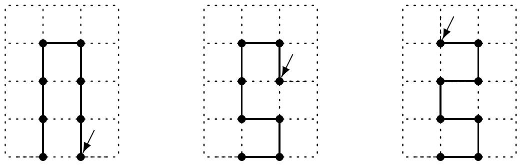


Figure 10.9: Possible hydrophobic loops of eight consecutive 1's. The positions of the embedding of the last 1 in the stretch is indicated with an arrow.

an asymptotic approximation ratio of the C-fold algorithm of $\beta \cdot \gamma > 1/4$. We have not yet solved these problems but will in the following report on some promising approaches and experiments.

The task of transforming a matching in a balanced parity labelling of S to a C-fold of S is not as straightforward as transforming the non-local bonds between the stems of a C-fold of S to a matching in one of the balanced parity labellings of S. Though one can identify the labels with 1's it will not always be the case that there is a legal C-fold of S where the non-local bonds between the stems corresponds to the connections between the corresponding labels in a matching in a balanced parity labelling of S.

To observe this, consider the two strings $S' = 1^{2i}$ and $S'' = (100)^{2i-1}1$, both with balanced parity labellings $P_{S'} = P_{S''} = (-+)^i$. Assume that S contains S' and S'' as substrings and that the labels of these two substrings have been connected with each other in the matching in a balanced parity labelling of S. As illustrated in Figure 10.8, we get the same problem as in the right-hand example in Figure 10.3 with two loops protruding from the same element if we try to make the obvious transformation of this matching to a C-fold of S. We observe that the obvious transformation only fails when we have stretches of consecutive 1's in one of the stems. One approach to solve the problem of transforming a matching in the balanced parity matching of S to a C-fold of S would thus be to 'eliminate' or at least 'shorten' consecutive stretches of 1's by removing 1's while ensuring compensatory non-local bonds.

This can be done in much the same way as when contracting the stems of a C-fold by folding out loops. As illustrated in Figure 10.9 we can fold out a stretch of an even number of consecutive 1's in a hydrophobic loop such that only two 1's remains along the stem. In such a loop where $2i$ 1's have been removed, i of which are at positions in EVEN(S) and i of which are at positions in ODD(S), there will be i non-local bonds. As long as $\beta \cdot \gamma \leq 1/2$ we can thus ensure compensatory non-local bonds. This allows us to remove the 1's that can be folded out in hydrophobic loops from S before finding a matching of a balanced parity labelling of the modified sequence.

Two problems still remain, though. First, the hydrophobic loops make the sequence less flexible since we cannot contract the stems immediately after a hydrophobic loop simply by folding out another loop. As indicated in Figure 10.9 we can however choose the position of the embedding of the last 1 in the stretch of 1's folded out rather freely which allows almost any contracting by an even number of amino acids immediately after a hydrophobic loop. Only when the loop removes $2i$ 1's with i odd there is a problem with contracting the stem by $i+1$ amino acids as we have to round the corner of the loop from the position furthest away from the stem where we can embed the last 1. Secondly, we have not eliminated stretches of consecutive 1's but merely limited them to being of length at most three. Though we find this approach promising we have not yet been able to carry through with the rigorous case-by-case analysis, an analysis that will require additional tricks besides the hydrophobic loops to handle special cases, of the various situations that can arise when trying to transform a matching of a balanced parity labelling of S to a C-fold of S.

To lower bound the asymptotic ratio of $M(n)/n$ it is easy to observe that $M(n) \geq n/4$. Unless we, unrealistically, hope to transform a matching in the balanced parity labelling of S into a C-fold of S with *more* non-local bonds than connections in the matching this lower bound does not say anything we do not already known, namely that the approximation of the C-fold algorithm is at least 1/4. Narrowing the gap between the trivial lower bound $M(n) \geq n/4$ and the upper bound $M(n) \leq n/3 + 1$ presented above has turned out to be a very difficult problem.

To get an impression of whether or not the trivial lower bound is tight, we did two experiments. First, we computed the value of $M(n)$ for all $n \leq 34$. As illustrated in Figure 10.10, this showed that $M(n) \geq n/3$ for all $n \leq 34$. Secondly, we computed $M(n)$ for a large number of randomly selected larger balanced strings. This random search did not produce a string in which the size of the maximum matching was less than $n/3$. Combined these two experiments lead us to believe that $M(n) \geq n/3$.

To help prove a non-trivial lower bound, one might consider the restricted matching problem where the dividing line must be chosen such that it divides the circle into two halfs. This restriction does not seem to affect the lower bound, as rerunning the experiment presented in Figure 10.10 gives the same results. It might also be helpful to consider other formulations of the problem. We observe that a dividing line in the circular representation of P corresponds to a partition XYZ of P, where the one side of the divided circle is Y and the other side is ZX. The maximum size of a matching in P given a partition XYZ is the length of the longest common subsequence of Y and $\overline{ZX}^R$, so

$$M(P) = \max_{XYZ\,:\,P=XYZ} |LCS(Y, \overline{ZX}^R)|.$$

In this terminology the above restriction of the problem, i.e. that the circle should be divided into two halfs, corresponds to only maximizing over partitions XYZ of P where $|Y| = |ZX|$. Another formulation of the problem follows from

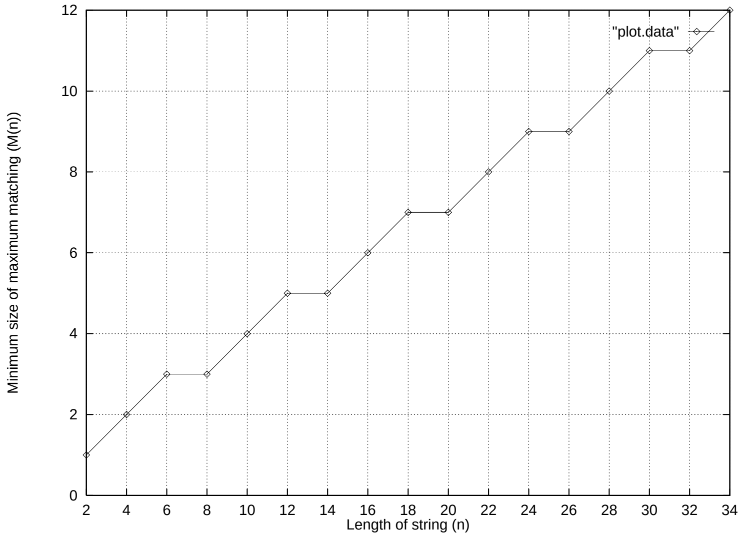


Figure 10.10: The minimum size of the maximum matching in balanced strings with length up to 34.

the observation that part of $LCS(Y, \overline{ZX}^R)$ is a subsequence of a prefix Y and $\overline{X}^R$ and the rest is a subsequence of the rest of Y and $\overline{Z}^R$. We can thus split Y according to this and reformulate the calculation of $M(P)$ as

$$M(P) = \max_{X_1,X_2} \{|X_1| + |X_2| \mid X_1\overline{X_1}^R X_2\overline{X_2}^R \text{ is a subsequence of P}\}.$$

This lends an immediate generalization of the problem as we can define

$$M_k(P) = \max_{X_1,\ldots,X_k} \{\sum_{i=1}^{k} |X_i| \mid X_1\overline{X_1}^R \ldots X_k\overline{X_k}^R \text{ is a subsequence of P}\},$$

where $M(P) = M_2(P)$ and $M_1(P)$ is the corresponding problem for the U-fold (equivalent to fixing one end-point of the dividing line in the circle formulation of the problem). One can observe that $M_k(n) < n/2$ for any k because the string $P = + + + - - + --$ gives that $M_k(P) = M_1(P) = 3$, but apart from this we have not been able to come up with any non-trivial bounds for $M_k(n)$.

10.5 Conclusion

We have presented three generalizations of the best known approximation algorithm for structure prediction in the 2D HP model. We have shown that two of these generalization do not improve the worst case approximation ratio, while the third generalization might be better. The future work is clear. First, prove that a matching in a balanced string can be transformed to a C-fold with score equal to the size of the matching. Secondly, prove or disprove that $M(n) \geq \alpha n$ for some $\alpha > 1/4$. Combined this would give whether or not our

C-fold algorithm improves the best known 1/4 approximation ratio for structure prediction in the 2D HP model. The experiments described in this paper make us conjecture that the described C-fold algorithm where non-local bonds in the loops are considered has an approximation ratio of 1/3.

Bibliography

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.

[2] A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, Massachusetts, 1974.

[3] S. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap cost. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

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

[5] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipmann. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[6] C. B. Anfinsen, E. Haber, and F. H. White. The kinetics of the formation of native ribonuclease during oxidation of the reduced polypetide domain. *Proceedings of the National Academy of Science of the USA*, 47:1309–1314, 1961.

[7] E. Anson and E. Myers. Algortihms for whole genome shotgun sequencing. In *Proceedings of the 3th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 1–9, 1999.

[8] A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *A selection of essays in honor of A. Ehrenfeucht*, volume 1261 of *Lecture Notes in Computer Science*. Springer, 1997.

[9] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119:247–265, 1993.

[10] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39:17–20, 1991.

[11] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.

[12] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15:481–494, 1996.

[13] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Daradzev. On economic construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194:487–488, 1970.

[14] L. Arvestad. Aligning coding DNA in the presence of frame-shift errors. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1264 of *Lecture Notes in Computer Science*, pages 180–190, 1997.

[15] K. Asai, S. Hayamizu, and K. Handa. Prediction of protein secondary strucuture by the hidden markov model. *Computer Applications in the Biosciences (CABIOS)*, 9:141–146, 1993.

[16] D. Bacon and W. Anderson. Multiple sequence alignment. *Journal of Molecular Biology*, 191:153–161, 1986.

[17] V. Bafna, E. L. Lawler, and P. A. Pevzner. Approximation algorithms for multiple sequence alignment. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science, pages 43–53, 1994.

[18] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 148–157, 1993.

[19] V. Bafna and P. A. Pevzner. Sorting by reversals: Genome rearrangements in plant organelles and evolutionary history of X chromosome. *Molecular Biology and Evolution*, 12:239–246, 1995.

[20] R. Bakis. Continuous speech word recognition via centisecond acoustic states. In *Proceedings of the ASA Meeting*, April 1976.

[21] P. Baldi, Y. Chauvin, T. Hunkapiller, and M. A. McClure. Hidden markov models of biological primary sequence information. In *Proceedings of the National Academy of Science of the USA*, volume 91, pages 1059–1063, 1994.

[22] C. Barrett, R. Hughey, and K. Karplus. Scoring hidden Markov models. *Computer Applications in the Biosciences (CABIOS)*, 13(2):191–199, 1997.

[23] S. A. Benner, M. A. Cohen, and G. H. Gonnet. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *Journal of Molecular Biology*, 229:1065–1082, 1993.

[24] G. Benson and L. Dong. Reconstructing the duplication history of a tandem repeat. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 44–53, 1999.

[25] S. A. Benson, M. N. Hall, and T. J. Silhavy. Genetic analysis of protein export in *Escherichia coli* K12. *Annual Review of Biochemistry*, 54:101–134, 1985.

[26] B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. *Journal of Computational Biology*, 5(1):27–40, 1998.

[27] P. Berman and S. Hannenhalli. Fast sorting by reversal. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *Lecture Notes in Computer Science*, pages 168–185, 1996.

[28] D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44:345–347, 1992.

[29] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. Technical Report RS-99-12, BRICS, April 1999.

[30] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149, 1999.

[31] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. *Journal of Discrete Algorithms*, 2000.

[32] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. Technical Report RS-99-25, BRICS, September 1999.

[33] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411, 2000.

[34] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[35] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB)*, 1997.

[36] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48:1073–1082, 1988.

[37] C. Chothia. One thousand faimilies for the molecular biologist. *Nature*, 357:543–544, 1992.

[38] G. A. Churchill. Stochastic models for heterogeneous DNA sequences. *Bulletin of Mathematical Biology*, 51:79–94, 1989.

[39] F. Collins and D. Galas. A new five-year plan for the U.S. Human Genome Project. *Science*, 262:43–46, 1993.

[40] P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. *Journal of Computational Biology*, 5(3):423–465, 1998.

[41] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[42] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[43] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[44] K. A. Dill. Theory for the folding and stability of globular proteins. *Biochemistry*, 24:1501, 1985.

[45] K. A. Dill, S. Bromberg, K. Yue, K. M. Fiebig, D. P. Yee, P. D. Thomas, and H. S. Chan. Principles of protein folding – a perspective from simple exact models. *Protein Science*, 4:561–602, 1995.

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

[47] S. Eddy. *HMMER User's Guide*. Department of Genetics, Washington University School of Medicine, 2.1.1 edition, December 1998. (See http://hmmer.wustl.edu).

[48] S. R. Eddy. Hidden markov models. *Current Opinion in Structurel Biology*, 6:361–365, 1996.

[49] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14:755–763, 1998.

[50] S. R. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22:2079–2088, 1994.

[51] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 488–496, 1988.

[52] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. *Theoretical Computer Science*, 64:107–118, 1989.

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

[54] D. Feng and R. Doolittle. Progressive sequence alignment as prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360, 1987.

[55] W. S. Fitch and T. F. Smith. Optimal sequence alignments. *Proceedings of the National Academy of Science of the USA*, 80:1382–1386, 1983.

[56] R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[57] A. S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 55(6):1199–1210, 1993.

[58] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82:112–120, 1998.

[59] A. S. Fraenkel and J. Simpson. The exact number of squares in fibonacci words. *Theoretical Computer Science*, 218(1):95–106, 1999.

[60] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

[61] M. L. Fredman. Algorithms for computing evolutionary similarity measures with length independent gap penalties. *Bulletin of Mathematical Biology*, 46:553–566, 1984.

[62] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.

[63] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[64] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[65] G. H. Gonnet, M. A. Cohen, and S. A. Benner. Exhaustive matching of the entire protein sequence database. *Science*, 256:1443–1445, 1992.

[66] J. Gorodkin, L. J. Heyer, and G. D. Stormo. Finding common sequence and structure motifs in a set of RNA sequences. In *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 120–123, 1997.

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

[68] O. Gotoh. Alignment of three biological sequences with an efficient traceback. *Journal of Theoretical Biology*, 121:327–337, 1986.

[69] O. Gotoh. Optimal alignment between groups of sequences and its application to multiple sequence alignment. *Computer Applications in the Biosciences (CABIOS)*, 9:361–370, 1993.

[70] M. Gribskov, R. Lüthy, and D. Eisenberg. Profile analysis. *Methods in Enzymology*, 183:146–159, 1990.

[71] M. Gribskov, A. D. McLachlan, and D. Eisenberg. Profile analysis: Detection of distantly related proteins. In *Proceedings of the National Academy of Science of the USA*, volume 84, pages 4355–4358, 1987.

[72] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[73] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993.

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

[75] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. Technical Report CSE-98-4, Department of Computer Science, UC Davis, 1998.

[76] R. R. Gutell. RNA secondary structures. Available via WWW from http://pundit.icmb.utexas.edu.

[77] R. R. Gutell, M. W. Gray, and M. N. Schnare. A compilation of large subunit (23S and 23S-like) ribosomal RNA structures. *Nucleic Acids Research*, 21:3055–3074, 1993. Database issue.

[78] S. Hannenhalli and P. Pevzner. Transforming cabbage into turnip (polynomial time algorithm for sorting signed permutations by reversals). In *Proceedings of the 10th Annual Symposium on Theory of Computing (STOC)*, pages 178–189, 1995.

[79] W. E. Hart and S. Istrail. Chrystallographical universal approximability: A complexity theory of protein folding algortihms on crystal lattices. Technical Report SAND 95-1294, Sandia National Laboratories, August 1995.

[80] W. E. Hart and S. Istrail. Fast protein folding in the hydrophobic-hydrophilic model within three-eights of optimal. *Journal of Computational Biology*, Spring 1996, 1996.

[81] W. E. Hart and S. Istrail. Invariant patterns in crystal lattices: Implications for protein folding algorithms. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *Lecture Notes in Computer Science*, pages 288–303, 1996.

[82] J. Hein. Unified approach to alignment and phylogenies. *Methods in Enzymology*, 183:626–645, 1990.

[83] J. Hein. An algorithm combining DNA and protein alignment. *Journal of Theoretical Biology*, 167:169–174, 1994.

[84] J. Hein and J. Støvlbæk. Genomic alignment. *Journal of Molecular Evolution*, 38:310–316, 1994.

[85] J. Hein and J. Støvlbæk. Combined DNA and protein alignment. *Methods in Enzymology*, 266:402–418, 1996.

[86] D. S. Hirschberg. A linear space algortihm for computing maximal common subseqeunce. *Communications of the ACM*, 18(6):341–343, 1975.

[87] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie*, 125:167–188, 1994.

[88] K. Hopkin. When RNA ruled – another lost world? *HMS Beagle, The BioMedNet Magazine*, 27, March 1998. Available via WWW from http://biomednet.com/hmsbeagle/1998/27/resnews/meeting.htm.

[89] Y. Hua, T. Jiang, and B. Wu. Aligning DNA sequences to minimize the change in protein. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 221–234, 1998.

[90] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[91] R. Hughey and A. Krogh. SAM : Sequence alignment and modeling software system. Technical Report UCSC-CRL-95-7, University of California, Santa Cruz, January 1995. (The SAM documentation is regularly updated. See http://www.cse.ucsc.edu/research/compbio/sam.html).

[92] R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *Computer Applications in the Biosciences (CABIOS)*, 12(2):95–107, 1996.

[93] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[94] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[95] C. S. Iliopoulos and L. Mouchard. Fast local covers. Technical Report TR 98-03, Department of Computer Science, King's College London, March 1998.

[96] C. S. Iliopoulos and L. Mouchard. Quasiperiodicity: From detection to normal forms. *Journal of Automata, Languages and Combinatorics*, 4(3):213–228, 1999.

[97] H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8:1–18, 1987.

[98] A. B. Jacobson. Secondary structure of coliphage Qβ RNA. *Journal of Molecular Biology*, 221:557–570, 1991.

[99] F. Jelinek. Continuous speech recognition by statistical methods. *Proceedings of the IEEE*, 64:532–536, April 1976.

[100] D. T. Jones, W. R. Taylor, and J. M. Thornton. A new approach to protein fold recognition. *Nature*, 358:86–89, 1992.

[101] S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science of the USA*, 85:841–845, 1988.

[102] J. D. Kececioglu and R. Ravi. Of mice and men: Algorithms for evolutionary distance between genomes with translocation. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms (SODA)*, pages 604–613, 1995.

[103] J. D. Kecicioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algoritmica*, 13:180–210, 1995.

[104] B. Knudsen and J. Hein. RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. *Bioinformatics*, 15:446–454, 1999.

[105] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison–Wesley, third edition, 1998.

[106] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[107] R. Kolpakov and G. Kucherov. Maximal repetitions in words or how to find all squares in linear time. Technical Report 98-R-227, LORIA, 1998.

[108] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 596–604, 1999.

[109] S. R. Kosaraju. Computation of squares in a string. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 807 of *Lecture Notes in Computer Science*, pages 146–150, 1994.

[110] A. Krogh. Two methods for improving performance of an HMM and their application for gene finding. In *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 179–186, 1997.

[111] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler. Hidden markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, 1994.

[112] J. B. Kruskal. An overview of sequence comparison. In D. Sankoff and J. B. Kruskal, editors, *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison*, chapter 1. Addison–Wesley, 1984.

[113] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 684 of *Lecture Notes in Computer Science*, pages 120–133, 1993.

[114] R. H. Lathrop. The protein threading problem with sequence amino acid interaction preferences is NP-complete. *Protein Engineering*, 7:1059–1068, 1994.

[115] R. H. Lathrop and T. F. Smith. Global optimum protein threading with gapped alignment and empirical pair score functions. *Journal of Molecular Biology*, 255:641–665, 1996.

[116] M.-Y. Leung, B. E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221:1367–1378, 1991.

[117] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 6:707–710, 1966.

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

[119] D. J. Lipman and W. R. Pearson. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Science of the USA*, 85:2444–2448, 1988.

[120] R. Luthy, J. U. Bowie, and D. Eisenberg. Assessment of protein models with three-dimensional profiles. *Nature*, 356:83–85, 1992.

[121] R. B. Lyngsø. Computational aspects of biological sequences and structures. Master's thesis, University of Aarhus, Department of Computer Science, DK-8000 Århus C, Denmark, June 1997.

[122] R. B. Lyngsø and C. N. S. Pedersen. Prediction of protein structures using simple exact models. Project in a Graduate Course. Available from http://www.daimi.au.dk/∼cstorm/papers, June 1996.

[123] R. B. Lyngsø and C. N. S. Pedersen. Protein folding in the 2D HP model. Technical Report RS-99-16, BRICS, June 1999.

[124] R. B. Lyngsø and C. N. S. Pedersen. Protein folding in the 2D HP model. In *Proceedings of the 1st Journées Ouvertes: Biologie, Informatique et Mathématiques (JOBIM)*, 2000.

[125] R. B. Lyngsø and C. N. S. Pedersen. Pseudoknots in RNA secondary structures. In *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 201–209, 2000.

[126] R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Measures on hidden Markov models. Technical Report RS-99-6, BRICS, April 1999.

[127] R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden Markov models. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 178–186, 1999.

[128] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999.

[129] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. An improved algorithm for RNA secondary structure prediction. Technical Report RS-99-15, BRICS, May 1999.

[130] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen. Internal loops in RNA secondary structure prediction. In *Proceedings of the 3th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 260–267, 1999.

[131] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.

[132] M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 271–278. Springer, Berlin, 1985.

[133] W. J. Masek and M. S. Paterson. A faster algorithm for computing string-edit distances. *Journal of Computing Systems Science*, 20:18–31, 1980.

[134] D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of Molecular Biology*, 288:911–940, 1999.

[135] G. Mauri, G. Pavesi, and A. Piccolboni. Approximation algorithms for protein folding prediction. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA)*, pages 945–946, 1999.

[136] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990.

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

[138] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1994.

[139] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

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

[141] D. Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the 5th Annual Symposium on Discrete Algorithms (SODA)*, pages 511–515, 1994.

[142] N. Morling. Retsgenetik. *Ugeskrift for Læger*, 161(21):3073–3076, May 1999. In Danish.

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

[144] R. K. Moyzis. The human telomere. *Scientific American*, pages 48–55, April 1991.

[145] M. Murate, J. Richardson, and J. Sussman. Simultaneous comparison of three protein sequences. *Proceedings of the National Academy of Science of the USA*, 82:3073–3077, 1985.

[146] E. W. Myers. A Four-Russian algorithm for regular expression pattern matching. Technical report, Department of Computer Science, University of Arizona, 1988.

[147] E. W. Myers. A Four-Russian algorithm for regular expression pattern matching. *Journal of the ACM*, 39:430–448, 1992.

[148] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences (CABIOS)*, 4:11–17, 1988.

[149] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid seqeunce of two proteins. *Journal of Molecular Biology*, 48:433–443, 1970.

[150] J. T. Ngo and J. Marks. Computational complexity of a problem in molecular-structure prediction. *Protein Engineering*, 5(4):313–321, 1992.

[151] J. T. Ngo, J. Marks, and M. Karplus. Computational complexity, protein structure prediction, and the levinthal paradox. In *The Protein folding problem and structure prediction*, pages 435–508. Birkhausen, 1994.

[152] H. Nielsen, S. Brunak, J. Engelbrecht, and G. von Heijne. Identification of prokaryotic and eukaryotic signal peptides and prediction of their cleavage sites. *Protein Engineering*, 10:1–6, 1997.

[153] H. Nielsen and A. Krogh. Prediction of signal peptides and signal anchors by a hidden Markov model. In *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 122–133, 1998.

[154] R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Science of the USA*, 77(11):6309–6313, 1980.

[155] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35:68–82, 1978.

[156] C. Papanicolaou, M. Gouy, and J. Ninio. An energy model that predicts the correct folding of both the tRNA and the 5S RNA molecules. *Nucleic Acids Research*, 12:31–44, 1984.

[157] M. Paterson and T. Przytycka. On the complexity of string folding. In *Proceedings of the 23th International Colloquium om Automata, Languages and Programming (ICALP)*, volume 1099 of *Lecture Notes in Computer Science*, 1996.

[158] C. N. S. Pedersen, R. Lyngsø, and J. Hein. Comparison of coding DNA. Technical Report RS-98-3, BRICS, January 1998.

[159] C. N. S. Pedersen, R. Lyngsø, and J. Hein. Comparison of coding DNA. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 153–173, 1998.

[160] J. T. Pedersen and J. Moult. Protein folding simulations with genetic algorithms and a detailed molecular description. *Journal of Molecular Biology*, 269:240–259, 1997.

[161] H. Peltola, H. Söderlund, and E. Ukkonen. Algorithms for the search of amino acid patterns in nucleic acid sequences. *Nucleic Acids Research*, 14(1):99–107, 1986.

[162] A. E. Peritz, R. Kierzek, N. Sugimoto, and D. H. Turner. Thermodynamic study of internal loops in oligoribonucleotides: symmetric loops are more stable than asymmetric loops. *Biochemistry*, 30:6428–6436, 1991.

[163] P. Pevzner. Multiple alignment, communication cost, and graph matching. *SIAM Journal of Applied Mathematics*, 52:1763–1779, 1992.

[164] N. Qian and T. J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

[165] M. Rabin. Discovering repetitions in strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 279–288. Springer, Berlin, 1985.

[166] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–286, 1989.

[167] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285:2053–2068, 1999.

[168] B. Rost. PHD predicting 1D protein structure by profile based neural networks. *Methods in Enzymology*, 266:525–539, 1996.

[169] M.-F. Sagot and E. W. Myers. Identifying satellites in nucleic acid sequences. In *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 234–242, 1998.

[170] A. Sali, E. Shahknovich, and M. Karplus. How does a protein fold? *Nature*, 369:248–251, 1994.

[171] D. Sankoff. Matching sequences under deletion/insertion constraints. In *Proceedings of the National Academy of Science of the USA*, volume 69, pages 4–6, 1972.

[172] D. Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, 45:810–825, 1985.

[173] P. H. Sellers. On the theory and computation of evolutionary distance. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974.

[174] M. J. Sippl. The calculation of conformational ensembles from potentials of mean force. An approach to knowledge based prediction of local structures in globular proteins. *Journal of Molecular Biology*, 213:859–883, 1990.

[175] E. L. L. Sonnhammer, G. von Heijne, and A. Krogh. A hidden Markov model for predicting transmembrane helices in protein sequences. In *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 1998.

[176] P. F. Stelling. *Application of Combinatorial Analysis to Repetitions in Strings, Phylogeny, and Parallel Multiplier Design.* PhD thesis, University of California, Davis, 1995.

[177] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152, 1998.

[178] J. E. Tabaska, R. B. Cary, H. N. Gabow, and G. D. Stormo. An RNA folding method capable of identifying pseudoknots and base triples. *Bioinformatics*, 14(8):691–699, 1998.

[179] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[180] W. R. Taylor. Identification of protein sequence homology by consessus template alignment. *Journal of Molecular Biology*, 188:233–258, 1986.

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

[182] J. L. Thorne, H. Kishino, and J. Felsenstein. Inching toward reality: An improved likelihood model of sequence evolution. *Journal of Molecular Evolution*, 34:3–16, 1992.

[183] A. Thue. Über unendliche Zeichenreihen. *Skrifter udgivet af Videnskabsselskabet i Christiania, Mathematisk- og Naturvidenskabeligklasse*, 7:1–22, 1906.

[184] A. Thue. Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivet af Videnskabsselskabet i Christiania, Mathematisk- og Naturvidenskabeligklasse*, 1:1–67, 1912.

[185] I. Tinoco, P. N. Borer, B. Dengler, M. D. Levine, O. C. Uhlenbeck, D. M. Crothers, and J. Gralla. Improved estimation of secondary structure in ribonucleic acids. *Nature New Biology*, 246:40–41, 1973.

[186] I. Tinoco, O. C. Uhlenbeck, and M. D. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971.

[187] N. Tran. An easy case of sorting by reversals. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1264 of *Lecture Notes in Computer Science*, 1997.

[188] D. H. Turner, N. Sugimoto, and S. M. Freier. RNA structure prediction. *Annual Review of Biophysics and Biophysical Chemistry*, 17:167–192, 1988.

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

[190] R. Unger and J. Moult. Finding the lowest free energy conformation of a protein is a NP-hard problem: Proof and implications. *Bulletin of Mathematical Biology*, 55(6):1183–1198, 1993.

[191] R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecular Biology*, 231:75–81, 1993.

[192] J. C. Venter, H. O. Smith, and L. Hood. A new strategy for genome sequencing. *Nature*, 381:364–366, 1996.

[193] G. von Heijne. Signal sequences. The limits of variation. *Journal of Molecular Biology*, 184:99–105, 1985.

[194] G. von Heijne and L. Abrahmsén. Species-specific variation in signal peptide design. *FEBS Letter*, 244:439–446, 1989.

[195] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[196] R. A. Wagner and M. J. Fisher. The string to string correction problem. *Journal of the ACM*, 21:168–173, 1974.

[197] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.

[198] M. S. Waterman. Efficient sequence alignment algorithms. *Journal of Theoretical Biology*, 108:333–337, 1984.

[199] M. S. Waterman and T. F. Smith. Rapid dynamic programming methods for RNA secondary structure. *Advances in Applied Mathematics*, 7:455–464, 1986.

[200] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequnece metrics. *Advances in Mathematics*, 20:367–387, 1976.

[201] J. D. Watson and F. H. C. Crick. A structure for Deoxyribose Nucleic Acid. *Nature*, 171:737, 1953.

[202] J. Weber and E. Myers. Human whole genome shotgun sequencing. *Genome Research*, 7:401–409, 1997.

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

[204] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[205] Y. Xu and E. C. Uberbacher. A polynomial-time algorithm for a class of protein threading problems. *Computer Applications in the Biosciences (CABIOS)*, 12(6):511–517, 1996.

[206] J. Yadgari, A. Amir, and R. Unger. Genetic algorithms for protein threading. In *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 193–202, 1998.

[207] K. Yue and K. A. Dill. Forces of tertiary structural organization in globular proteins. In *Proceedings of the National Academy of Science of the USA*, volume 92, pages 146–150, 1994.

[208] Z. Zhang, W. R. Pearson, and W. Miller. Aligning a DNA sequence with a protein sequence. *Journal of Computational Biology*, 4(3):339–349, 1997.

[209] W. Zillig, I. Holz, D. Janekovic, W. Schäfer, and W. D. Reiter. The archaebacterium *Thermococcus celer* represents, a novel genus within the thermophilic branch of archaebacteria. *Systematic and Applied Microbiology*, 4:88–94, 1983.

[210] M. Zuker. RNA web-page at http://www.ibc.wustl.edu/∼zuker/rna/-energy/node2.html.

[211] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244:48–52, 1989.

[212] M. Zuker and D. Sankoff. RNA secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46:591–621, 1984.

[213] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9:133–148, 1981.

[214] M. Zuker and D. H. Turner. The mfold server (version 3.0). Available at http://mfold2.wustl.edu/∼mfold/rna/form1.cgi.

Recent BRICS Dissertation Series Publications

DS-00-4 Christian N. S. Pedersen. *Algorithms in Computational Biology*. March 2000. PhD thesis. xii+210 pp.

DS-00-3 Theis Rauhe. *Complexity of Data Structures (Unrevised)*. March 2000. PhD thesis. xii+115 pp.

DS-00-2 Anders B. Sandholm. *Programming Languages: Design, Analysis, and Semantics*. February 2000. PhD thesis. xiv+233 pp.

DS-00-1 Thomas Troels Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow*. February 2000. PhD thesis. x+141 pp.

DS-99-1 Gian Luca Cattani. *Presheaf Models for Concurrency (Unrevised)*. April 1999. PhD thesis. xiv+255 pp.

DS-98-3 Kim Sunesen. *Reasoning about Reactive Systems*. December 1998. PhD thesis. xvi+204 pp.

DS-98-2 Søren B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. December 1998. PhD thesis. x+126 pp.

DS-98-1 Ole I. Hougaard. *The CLP(OIH) Language*. February 1998. PhD thesis. xii+187 pp.

DS-97-3 Thore Husfeldt. *Dynamic Computation*. December 1997. PhD thesis. 90 pp.

DS-97-2 Peter Ørbæk. *Trust and Dependence Analysis*. July 1997. PhD thesis. x+175 pp.

DS-97-1 Gerth Stølting Brodal. *Worst Case Efficient Data Structures*. January 1997. PhD thesis. x+121 pp.

DS-96-4 Torben Braüner. *An Axiomatic Approach to Adequacy*. November 1996. Ph.D. thesis. 168 pp.

DS-96-3 Lars Arge. *Efficient External-Memory Data Structures and Applications*. August 1996. Ph.D. thesis. xii+169 pp.

DS-96-2 Allan Cheng. *Reasoning About Concurrent Computational Systems*. August 1996. Ph.D. thesis. xiv+229 pp.

DS-96-1 Urban Engberg. *Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*. August 1996. Ph.D. thesis. xvi+222 pp.