Some String Problems in Computational Biology

by

J. Kevin Lanctot

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2000

0-612-51207-X

Abstract

This thesis introduces and analyzes a collection of string algorithms that are at the core of several biological problems.

First. it presents the Grammar Transform Analysis and Compression (GTAC) entropy estimator. the first entropy estimator for DNA sequences that has both proven properties and excellent entropy estimates. Additionally. the estimator uses a novel data structure to repeatedly solve the Longest Non-overlapping Pattern Problem in linear time. GTAC beats all known competitors in running time. in the low values of its entropy estimates. and in the number of properties that have been proven about it.

Second. it presents the Distinguishing String Problem. which has many biological applications such as creating diagnostic probes. universal primers. unbiased consensus sequences. and discovering potential drug targets. All these applications reduce to the task of finding a pattern that. with some error. occurs in one set of strings (the Closest String Problem and the Closest Substring Problem) and does not occur in another set (the Farthest String Problem and the Farthest Substring Problem). The NP-hardness of approximation properties of these problems are characterized. and approximation algorithms are presented.

Acknowledgements

I would like to thank my thesis supervisor. Prof. Ming Li. as well as several other professors who have assisted me along the way. including Prof. En-Hui Yang for letting me be involved in. and explaining many aspects of. Grammar Based Codes: Prof. Forbes Burkowski. Prof. Gordon Cormack. Prof. Paul Kearney. Prof. Frank Tompa. and Prof. Patricia Schulte here at the University of Waterloo. and Prof. Tao Jiang at the University of California at Riverside. and Prof. Todd Wareham at Memorial University for their discussions on Computational Biology. A special thank-you goes to Prof. Ian Munro who took the time to explain many of the complexities of both research and academia.

I'd also like to thank my co-authors on the papers. [33]. [34] and [35]. that were written as part of my work on this thesis. including Joe Wang. Bin Ma. Louxin Zhang. and also Ming Li and En-Hui Yang already mentioned above.

Many other fellow graduate students in the Department of Computer Science have also helped me out. including Christian Cumbaa. John Dickinson. Michael Easter. Michael Nidd. Riston Tapp. John Tsang. and Haoyong Zhang. Other Department of Computer Science personnel that have helped include Jonathan Badger. Martin Demaine. Brenda McBay. Jane Prime and Wendy Rush.

In the broader Computer Science Community. I would like to thank the SODA 1999 and SODA 2000 program committees and two anonymous reviewers for the journal Information and Computation for their help and suggestions.

Contents

List of Tables

List of Figures

Chapter 1

A Tale of Two Subjects

In the twentieth century. two areas of study have grown tremendously. molecular biology and computer science. In one sense these two areas are quite different: molecular biology is concerned with understanding natural processes: whereas computer science is concerned with solving problems using a machine. However. in another sense these areas are quite similar: they both deal with the processing of information. Moreover. throughout their history they have faced similar challenges.

1.1 A Brief History

While the idea that information is inherited through genes was developed by Mendel in the mid 1800s, it was not until 1902 that Sutton and Boveri noticed the parallel between Mendel's notion of genes and the action of chromosomes during cell division. They articulated this parallel in their Chromosomal Theory of Inheritance[1]. This theory proposed that chromosomes were the cellular structures behind Mendel's observations. Two years after these cellular components were recognized. the electronic components that would become essential elements in the first electronic computers began to be invented. First. Fleming patented the diode vacuum tube. followed by de Forest's modification to create a three-electrode version. Next. Eccles and Jordan developed the flip-flop electronic switching circuit. These simple elements were soon followed by larger components such as Shannon's electronic adder for binary encoded numbers and Stibitz's binary circuit that performed Boolean algebra.

1.1.1 The Big Picture Emerges

By the late 1930s and early 1940s both areas made major strides in seeing how the basic components fit together. In molecular biology. Beadle and Tatum related genes to proteins saying that one specified the structure of the other in what is known as the One Gene. One Enzyme Hypothesis. Later. Avery and others performed a series of experiments that suggested that DNA was the material in which the gene was encoded: the path between chromosomes. DNA. and protein had begun to emerge. In the computer world. the bigger picture was materializing as the various components of the previous era were combined to create the first electronic and electro-mechanical computers. These first computers were developed by researchers such as Atanasoff. Zuse. Aiken. and the team of Turing. Flowers. and

[1]The source of the molecular biology mentioned in this section comes from [10. 37] and the computer science information from [26].

Newman[2]. The developments in these two different areas actually crossed paths to a certain extent. Only a few years after Caspersson had shown that DNA was a very long molecule, Turing introduced the concept of a Turing machine which had as one of its components an infinitely long one-dimensional tape.

1.1.2 Information Storage is Considered

In the late 1940s, and throughout the 1950s, both areas began to research how information could be stored. In molecular biology, Watson and Crick proposed a double helix model for the structure of DNA, realizing not only how genetic information was stored, but also how it could be replicated as well. In computer science, von Neumann suggested a design for computational devices that not only stored data, but also had stored program control, creating a practical, but limited, implementation of a universal Turing machine. In terms of hardware, besides the invention of the transistor, various forms of bulk storage such as the magnetic drum memory, matrix core memory, and hard disk drives were developed.

1.1.3 Information Encoding is Investigated

In the 1950s and early 1960s both areas looked at how information was encoded. While the genetic code was being worked out by researchers such as Crick and Nirenberg, who looked at which sequences of DNA corresponded to which amino acids, Monod and Jacob were proposing a model of gene expression, to explain

[2]Note that this was M.A.H. Newman, not J. von Neumann the mathematician who introduced the idea of stored program control.

how it was regulated. In computer science. researchers were creating machine-independent ways of encoding algorithms by creating high level languages like the Short Order Code. Fortran. and Cobol. In another direction. the American National Standards Institute accepted ASCII and a standard for encoding English characters during this period.

1.1.4 The Interrelationship

As research continues today. computers are getting more complex. and biologists are understanding more of the complexity of life. Surprisingly. while both areas deal with information and computation. there has been very little interplay between them. Recently. however. the idea of genetic algorithms has been introduced into computer science. These algorithms model the way that information evolves in biological systems to solve computer science problems.

Another recent development. of particular importance. is the area of computational biology. This area of research uses the power of computers to solve complex problems in biology. Indeed. biology is currently facing a problem that computer programmers faced in the early 1950s. As the capacity of computers increased and the programs became more complex. it became more difficult for the programmers to work with the low level machine languages, so they developed higher level languages such as Fortran. Similarly, one of the key problems currently facing biologists, given the wealth of genetic information being generated by programs such as the Human-Genome Project, is how to understand all the low-level sequence data contained in genetic databases and put it into higher level constructs. The

The Electronic Computer	The Living Cell
Mass storage is on a tape or hard drive.	Mass storage is in DNA.
Information is stored in a magnetic medium.	Information is stored in a molecular structure.
Information is represented using the alphabet {0.1}.	Information is represented using the alphabet {a. c. g. t}.
The reliability is enhanced by storing redundant information.	The reliability is enhanced by also storing the reverse complement of the data.
Mass storage is broken up into a number of similar structures called platters.	Nuclear DNA is broken up into a number of similar structures called chromosomes.

Table 1.1: Analogy Between Cells and Computers–Mass Storage

challenge of using this information beneficially presents a new series of problems to be solved and the biological compounds at the center of this challenge—and also at the center of most living organisms—are DNA. RNA. and protein.

Since both organisms and computers process information. an excellent way to understand these classes of genetic compounds is through an extended analogy with computers.

1.2 DNA is Analogous to a Hard Drive

In a computer system. the largest and most permanent store of information is called the mass storage unit. which is typically a hard disk drive. As outlined in Table 1.1. analogously. in a living cell. the largest and most permanent store of genetic information (the traits of the organism that are inherited) is a class of chemicals called deoxyribonucleic acid or DNA.

1.2.1 Information Is Stored in a Molecular Structure

In a hard drive. a thin circular aluminum alloy surface called a platter acts as a support base on which the information is stored. This support base is coated with a thin layer of a hard metal alloy that is capable of being magnetized. Information is stored by creating magnetized patterns in one of two orientations. Using a method called flux reversal. a change in the magnetic pole direction represents a one. and a continuation of the same polarity represents a zero.

In living cells. long chains comprised of alternating molecules of deoxyribose (the D in DNA) and phosphate are chemically linked together to provide the support base. or backbone. on which the information is stored. DNA more closely resembles magnetic tape than a hard drive in the sense that the information is encoded along the length of the molecular backbone rather than in concentric circles.

Each deoxyribose in the backbone can have one of four different chemicals attached to it. namely adenine (a). guanine (g). cytosine (c) or thymine (t) and it is a sequence of these four different bases that encodes genetic information. Hence a DNA molecule can be thought of as a string in the alphabet $\{a. c. g. t\}$. If unwound. the strands of DNA in a single cell of a human being would be about 5 feet long and only 50 trillionths of an inch wide.

1.2.2 Reliability is Enhanced Using Reverse Complements

One way to enhance a system's reliability is to duplicate its information. Computers can store redundant information on separate drives. such as the Redundant Arrays of Independent Disks (RAID) family of techniques. A living cell also duplicates its information in the following manner. A DNA molecule has the property that if reversed. that is rotated 180 degrees. two slightly polar regions with opposite charges align nicely on the chemicals *a* and *t* to form a more stable structure. With *c* and *g*. three slightly polar regions align. Because of this property. *a* and *t* are called the complements of each other. and likewise *c* and *g* are complements. For example. the complement of *catgg* is *gtacc*. Furthermore. orientation is important for a DNA molecule. The beginning of a sequence is labeled 5'. and the end 3'. so given a sequence $5' - catgg - 3'$. the reverse complement is $5' - ccatg - 3'$. Here. the 3' and the 5' refer to carbons in the deoxyribose component of the nucleotide.

When two strands are exactly the reverse complement of each other. they can hybridize. or twist together. to form a structure that is more stable than just a single strand. In fact. cellular DNA is typically found as two strands. each the reverse complement of the other. twisted together to form a double helix. The deoxyribose and phosphate backbone forms the sides of the twisted ladder. and the *at* and *gc* base pairs form the rungs. As noted by Watson and Crick. this structure also suggests a mechanism for the replication of DNA. each existing strand acting as a template for a new strand.

1.2.3 Chromosomes are Subunits

While reverse complementation enhances reliability. there is also a need to increase capacity. The capacity of hard drives is increased by having several platters together on a single shaft. called a disk stack assembly. Likewise a cell can have several molecules of DNA, each called a chromosome, in the nucleus of the cell.

The Electronic Computer	The Living Cell
I/O is the process of transferring information between permanent storage and temporary storage.	Transcription is the process of transferring information from permanent storage to temporary storage.
Temporary storage uses electrical signals in RAM.	Temporary storage uses molecular structure in mRNA.
Information now represented as voltage levels. for example {0 v. -5 v}.	Information is now represented as the bases {a. c. g. u}.

Table 1.2: Analogy Between Cells and Computers–Temporary Storage

A strand of DNA is rather delicate. so breaking it up into many structures allows the amount of information to grow yet keeps each individual strand from growing past a certain size. Chromosomes can also be used to enhance reliability in another sense. Organisms that undergo sexual reproduction can contain two copies of each chromosome in each cell. one which came from the father and the other from the mother. Thus. redundancy enhances reliability.

Another parallel between the two subjects is that the information in both hard drives and in a human cell's chromosomes are a similar order of magnitude: the human genome contains roughly three billion base pairs of information. Cells and hard drives are different. however. in the sense that all the platters in a hard drive are exactly the same size. but chromosomes can differ in size. Another difference is that DNA is predominantly a WORM (write once. read many) storage device. whereas hard disks are write many. read many.

1.3 mRNA is Analogous to RAM

Table 1.2 illustrates that both computers and cells have to deal with the problem of getting information from unwieldy mass storage into a size more amenable to processing. Computers copy the information in fixed sized blocks from the hard disk into faster and transient storage. RAM. The cell transcribes a portion of the DNA of varying lengths into an intermediate compound called messenger ribonucleic acid (mRNA). For both computers and cells. the transient storage uses a different representation than the bulk storage. Computers use differing voltage levels in RAM rather than magnetic field. and cells use a different chemical. messenger ribonucleic acid (mRNA). The polymer mRNA is different from DNA in two ways: the backbone has been modified so that it is chemically less stable by adding an extra oxygen molecule to deoxyribose (the D in DNA) to create a new compound called ribose (the R in RNA). and there is a modification of one of the bases. thymine. creating a new chemical uracil (u). Uracil. like thymine. binds to adenine. So mRNA. like DNA. can be thought of as a string but in a slightly different alphabet $\{a. c. g. u\}$.

Because DNA and mRNA are similar molecules. DNA can act as a template for the creation of mRNA. The process of transcription. much like the process of replicating DNA, involves individual nucleotides lining up across from their corresponding reverse complements in the DNA template strand as the cell's transcription machinery moves along the growing strand polymerizing the new nucleotides into a growing mRNA chain. Another similarity that also involves quite a bit of extra machinery is the process of accessing the information in mass storage. For

The Electronic Computer	The Living Cell
A fundamental unit of programming is the function. which takes input and converts it to an output.	The basic functional units of the cell are proteins.
Functions are made up of a sequence of elementary instructions or statements.	Proteins are made up of a sequence of amino acids.
Voltage levels encode CPU instructions.	The mRNA sequence codes for amino acids.

Table 1.3: Analogy Between Cells and Computers–Functional Units

hard disks. reading process involves head stack assembly. rotary positioning assembly and read/write heads. For DNA. the process involves chemicals like RNA polymerase.

1.4 Protein is Analogous to the ALU

Table 1.3 outlines the analogy between functional units. The basic functional units of a computer are either the machine language commands into which the software is compiled. or the logic gates that implement these commands in the hardware. Both accept input. and generate output. The basic functional units of cells are proteins. but rather than operating on data. proteins operate on chemicals. combining them. modifying them. or breaking them down. However. proteins have a wider range of roles in organisms. Besides catalyzing chemical reactions in the cell. proteins make up a vital component of an organism's support structure. such as collagen. which provides the tensile strength of ligaments. tendons. and cartilage. Proteins can also facilitate communication. such as insulin which controls the absorption of sugar by cells of the human body.

1.4.1 Amino Acids are the Building Blocks

While a computer program is composed of a sequence of elementary statements. proteins are composed of a sequence of amino acids. This sequence of amino acids is specified by an mRNA sequence. An mRNA sequence in the alphabet $\{a, c, g, u\}$ is read three bases at a time. and this triplet. also called a codon. either specifies the next amino acid (out of 20 possible amino acids) to be added to the growing chain that will eventually comprise the protein or it specifies one of three stop codons which signifies the end of the protein sequence. The stop codons play the same role as the control-D character in UNIX. which signifies the end of a text string or text file. Although the protein is initially a linear structure. typically those that catalyze biochemical reactions fold up into a three dimensional globular structure. This process of using the information coded in mRNA to create a protein is called translation. Since the genetic code is based on triplets. this feature allows for the possibility of the cell translating in one of three different reading frames for any sequence. depending if the cell starts at positions one. two or three. Typically in protein coding regions. there are many stop codons occurring in the non-coding reading frames. but only one in the reading frame that codes for protein.

1.4.2 Genetic Sequences Exhibit Redundancy

Because there are 64 possible codons and only 20 amino acids. more than one codon can correspond to the same amino acid. Because of potential for more than one codon to specify the same amino acid, the genetic code is called redundant. and because of this redundancy, different sequences can code for the same protein.

When this multiple correspondence occurs. the synonymous codons usually differ in the third position (except for those amino acids which have six codons in which case they differ in both the third and either the first or second position).

Both genetic sequences and programming languages display two types of redundancy. First. different statements can mean exactly the same thing. In C++. the statements `++i;` `i+=1;` and `i=i+1;` are all synonymous for incrementing the value of *i*. just like the codons *uua*. *uug*. and *cuu* are all synonymous for leucine. Second. redundancy can also occur at a higher level. For example. the following two programs below produce the same output.

Program 1

```
#include<iostream.h>
main() {
   for ( int i=1; i<3; cout << i++);
}
```

Program 2

```
#include<stdio.h>
main() {
   long j=0;
   while (j<=1) {
      printf("%ld", j+=1);
   }
}
```

Likewise two proteins, such as human insulin and bovine insulin can be very similar in their action to stimulate the metabolism of carbohydrates, but have different amino acid sequences.

1.5 Genetic Sequences Have A Specific Format

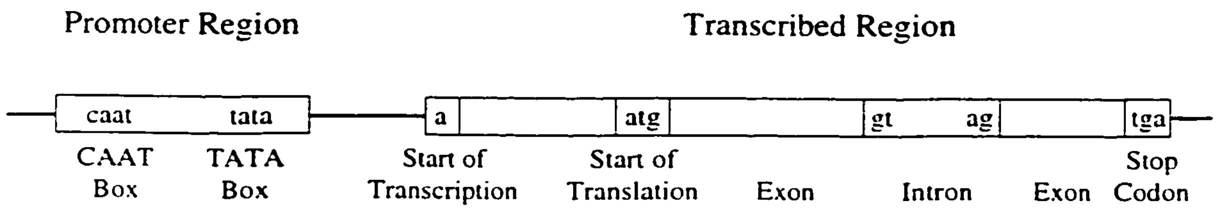

Figure 1.1: A Sample Format of Gene Coding Region

While the electronic computer and the living cell have many similarities in terms of information processing, they also have notable differences. In particular, how information is organized in DNA is less well-defined than that of a traditional computer file system. Figure 1.1 illustrates the format of one example of a protein coding region for a eukaryote.

In general terms, the region of DNA that codes for a protein, also called a gene, consists of two major sections:

1. The promoter region. This is the region of DNA which determines how much, in what time period, and in what tissue the protein is created. Some proteins are highly expressed; others may only be expressed at a certain stage of the organism's development, or under certain conditions. This region can be

thought of as a control region that determines when and how often a gene is expressed.

2. The transcribed region. This is the region of DNA which actually gets transcribed into mRNA.

1.5.1 The Promoter Region

Unfortunately. it is not easy to predict the location of the promoter region in relation to the transcribed region. It can be at varying distance and either upstream. downstream or actually in the transcribed region. As well. the actual sequences are short. and may differ slightly from each other. In the case illustrated in Figure 1.1. the promoter region occurs before. which is referred to as upstream from. the transcribed region. This region. in this case. consists of two major signals. namely the CAAT box and the TATA box.

1. CAAT box. The signal is named after the substring that most likely occurs at this position. This is the area of the promoter region that controls how likely transcription is to start. The more this region resembles the substring $5' - gg(t/c)caattct - 3'$. the more frequently the downstream region will be transcribed. Here (t/c) refers to the fact that either t or c can occur in the third position. Typical CAAT boxes will differ from the prototype. or consensus sequence. in one or two bases. The sequence usually occurs about 75 bases upstream of the start of transcription. but can vary in location by as much as a dozen bases.

2. TATA box. Like the CAAT box. this signal is named after the substring that most likely occurs here. It is the region just upstream from the start of transcription that indicates where transcription starts. This region is typically rich in a's and t's with the most likely sequence being $5' - tata(a/t)a(a/t) - 3'$ and it usually occurs about 25 bases upstream of the start of transcription.

1.5.2 The Transcribed Region

Like the promoter region. the transcribed region is currently not completely understood. For higher eukaryotes. such as animals and plants. there are typically four components.

1. Start of transcription. The first base to be transcribed is an a (or occasionally a g) surrounded on both sides by a c or a t. This region can be represented as $(c/t)(a/g)(c/t)$.

2. Start of translation. Occurring a short distance downstream from the start of transcription. the triplet atg marks the start of translation. which is the section of mRNA that encodes amino acids.

3. Exons. The exons are the part of the DNA region that gets transcribed and is eventually translated into a protein.

4. Introns. The introns. or intervening sequences. are the part of the transcribed region that does not get translated into protein. In fact. they are removed sometime between the end of transcription and the start of translation.

After the transcribed region is copied onto mRNA. the intron regions of the RNA are removed and the exon regions are spliced together so that they appear in the same order as on the DNA. The number of introns can vary from none to a few dozen. Exons can sometimes be mixed and matched in a certain sense. For example. a protein may exist in one of two variants: one consisting of exon 1 combined with exon 2. and another consisting of exon 1 combined with exon 3.

Chapter 2

Estimating DNA Sequence Entropy

2.1 The Challenge

Traditional Methods Perform Poorly

With the wealth of genetic information being generated. a key challenge is to use that information effectively. such as being able to distinguish introns from exons. One possible approach for identifying these regions is by characterizing their entropy. Since the alphabets of DNA and mRNA have four symbols each. the naive method of encoding them would take two bits per symbol. which would be the shortest possible encoding on average for a random sequence whose symbols were chosen uniformly and independently.[1] However, only a minuscule fraction of all the

[1]Here the term *random* is being used in the sense of Kolmogorov complexity: that is. a random string is an incompressible one.

possible DNA sequences result in a viable organism. therefore there are powerful constraints on the sequences which correspond to a living organism. These powerful constraints should be reflected in a lower entropy for the essential regions of DNA [2]. since not all sequences are possible. Indeed. Grumbach and Tahi [22] tried to compress a variety of DNA sequences. listed in Table 3.1. using some standard compression algorithms. The algorithms they tried included both static and adaptive Huffman. static and adaptive arithmetic coding of orders one, two. three. and four. and a variety of Lempel-Ziv based algorithms including LZ77 [73]. LZ78 [74]. LZSS [60]. LZW [65]. and the UNIX utility *compress*.

Surprisingly, they found that these standard algorithms performed rather poorly. generally worse than the naive algorithm. The Lempel-Ziv family of algorithms performed worse than the naive method. encoding the sequences between 2.07 to 2.79 bits per symbol. Their data for UNIX *compress* is presented in Table 3.1. The Huffman approaches also compressed poorly. encoding the sequences at between 2.08 to 2.24 bits per symbol. However. the arithmetic approaches generally encoded the sequences with less than two bits per symbol with the best performance given by dynamic arithmetic encoding of order two. with compression rates varying from 1.84 to 1.97 bits per base. With only 61 of the 64 possible mRNA triplet combinations being used to code for protein, that property alone suggests an encoding of 1.977 bits per symbol for exons, so the arithmetic encoding of order two provides only a slight improvement for the sequences that consist mostly of exons.

[2]The two definitions of entropy that are used in this thesis are presented in Equations 2.2 and 2.1

2.2 Previous Work

2.2.1 Biocompress

Recognizing the poor performance of traditional algorithms. Grumbach and Tahi sought to create custom algorithms that would perform better on genetic sequences. The principle that motivated their approach was the observation of the previous section: sometimes compression algorithms make the sequence larger because the overhead of the compression algorithm adds more than is saved. They created two algorithms. Biocompress [21] and Biocompress-2 [22]. which centred around the idea that compression would be used on a particular substring only if the compression plus the overhead of using that method performed better than the naive method. guaranteeing that the encoding would never take more than two bits per symbol.

Biocompress Combines Two Approaches

Biocompress is a combination of two approaches:

1. Literal encoding. where each symbol is coded as a two bit number.
2. Lempel-Ziv LZ77 style encoding. where a substring is encoded as a pair of integers. one representing the length of the match. and the second representing the position of the match somewhere to the left of the current position.

Biocompress checks to see which method is more efficient to encode a small portion of the input. First. it tries LZ77 compression which works as follows. Given a position i. find that longest substring starting at i that matches a previously occurring substring in the range 1 to $i-1$. Biocompress then compares the cost of encoding the string as a LZ77 phrase (start point and length) compared with using literal encoding and picks the shortest approach. Finally. Biocompress encodes the type of encoding. the length of the phrase. followed by the actual encoding.

Biocompress Recognizes Reverse Complements

Another feature that Biocompress has is that it not only recognizes a pattern that occurs twice. but it also recognizes patterns that appears both in the normal direction and as the reverse complement. So. Biocompress not only has to encode the LZ77 phrase as described above. but also has to encode whether it occurs in the forward direction or as a reverse complement. Grambach and Tahi found that a pattern and its reverse complement occur more often than would be expected by chance. which suggests that encoding this feature would pay off with better compression results.

Biocompress-2 Also Uses Arithmetic Encoding

Biocompress-2 is an extension of Biocompress that not only tries literal encoding and LZ77 style encoding. but also arithmetic encoding of order two. and then decides which of these three methods encodes the given phrase in the shortest space. The entropy estimates provided by Biocompress-2 are presented in Table 3.1. and in general Biocompress-2 compresses between two and seven percent better than Biocompress. So the extra compression gained by using three different methods compensates for the overhead.

2.2.2 Match Length Entropy Estimator

Farach *et al.* [16] developed a novel algorithm to estimate the entropy of DNA sequences called a match length entropy estimator. Unlike the Biocompress approach. the match length entropy estimator calculates the entropy directly rather than using the compressed size of the file (which includes some overhead) as an upper bound on the entropy.

Match Length Uses a Sliding Window

The match length entropy estimator works as follows. Letting L_i represent the match length at the i^{th} character in a sequence. the value of L_i is the length of the longest substring that occurs in two places: 1) starting at position $i+1$. and 2) anywhere in a sliding window consisting of the previous N_w characters (where N_w is a fix parameter describing the window size).

More formally. letting $x[i.j]$ represent the substring of x that starts at the i^{th} character and ends at the j^{th}. and using the relation $x \not\subseteq y$ to represent the condition x is not a substring of y. the match length at position i is defined as:

$$L_i = \min\{k : x[i+1. i+k+1] \not\subseteq x[i-N_w+1. i]\}.$$

For example. setting $N_w = 16$. with the 16 characters in the sliding window as $x[i-15. i] = \textit{tttagcatttcccgca}$. and the next five characters as $x[i+1. i+5] = \textit{catcc}$. since the first three characters of $x[i+1. i+5]$ are a substring of the sliding window. but the first four characters are not. this means $L_i = 3$. Letting $\overline{L}$ be the mean of all the L_i values, the entropy, H, can be estimated using the following equation:

$$H = \frac{\log_2 N_w}{\overline{L}}$$

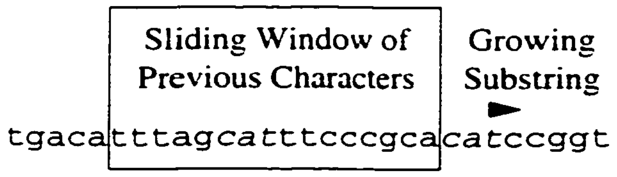
Sliding Window of Previous Characters
Growing Substring
tgacatttagcatttcccgcacatcccggt

Figure 2.1: The Match Length Entropy Estimator

Match Length's Convergence

Martin Farach *et al.* argue that their entropy estimator converges quicker than algorithms like LZ77, because while both algorithms perform the routine of growing a substring until there is a mismatch, LZ77 algorithms do this routine once per shortest unique substring in the input, whereas the match length estimator does this once per character. However, although they argue that their estimator converges quickly, they do not provide any standard datasets or comparison tests with other entropy estimators, so that the relative speed of their convergence can be compared with benchmark algorithms.

Match Length Results

There are a couple of curious features about their data. First, when the dataset is originally introduced (page 51), there are 659 introns, but in the results section, 579 introns are used in one experiment (page 54) and 574 are used in another (page 55). but the authors fail to mention any reason for not using all the intron data. Second. the authors failed to identify which sequences they used.

While the identity of their data is not clear. the motivation for their results is quite clear. Generally random changes in the sequence are thought to be more deleterious if they take place in an exon (which codes for protein) rather than in an intron (which gets removed before translation into protein). Hence these two regions should have different information-theoretic entropy. because the modifications have different consequences. Farach *et al.* used the match length estimator to distinguish introns from exons. In particular. they set the window size to 16. and ran their estimator on a dataset of 303 intron-exon pairs. To create an intron-exon pair. they used the i^{th} exon with the i^{th} intron from a complete coding sequence: that is. the exon and intron pair in their dataset occur adjacent to each other in the original DNA sequence. Using a signed rank test with these pairs. which compares the entropy of the exon to the intron to determine which was greater. they found that the average match length for introns was larger 73% of the time and that the variance of match length for introns was larger 80% of the time. As a result. they concluded that their findings support the hypothesis that the entropy of introns and exons are different with probability greater than 0.9999.

Finally. from a theoretical perspective. Farach *et al.* also proved that their algorithm was universal. that is. that their entropy estimate will approach the true entropy of the sequence as the length of the sequence increases assuming that the sequence is generated by a Markov process, which is when the probability of any future state is independent of past states and depends only on the present state.

2.2.3 CDNA

Loewenstern and Yianilos [41. 72] developed CDNA. a program that estimates the entropy of DNA sequences. The motivation for CDNA comes from the observation that naturally occurring DNA sequences contain many more near repeats then would be expected by chance. Here nearness is measured by Hamming distance. that is. the number of mismatches between two substrings of the same length. Two parameters that CDNA uses to capture the inexact matches are w. which represents the substring size. and h. which represents the Hamming distance. These parameters are used to create what they called a panel of predictive experts. $p_{w,h}$. each collecting information about the sequences of length w and allowing for h mismatches. CDNA then learns the weightings of these various experts. using Expectation Maximization. so that their ability to predict the next symbol in the sequence is maximized when combined into a single prediction.

CDNA's Approaches

CDNA has been implemented in two different ways. In the cross validation approach. the sequence is partitioned into 20 equal segments. the algorithm is trained on 19 of the segments and predicts the remaining segment. This approach is repeated 20 times using a different segment as the test segment each time. The value reported is the average of the 20 iterations.

While the cross validation estimate is used to remove the overhead associated with compression. a simple example will illustrate how this approach can severely underestimate the entropy of a genetic sequence. Let the input be a tandem repeat. say rr. where r is a random genetic sequence and hence has an entropy of two bits per symbol. The cross validation approach will report an entropy of close to zero. yet the entropy of rr is one bit per base. How likely is such an example? Both tandem repeats. such as the one above. and dispersed repeats are known to occur in DNA and comprise a substantial fraction of the human genome [37]. Since this method can severely underestimate the entropy. its results are not presented in Table 3.1.

The second method that Loewenstern and Yianilos use is called CDNA compress. in which the algorithm uses everything to the left of a nucleotide as the learning set to predict the next nucleotide's identity. The average entropy over all positions is calculated and this value is presented for various benchmark sequences in Table 3.1.

CDNA's Results

Since CDNA compress reports an entropy estimate. rather than the actual size of a compress file as Biocompress does. their results are. not surprisingly. generally better than Biocompress-2. Of the 12 standard sequences that Biocompress and Biocompress-2 used. CDNA was run on only ten of them. Loewenstern and Yianilos did not provide a reason why the other two sequences were omitted. Of the ten that Loewenstern and Yianilos did consider, CDNA compress beats Biocompress-2 on six of them. ties on one, and does worse on three test sequences. Two of the three that CDNA does worse on. namely CHNTXX and VACCG. and the two that

Loewenstern and Yianilos left out are known to contain long inverted repeats. The size of a single repeat in each of these sequences is between 5-15% of the size of the whole file. so dealing with them explicitly is probably what gives Biocompress-2 the advantage.

In addition to dealing poorly with inverted repeats. another disadvantage of CDNA compress is that its convergence properties have not been characterized. They do state how they define entropy. Given a DNA sequence x_t and a predictive model χ. the sequence entropy is as follows:

$$\lim_{t\to\infty} -\log_2 P(x_t|x_1.x_2.\dots x_{t-1}.\chi) \tag{2.1}$$

However. they do not prove that their algorithms are guaranteed to converge to this entropy estimate. In fact they state their purely compressive estimate probably overstates the source's entropy. and as pointed out above. their cross validation entropy estimate can severely underestimate the source entropy.

2.3 A Comparison

This dissertation proposes a novel entropy estimator of DNA sequences. Grammar Transform Analysis and Compression (GTAC). This entropy estimator is based on an approach developed by Kieffer and Yang [30] regarding the design and analysis of grammar based codes. and additionally. it recognizes the reverse complement property of DNA sequences. The GTAC entropy estimator is universal in the sense that it does not assume any source model and works for any individual sequence.

Moreover. GTAC is well justified from the information-theoretic point of view. Before presenting GTAC. and grammar based codes in general. its basic properties will be compared with the approaches of the previous sections.

In Table 2.1. GTAC is compared with the best known entropy estimators using three criteria:

- Universality: A code is universal with respect to any stationary source (that is. the source is homogeneous which means the statistics do not change over time) if the entropy estimate will converge to the actual entropy if the sequence is long enough. of being in any particular state does not change over time.

  A less general code. such as the Match Length entropy estimator. must make the additional assumption that the source is a Markov process for the code to be universal.

- Run time: A linear run time is important because genetic sequences can be quite long. millions or even billions of base pairs.

- Entropy Estimates: This is the goal of designing the algorithm.

Since compression algorithms such as UNIX *compress* and Biocompress-2 were designed as compression algorithms. they tend to overestimate the entropy for short sequences because they include overhead necessary for compression. so it is not surprising that they provide the worst entropy estimates. An example of overhead would be an online arithmetic encoder which initially assumes that each character occurs with equal probability. only to discover at the end of the sequence that this is not the case. CDNA was designed as an entropy estimator. but no convergence

Algorithm	Universal	Linear Run Time	Entropy Estimate
UNIX compress	yes	yes	worst
Match Length	limited	yes	-
Biocompress-2	yes	yes	3rd best
CDNA	no	no	2nd best
GTAC	yes	yes	best

Table 2.1: Features of Various Entropy Estimators

properties have been proven about its entropy estimates. As well. its run time is inferior to GTAC's. Overall. GTAC is good as or better than every other estimator for each factor.

2.4 Grammar Based Codes

A context-free grammar (CFG) is a quadruple $G = (V, T, P, S)$ where V is a finite non-empty set of variables. T is a finite non-empty set of terminal symbols that is disjoint from V. S is a distinguished element of V called the start symbol. and P is a set of production rules which map elements of V into $(V \cup T)^*$.

Context free grammars have been used in a number of data compression applications and their use can be classified into two broad approaches. In the first approach (for example [9]. [28]. [32]), the grammar between the compressor and decompressor is fixed. To compress an input, find a derivation of the input from the grammar and compress the derivation. The second approach (for example [52]. [60]) uses a different grammar for each string. The compressor creates a grammar that generates the string and the decompressor reconstructs the string by deriving it from the grammar.

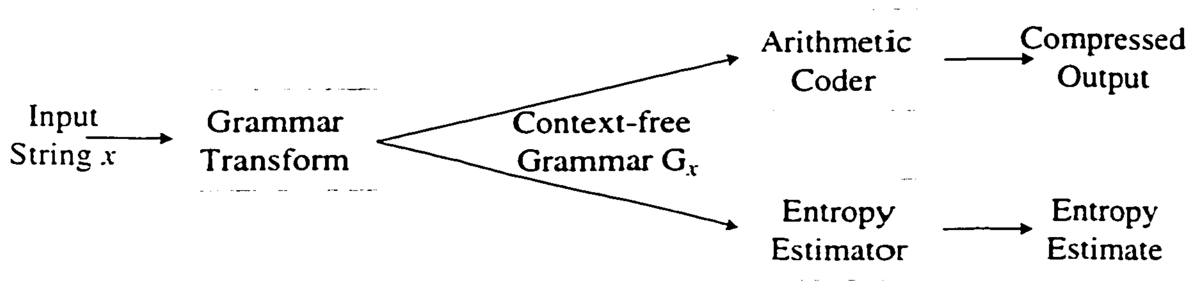

Figure 2.2: The Components of Grammar Based Codes

The Structure of Grammar Based Codes

A recent development of particular importance in this second approach is a class of CFG based compression algorithms developed by Kieffer and Yang [30] called Grammar Based Codes (GBC). This class of compression algorithms are both universal and lossless and provide a general framework for both practical compression of files. as well as directly reporting entropy estimates. Figure 2.2 illustrates the general components of the framework. First. the input string is converted to a context free grammar using a grammar transform. This is the key step. and much of the rest of the chapter will explain the ideas behind a grammar transform. Next. given a CFG. two different steps can be taken. First the CFG can be converted into a compressed file using an arithmetic encoder. or it can be used as a basis to calculate an entropy estimate.

2.4.1 Admissible Grammars

One of the goals of a grammar transform is to develop a grammar that represents a unique string; that is, given an input string x, create a grammar G_x such that G_x derives x and only x. One way of achieving this is for the CFG G_x to have an additional property called admissibility [30].

Definition 1 *A CFG,* $G_x = (V, T, P, S)$, *is admissible if it satisfies the following constraints:*

1. *Any variable in* V *appears only once on the left hand side of the production rules,* P.
2. *The empty string does not appear on the right hand side of any rule in* P.
3. G *has no useless rules, so when* x *is derived from* S *each rule is used at least once.*
4. *The language generated by* G_x *is nonempty.*

Example 1 *The following is an example of an admissible grammar* G_x *with* $x = tagcatacatacattag$.

$$\begin{aligned} S &\rightarrow BCCAB \\ A &\rightarrow cat \\ B &\rightarrow tag \\ C &\rightarrow Aa \end{aligned}$$

Admissible Grammars Have a Canonical Form

Admissible CFGs with terminal symbols contained in an alphabet $\mathcal{A}$ can be put in a canonical form referred to as $\mathcal{G}(\mathcal{A})$. by creating a set of variables $\{A_0. A_1. A_2. \ldots\}$ none of which are in $\mathcal{A}$ and rewriting the grammar G in the following format.

1. $V = \{A_0. A_1. A_2. \ldots. A_{|V|-1}\}$
2. The start symbol of G is A_0.
3. The variables make their first appearances in the order $A_0. A_1. A_2. \ldots. A_{|V|-1}$ when the following string is scanned left to right:

$$f^0(A_0) * f^1(A_0) * f^2(A_0) * \ldots * f^{|V|-1}(A_0).$$

Here the $*$ operator represents concatenation and the function $f(\ )$ represents the rewriting process. For example. $f^0(A_0) = A_0$. $f^1(A_0)$ represents rewriting each variable that appears in A_0 with its corresponding right hand side as specified in P. and $f^i(A_0)$ represents repeating this rewriting i times. Intuitively. the canonical form is when the variables are rewritten as $A_0. A_1. A_2. \ldots$ so that A_0 is the first variable to appear. A_1 is the second one to appear. and so on. as the start symbol is rewritten until it derives x.

Example 2 *Continuing on with Example 1. the grammar for x = tagcatacatacattag consisted of four non-terminals $\{S. A. B. C\}$ and the order that they first appear in the listing of the grammar is $\{S, B. C. A\}$. So setting S to*

A_0. B to A_1 and so on. the following grammar is created for G_x:

$$\begin{aligned} A_0 &\rightarrow A_1A_2A_2A_3A_1 \\ A_3 &\rightarrow cat \\ A_1 &\rightarrow tag \\ A_2 &\rightarrow A_3a \end{aligned}$$

Putting the rules in their order of first appearance results in the following:

$$\begin{aligned} A_0 &\rightarrow A_1A_2A_2A_3A_1 \\ A_1 &\rightarrow tag \\ A_2 &\rightarrow A_3a \\ A_3 &\rightarrow cat \end{aligned}$$

In the case of creating an actual compressed file. the right hand side of all the rules would then be concatenated together as follows:

$$A_1A_2A_2A_3A_1tagA_3acat$$

where an extra symbol (in this case. a space) is added as a separator and then an arithmetic encoder would create a file based on this input string.

2.4.2 Entropy Estimates

For genetic sequences, the issue that is of particular concern is not the actual compressed file but the entropy estimates that result from the Grammar Based

Code approach. In order to characterize the entropy. let $G_x = (V.T.P.S)$ be a CFG. Next. let $\omega(G_x)$ represent the string that is obtained from G_x in the following manner. First. concatenate together the right hand side of all the production rules. P. Second. remove the first occurrence of each of the non-terminals. V. Considering Example 2 above. this process would remove the first occurrence of $A_1.A_2.A_3$. resulting in the following:

$$\omega(G_x) = A_2A_1tagA_3acat$$

The entropy of G_x. or more specifically. the unnormalized entropy of the grammar G_x. is specified by the following equation

$$H(G_x) = -\sum_{s\epsilon(V\cup T)} n(s)\log_2\frac{n(s)}{|\omega(G_x)|} \tag{2.2}$$

where the convention that $0log0 = 0$ is followed. $|\omega(G_x)|$ represents the length of $\omega(G_x)$. and $n(s)$ represents the number of times that the symbol s occurs in the string $\omega(G_x)$. In the example above. $n(a) = 3$. $n(c) = 1$. and $n(A_0) = 0$.

Overhead and Entropy

One of the strengths of the Grammar Based Codes is the ability to separate out the unnormalized entropy estimate from the overhead involved in compressing the grammar. The theorem that achieves this result. which is from [30]. is as follows:

Theorem 1 *Let $G_x = (V. T. P. S)$ be an admissible grammar and let $|B(G_x)|$ represent the size of the string representing G_x. then*

$$|B(G_x)| \leq H(G_x) + 5|G_x| + |T| \tag{2.3}$$

where $|T|$ is the number of terminal symbols. and $|G_x|$ is the total size of the right hand sides of the rules in P. and $B(G_x)$ is a specific method for encoding all information about the grammar (including the number of terminals. number of nonterminals. a way to separate between each rule).

Since the entropy estimates are being used solely for DNA and mRNA sequences. $|T| = 4$. The above theorem is proved by construction. actually creating an encoding of the grammar and showing that the terminals. non-terminals and production rules can be encoded in a binary sequence whose length in bits is bounded by $H(G_x) + 5|G_x| + |T|$. Theorem 1 also suggests a way of breaking up the size of the codeword into the unnormalized entropy. $H(G_x)$. and the compressive overhead due to the encoding of the grammar's components. $5|G_x| + |T|$.

2.4.3 Irreducible Grammars

Inefficiency is Still a Problem

While Theorem 1 suggests how to separate out the grammar entropy from the overhead. it does not guarantee that either $5|G_x|$ or $H(G_x)$ is particularly small.

Consider the following example:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow B \\ B &\rightarrow C \\ C &\rightarrow aa \end{aligned}$$

It takes four non-terminals to represent two terminals. so the grammar is actually expanding the size of the sequence.

Irreducible Grammars Deal with Inefficiency

In order to deal with the possibility of inefficiency. some sort of restriction on the grammar is needed so that it does not expand unnecessarily. In order to achieve this end. Kieffer and Yang identified a subclass of admissible grammars called irreducible grammars [30]. Letting $R(P)$ represent the set of strings consisting of the right hand sides of all rules in P. the definition is as follows:

Definition 2 *An irreducible grammar is an admissible grammar* $G_x = (V. T. P. S)$ *with the following additional properties:*

1. *Every variable in* V *other than the start symbol is used more than once in* $R(P)$.
2. *There are no patterns in* $R(P)$. *Here. a pattern refers to a substring of length at least two that appears at least twice.*
3. *Each variable in* V *represents a distinct substring of the original input* x.

Essentially. an irreducible grammar ensures a measure of efficiency in dealing with patterns. The first property ensures that each variable represents a pattern. the second property ensures that every pattern has been removed. and the third property ensures that each represented pattern is unique. The admissible grammar presented in Examples 1 and 2 is also irreducible. In essence. an irreducible grammar does not restrict the CFG to only one particular algorithm but allows for a number of different approaches to creating a CFG. each of which must deal with patterns efficiently.

Irreducible Grammars and Compression

The main advantage of creating irreducible grammars is that they lead to efficient universal compression algorithms and entropy estimators. However. Theorem 1 only characterizes the length of the codeword that represents x. namely $B(G_x)$. in terms of the size of the grammar. G_x. rather than the size of the input. x. The approach that was taken by Kieffer and Yang in [30] was to provide a bound over $\mathcal{G}(x)$ where $\mathcal{G}(x)$ is the set of all irreducible grammars G_x representing x. The theorems they proved are as follows:

Theorem 2 *There is a constant c. which depends only on the cardinality of the alphabet A. such that for any sequence x. which is long enough.*

$$\max_{G_x \in \mathcal{G}(x)} |G_x| \leq \frac{c|x|}{\log |x|}$$

where $|x|$ denotes the length of x.

They also proved the following result:

Theorem 3 *For any stationary. ergodic source $\{X_i\}_{i=1}^{\infty}$ with entropy H. the quantity*

$$\max\left\{\left|\frac{|B(G_x)|}{n} - H\right| : G_x \in \mathcal{G}(X_1 \cdots X_n)\right\}$$

goes to 0 *with probability one as $n \to \infty$.*

A source is ergodic if the temporal statistics (statistics over time) is the same as the ensemble statistics (statistics over a certain position of occurrence). Since both theorems consider the maximum for all $G \in \mathcal{G}(x)$. they characterize the worst case performance of Irreducible Grammars. Clearly the results can only improve for particular implementations of an Irreducible Grammar or for particular stationary ergodic sources.

Justification of Grammar Entropy

Using the previous two theorems. and assuming x is generated by a stationary ergodic sequence. the use of $H(G_x)$ as an entropy estimator can now be justified. Recall that by Theorem 1. $|B(G_x)|$. the length of the codeword representing the input x. is bounded in the following manner:

$$|B(G_x)| \leq H(G_x) + 5|G_x| + |T|$$

Dividing each term by n and taking the limit as n. which is the length of x. approaches infinity. $|B(G_x)/n|$ approaches the entropy of the stationary. ergodic source of x and the $5|G_x|$ term approaches zero because of the bound on $|G_x|$

presented in Theorem 2. and $|T|$ is a constant. so the normalized entropy estimate $H(G_x)/n$ approaches the true entropy of the ergodic source. Hence the $5|G_x| + |T|$ represents overhead and including it generally causes an overestimate of the true entropy. so the normalized entropy for a grammar based code is simply defined as $H(G_x)/n$.

Optimality and Ergodic Processes

While $H(G_x)/n$ provides a good estimate of the sequence entropy there is no guarantee that it provides the best. Indeed. finding the irreducible grammar which results in the least grammar entropy estimate is probably NP-hard. so there is no guarantee that another irreducible grammar cannot perform better. Another point of hesitation is the nature of genetic sequences. They are not generated by stationary ergodic sequences. Certain regions. such as those that contain a lot of tandem repeats. are highly compressible and other regions are not. As well. viruses and other organisms can integrate their genetic material into the host DNA. changing the properties in regions where the DNA has been inserted. Additionally. highly expressed genes have codon bias. which is the property that among codons that code for the same amino acid. some occur more frequently that others. This bias is believed to be present because the compounds that carry the amino acids to the growing protein chain. tRNAs. are not available in equal numbers. so the codon bias adjusts to correspond to the most available tRNAs. Hence there are several mechanisms that upset the notion that DNA is generated by a stationary ergodic process.

Grammars Never Severely Underestimate

At this stage. it is not clear what the underlying model is for the generation of DNA sequences. One approach for dealing with this uncertainty is to show that whatever the model. with high probability. the irreducible grammar approach will never underestimate the actual entropy by a large amount. This result is achieved in the following theorem by Yang [30].

Theorem 4 *Letting $\{X_i\}_{i=1}^{\infty}$ be any data source. and letting $P(X^n)$ denote the probability of $X^n = X_1 \cdots X_n$. then for any constant $d > 0$. the following holds with probability at least $1 - n^{-d}$:*

$$\frac{|H(G_{X^n})|}{n} \geq -\frac{1}{n}\log P(X^n) - \frac{5G_{X^n} + |T|}{n} - \frac{d\log n}{n}$$

for any irreducible grammar transform G_x that represents x.

The term $(-\log P(X^n))/n$ can be interpreted as the information-theoretic entropy in bits per symbol of X^n and the $(5G_{X^n} + |T|)/n$ term is bounded by $O(1/\log n)$ from Theorem 2. Therefore the entropy estimate from a reducible grammar will never underestimate the actual entropy by a large amount.

2.4.4 Actual Implementations

Kieffer and Yang not only provided a theoretical framework for Grammar Based Codes. they also developed several practical implementations for both compression and entropy estimation. Among them are the Yang-Kieffer greedy sequential transform [70] which like Lempel-Ziv compressors moves along a sequence finding the longest match between the input stream and a growing dictionary. or grammar. of previously detected patterns. However. the Yang-Kieffer greedy sequential transform outperforms such Lempel-Ziv style of algorithms as UNIX *compress* and Gzip [70]. Another greedy approach that Kieffer and Yang developed is the longest matching substring grammar transform. and it is this algorithm that has been adapted for DNA sequences and for which I have developed a linear time algorithm.

Chapter 3

The GTAC Algorithm

3.1 The Grammar Transform Analysis and Compression Algorithm

The Grammar Transform Analysis and Compression (or GTAC) is an example of a Grammar Based Code. The core of GTAC is to repeatedly solve the longest non-overlapping pattern problem.

Definition 3 *The longest non-overlapping pattern (LNP) problem is as follows: Given a set of strings, $\mathcal{P}$, find the longest substring β such that β occurs in at least two non-overlapping positions somewhere in $\mathcal{P}$.*

The LNP problem can appear in the context of a grammar, $G = (V, T, P, S)$, by letting $\mathcal{P}$ be the set of all right hand sides of the production rules P, and adding the additional constraint that the length of β is at least two. GTAC's goal is to repeatedly find the LNP and reduce it, creating a new rule. To achieve this end, GTAC has two rewrite rules.

Definition 4 *Rewrite Rule 1: If an LNP β appears in the following form (both in the same right hand side of a rule) with $\alpha_1, \alpha_2, \alpha_3$ possibly equal to the empty string, and $|\beta| > 1$*

$$A \rightarrow \alpha_1\ \beta\ \alpha_2\ \beta\ \alpha_3$$

then rewrite the previous rule as two rules.

$$A \rightarrow \alpha_1\ B\ \alpha_2\ B\ \alpha_3$$

$$B \rightarrow \beta$$

Definition 5 *Rewrite Rule 2: If an LNP β appears in the following form (in the right hand side of two or more different rules) with $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ possibly equal to the empty string, and $|\beta| > 1$*

$$A \rightarrow \alpha_1\ \beta\ \alpha_2$$

$$B \rightarrow \alpha_3\ \beta\ \alpha_4$$

then rewrite the previous rules and introduce a new one, as follows.

$$A \rightarrow \alpha_1\ C\ \alpha_2$$

$$B \rightarrow \alpha_3\ C\ \alpha_4$$

$$C \rightarrow \beta$$

GTAC Optionally Recognizes Reverse Complements

GTAC can optionally recognize reverse complements. Recall from Section 1.2.2. that in DNA sequences, the symbols a and t are the complement of each other, and the symbols g and c are the complement each other. The following terminology will be used throughout this chapter. A string $\bar{\beta}^r$ is the reverse complement of β if $\bar{\beta}^r$ is the reverse of β with each character complemented. As Section 3.2 will make clear, the ability to detect reverse complements is an important feature of a DNA entropy estimator.

The GTAC algorithm deals with reverse complements by having two sets of non-terminals, regular non-terminals $A_1, A_2, \ldots$ and reverse complement non-terminals $R_1, R_2, \ldots$. These non-terminals come into play as follows. Given an input, x, the algorithm first creates the trivial grammar $S \rightarrow x$. Next, GTAC finds the LNP. If there are two or more occurrences of β, create a rule $A_i \rightarrow \beta$ ignoring any occurrences of $\bar{\beta}^r$. If there is only one occurrence of β and one of $\bar{\beta}^r$, then create a rule using one of the reverse complement non-terminals, $R_i \rightarrow \beta$, which means interpret the second occurrence of the non-terminal as the reverse complement of the rule.

An Example

For example, given the input x = *aatactgagtaaa*, GTAC first creates the trivial grammar.

$$S \rightarrow aatactgagtaaa$$

Next GTAC finds the largest LNP, which is *tact* and its reverse complement *agta*. GTAC reduces this substring using Rewrite Rule 1 and creates a new rule

$$S \rightarrow aaR_0gR_0aa$$

$$R_0 \rightarrow tact$$

Next, any remaining LNPs are rewritten. In this case there is one, *aa*.

$$S \rightarrow A_0R_0gR_0A_0$$

$$R_0 \rightarrow tact$$

$$A_0 \rightarrow aa$$

Next, the grammar is put in canonical form, which means relabeling the non-terminals in order of appearance with the reverse complement non-terminals starting with symbol R_0, and the normal non-terminals with symbol A_0.

$$A_0 \rightarrow A_1R_0gR_0A_1$$

$$R_0 \rightarrow tact$$

$$A_1 \rightarrow aa$$

When the last LNP of size two or more has been found, the rewriting process is complete. The right hand sides of the rules are concatenated together in the following order: the start rule is first, followed by any reverse complement rules, followed by any normal rules. The results, G_x, is as follows.

$$G_x = A_1 R_0 g R_0 A_1 tactaa$$

Next $\omega(G_x)$ is obtained by removing the first occurrence of each non-terminal.

$$\omega(G_x) = g R_0 A_1 tactaa$$

Finally, the entropy is calculated based on the frequency of appearance of each symbol using Equation 2.2.

Efficient Run-time is Important

Trivial implementations of this algorithm requires $O(n^3 \log n)$ time, where the size of the input, n, is often in the order of a million or more. For this size, even an $O(n^2)$ algorithm becomes intolerable. Since a key feature of the GTAC algorithm is to repeatedly look for the LNP, the generalized suffix tree (a suffix tree that contains more than one string [23]) is a natural data structure to consider because it can find the LNP in time that is linear in the total length of the right hand sides of the grammar. Given an input string x, a suffix tree that encodes x is a tree where each path from the root to a leaf represents a different suffix of x. Each interior node represents a substring that is in common with more than one suffix, hence it also represents a pattern (or repeated substring) in x. Finding the longest pattern is achieved merely by traversing the tree and finding the deepest interior node.

While suffix trees are a natural data structure for the LNP problem. the implementation is not straightforward. GTAC continually rewrites the rules. reducing the size of the grammar. so a key challenge is keeping the suffix tree up-to-date.

Original Input

tactag - - - *tag* ------ *cat* - *catact*

After creating the rule A $\longrightarrow$ tact

Aag *tag* ---- *cat* - - - *caA*

Figure 3.1: Destroying Patterns in a String

3.1.1 Keeping a Suffix Tree Up-To-Date is Challenging

Consider the example illustrated in Figure 3.1 where *tactag* is a substring in the input with *tag*. *cat*. and *catact* appearing elsewhere. When an LNP is discovered. in this case *tact*. replacing it with a non-terminal affects any patterns that overlap the substring *tact* anywhere else in the input. such as *cat* or *tag*. Generally. the patterns could be arbitrarily long. and so GTAC would have to check an arbitrary distance from each occurrence of the LNP. However. since GTAC is a greedy algorithm. working from the largest LNP to the smallest, when GTAC is rewriting an LNP of length l. then the longest pattern that can overlap the LNP is at most l symbols long. So in general. if an LNP occurs n times and is of length l. then when it is rewritten, at most $O(nl)$ other patterns may be affected.

Overlapping Patterns Can Cause Problems

Another complicating factor is that a generalized suffix tree directly reports the longest pattern (LP). but the algorithm requires the longest *non-overlapping* pattern. For example. if the string *aaaaaaaa* appears in the input at position k. the suffix tree will report it as two substrings of length seven occurring at positions k and $k+1$. whereas GTAC needs to interpret it as two substrings of length four. occurring at positions k and $k+4$. The following lemma gives a process for obtaining this result. and provides a bound for the size of the LNP given the LP.

Lemma 1 *If the length of the LP is l. then an LNP can be found with length at least $\lceil l/2 \rceil$.*

Proof. Although this lemma is almost obvious. the proof takes a bit of work. Let the reported LP start at positions k and $k+i$ and end at $k+l-1$ and $k+i+l-1$ respectively. In order to determine the locations of the corresponding LNPs. there are two cases: 1) $i \geq \lceil l/2 \rceil$ or 2) $i < \lceil l/2 \rceil$.

The first case is easy. Let the first LNP start at k and end at $k+i-1$. and let the second occurrence run from $k+i$ to $k+2i-1$. and both are at least $\lceil l/2 \rceil$ long.

In the second case. illustrated in Figure 3.2. since the two substrings are overlapping. the pattern is a periodic string; that is. the prefix from k to $k+i-1$ matches the substring from $k+i$ to $k+2i-1$, and also matches from $k+2i$ to $k+3i-1$. and so on. Hence the second LNP can start some suitable multiple of i characters after k. Since the two substrings are distinct. that is $i > 0$. there is a positive multiple of i. call it m. such that $(m+1)i \geq \lceil l/2 \rceil$ but $mi < \lceil l/2 \rceil$. This value is used to determine the start of the second LNP. The first LNP will

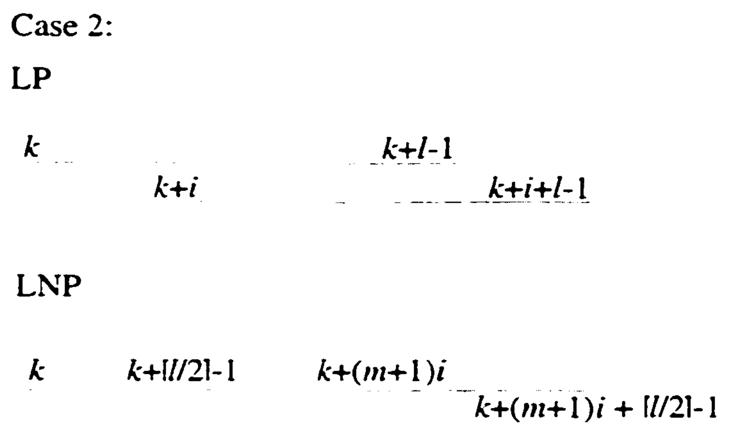

Figure 3.2: Dealing with Overlapping Patterns

run from k to $k + \lceil l/2 \rceil - 1$. and the second one will run from $k + (m + 1)i$ to $k + (m + 1)i + \lceil l/2 \rceil - 1$. It is clear that both substrings are $\lceil l/2 \rceil$ long. and that they do not overlap. so the only fact that needs to be verified is that the second LNP does not extend beyond the bounds of the original second LP. that is.

$$k + (m + 1)i + \lceil l/2 \rceil - 1 \leq k + i + l - 1.$$

The value of m was chosen so that $mi < \lceil l/2 \rceil$. Adding $\lceil l/2 \rceil$ and $k + i - 1$ to both sides of $mi \leq \lfloor l/2 \rfloor$ yields the required bound. □

The Greedy Approach Can Be Inefficient

The above greedy approach of dealing with overlapping substrings can create islands of terminals. For example. given a substring *acgtacgtacgta*. the above strategy would rewrite this portion of the grammar as the following.

$$A_i cgt A_i$$

$$A_i \rightarrow acgta$$

A second approach would be to rewrite the grammar as follows.

$$A_i A_i A_i a$$

$$A_i \rightarrow acgt$$

Perhaps the second approach, which does not take the longest pattern, is more desirable because a larger portion of the string has been captured by the non-terminals. Another difference is that the *cgt* in the center of the suffix in the first approach can only match other *cgt* substrings. This is because *acgta* is, at that point in time, the longest pattern, so there is no longer pattern that will include $A_i c$ or $t A_i$. In the second approach the a is available to be part of a larger substring to the right of a. In spite of these two observations, the first approach was chosen because it is consistent with the greedy approach of taking the LNP, as opposed to other greedy approaches, such as taking the pattern with the largest product of length and frequency.

While Lemma 1 characterizes the situation when the LP occurs twice, the next lemma is needed when the LP occurs three or more times.

Lemma 2 *Given an LP that occurs three times in a string, at least two of those* *occurrences do not overlap with each other.*

Proof. Proof by contradiction. Assume that any two of the three substrings overlap with each other. Without loss of generality let the three substrings start at k. $k+i$. and $k+j$ respectively with $0 < i < j < l$. where l is the length of the LP. Since the string that starts at k matches the string that starts at $k+i$. then the pattern is periodic with period of i. So. if any two symbols are i apart. they match. In particular the symbols that occur one past the end of the first and second LPs. at $k+l$ and $k+l+i$ respectively. match. This match contradicts the fact that these two substrings are the longest pattern. □

With these two lemmas. a subroutine for dealing with overlapping matches can be outlined. If an LP has just two occurrences. check for overlap. and if necessary create non-overlapping substrings. Given three or more matches. then at least two of those occurrences will be non-overlapping so the LP will also be an LNP.

3.1.2 Observations

The data structures for GTAC are a suffix tree. along with a copy of the original input. and an array of queues. In order to understand how they interrelate. a few observations are necessary first.

Observation 1 Since GTAC always considers the *longest* pattern at each iteration. if it is currently looking at an LNP of length l. the longest path in the tree is at most $2l$ long. reflecting the fact the tree may contain a length $2l$ LP that corresponds to a length l LNP. Hence the most number of symbols that the algorithm will have to search down the tree before uniquely identifying a suffix is $2l + 1$. because at that point it is guaranteed to be in a leaf. So if the LNP starts at position k. only suffixes in the range $[k - 2l, k + l - 1]$ need to be checked to see if they contain a pattern that overlaps the LNP.

Observation 2 No rule will contain a non-terminal from a previous iteration. For example. if a substring $\beta_1\beta_2\dots\beta_l$ is being replaced by a non-terminal A_i. then there will never be a pattern containing cA_i or A_ic. for some c. found later on in the grammar. because $\beta_1\beta_2\dots\beta_l$ was the longest pattern at that point of time. Since an LNP will never straddle over the beginning of non-terminal after it has been introduced. the suffix can be considered to end at that point in the suffix tree. For example. if a path from the root is $p_1p_2p_3\beta_1\beta_2\dots$. then that path could be edited as $p_1p_2p_3A_i$ or simply as $p_1p_2p_3\$$. where $\$$ is the end of file symbol. For convenience. this latter approach will be followed. However the rewrite from $\beta_1\beta_2\dots\beta_l$ to A_i has to be reflected somewhere. so the original input string is rewritten. rather than the suffix tree.

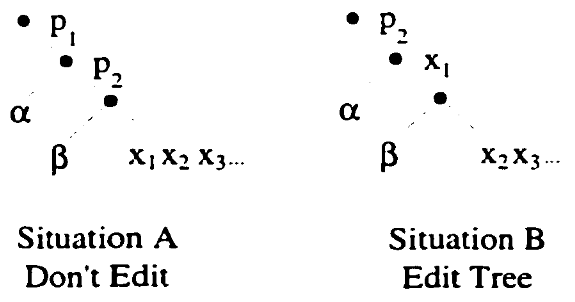

Figure 3.3: When GTAC Edits the Tree

Observation 3 Since GTAC is only concerned with repeating patterns. just the interior nodes and the first character of the leaves of the suffix tree need to be considered and kept up-to-date. A search need not continue beyond the first character of a leaf. Say for example. that GTAC is checking a suffix $p_1p_2\ldots$ to see if the pattern overlaps the current LNP. say $x_1x_2\ldots x_l$. If the situation is like Situation A in Figure 3.3 then there is no need to edit the tree since the substring p_1p_2 occurs twice. but the substring $p_1p_2x_1$ only occurs once. so rewriting the LNP. $x_1x_2\ldots x_l$ will not destroy the pattern p_1p_2. However in Situation B. the pattern $p_1p_2x_1$ does overlap the LNP. so when the LNP is removed from the tree. this branch will need updating.

3.1.3 A Linear Time Algorithm

With the above observations in mind. a more relaxed version of the generalized suffix tree can be used to implement the GTAC algorithm. In all. GTAC uses three data structures:

1. T - a generalized suffix tree. which provides information about the LPs:
2. Q - an array of doubly linked lists. which provides the data structure used to bucket sort the LNP information by length and remove items in constant time
3. x - an array holding the original input, which provides information about the LNP and the substrings that occur to the left of a given LNP.

First T is built from the original input, and then as it is traversed, information about the LNPs are stored in Q. Consequently, the LNPs are read from Q in descending order of size, and Q, x and T are kept up-to-date as the LNPs get removed. The whole algorithm is outlined and explained below. In the pseudo code below, suffix(y) refers to the path in the suffix tree that corresponds to the suffix y.

```
     GTAC(x) begin
1.     create rule: S → x;
       T = new suffix_tree(x);
       md = max_depth(T);
       Q[ ] = new array_of_lists(size = md);
2.     for each interior node n of T do
          if (n is an non-overlapping pattern)
             add n to Q[depth(n)];
       end for;
3.     for $l$ = md downTo 2 do
          while (Q[$l$] is non-empty) do
             n = next element of Q[$l$];
             B = new non_terminal;
4.           for each $\beta$[ ] = path to node n do
                p[ ] = 2$l$ chars to left of $\beta$[ ] in x;
                for i = 1 to 2$l$ do
```

```
                 if (suffix(p[i]) contains $\beta$[1] in T)
5.                  remove suffix in T after p[2$l$]:
              end for:
6.            for i = 1 to $l$ do
                 if (suffix($\beta$[i]) goes beyond $\beta$[$l$] in T)
                    remove suffix in T after $\beta$[$l$]:
              end for:
7.            replace $\beta$ with B in rules:
           end for:
8.         create rule: B $\rightarrow$ $\beta$:
        end while:
     end for:
  estimate entropy based on grammar:
  end alg:
```

1. **Initialize Data Structures:** Given an input string. x. create the trivial rule $S \rightarrow x$. a generalized suffix tree T. and an array of lists Q. with $Q[i]$ representing a list of LNPs of size i.

2. **Fill Q:** Traverse the tree, T. keeping track of the depth. At each interior node n. which represents a pattern. check to see if it also represents an non-overlapping pattern and if so create a pointer to that node. and add it to the list that corresponds to its depth. namely $Q[depth(n)]$. Also include a back-pointer from the interior node to its corresponding entry in the list. and

also include the positions where this match occurs in the string x.

3. **Get $p\beta s$:** Work through the Q array starting at the largest value md. Given an LNP. say β. from $Q[l]$. for each occurrence of β. consider the substring that extends up to $2l$ characters on either side of β. namely $p_1p_2\ldots p_{2l}\beta_1\beta_2\ldots\beta_l s_1 s_2\ldots s_{2l}$ where p represents the prefix. and s the suffix of β. This substring can be determined by consulting T to get the list of locations of β. and then consulting the input string x to get p and s for that occurrence.

4. **Find Suffixes:** For each suffix starting at $p_1p_2\ldots$ and ending at $\beta_l s_1 s_2\ldots$. perform Steps 4 to 7. Descend down the tree on the path $p_1p_2\ldots$ with two pointers. d-ptr and i-ptr. The d-ptr will point to the leaf that corresponds to the unique suffix that GTAC is seeking and will eventually delete: the i-ptr will point to the node where GTAC inserts the end-of-string marker (which is always the node between p_{2l-1} and β_1). Search down this path for the beginning of the LNP. β_1. Consistent with Observation 3 above. a search will only go as far as the first character in the leaf.

5. **Remove Suffixes Starting in p:** If. while searching down a path. β_1 is encountered. then the algorithm will begin modifying the tree. First the i-ptr stays at the node between p_{2l-1} and β_1. A node may have to be created here and a corresponding suffix link and entry made in the Q array. The d-ptr continues down the path to the leaf corresponding to this unique suffix. If this leaf has more than one sibling. then delete the leaf that d-ptr points to.

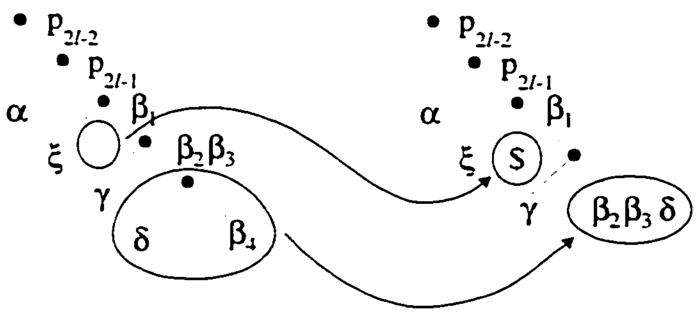

Figure 3.4: Removing a Suffix

If the parent node has only two child leaves. then delete the two leaves. and convert the parent node to a leaf corresponding to the sibling of the leaf that is being deleted. For example. as illustrated in Figure 3.4. check this if the path into the parent node is labeled $\beta_2\beta_3$ and the two leaves are labeled $\beta_4 \ldots$ and δ then the parent node becomes the leaf corresponding to the suffix $\beta_2\beta_3\delta$. Wherever a node is deleted. the back-pointer from this node is followed. if it exists. and its corresponding entry in the Q array is removed. As well. the end-of-string marker. \$. is added to where the i-ptr points to (representing the new place where this suffix ends. as explained in Observation 2). When finished with the current suffix and moving to the next one. suffix links are used. A suffix link is a pointer from the interior node that represents the suffix starting with $c\alpha$ to the one representing the suffix starting with α. where c is a single character and α is a possibly empty string. Both the i-ptr and the d-ptr independently take the suffix links from their current node to move on to the next suffix, or go up at most one node. and use that suffix link instead.

6. **Remove Suffixes Starting in β:** A similar process is followed for the suffixes that start in β. except that the entire suffix is eliminated with no need to insert an end-of-string marker anywhere. If the path is $\beta_i\beta_{i+1}\ldots\beta_l s_1 s_2\ldots s_j$. then the leaf corresponding to that suffix is eliminated from the tree entirely. and if necessary the parent node of that leaf becomes a leaf. and the corresponding entry in Q eliminated. The final suffix to be considered is $\beta_l s_1 s_2\ldots$.

7. **Edit Rules:** Finally. the rule containing β is updated by deleting that occurrence of β and adding the appropriate non-terminal in its place.

8. **Create New Rule:** A new rule is created and the right hand side of that rule. β. is added to the suffix tree (actually the last occurrence of β in the tree is converted to this new entry).

With only a few modifications. this algorithm also deals with reverse complements. In the first step both the string x and $\bar{x}^r$ are added to the suffix tree. and when removing a suffix both the forward and the reverse complement occurrences must be removed. As well. two sets of non-terminals are used. one to represent patterns that occur left to right and one to represent patterns that occur as the reverse complement. The decision about when to include a reverse complement non-terminal takes place between Steps 3 and 4.

With the description of the algorithm complete. the next step is to characterize the running time. which is as follows.

Theorem 5 *The GTAC entropy estimator runs in time linear in the size of its input.*

Proof. Assume that the input has m characters in it. In step 1. each statement can be performed in linear time. such as building the suffix tree [63]. Step 2 is to build and maintain Q. the array of lists. There are two aspects: the size of the array. and the number of elements in a list. The size of the array is at most $m/2$. The number of entries is bounded by the number of interior nodes in T. because every interior corresponds to a repeated pattern. Since each interior node in T has between two and five children. and a suffix tree built from a string of m characters has m leaves. then. picking the worst case. where each node has only two children. T will have at most m nodes. Placing and traversing at most m entries in at most $m/2$ lists can be done in linear time.

Steps 3-7 of the algorithm are removing an LNP and taking care of all the patterns that overlap the LNP. For this situation. there is the following lemma.

Lemma 3 *Given an occurrence of an LNP of length l. GTAC removes all possible patterns that overlap that occurrence in time $O(l)$.*

Proof. GTAC removes all the overlapping substrings for a given occurrence of an LNP in steps 4 to 7. During these steps. the i-ptr and d-ptr go down a few nodes in the tree. a leaf is possibly deleted. an internal node is converted to a leaf. and an entry in a list is deleted. The i and d pointers may go up a node before following a suffix link. and then begin dealing with the next suffix. Of these operations. the only one that may take more than constant time is when the i-ptr and d-ptr go down the tree a few nodes.

The following is an argument for the d-ptr. with the argument for the i-ptr being similar. While the d-ptr can travel as much as $2l + 1$ nodes to get to the leaf representing a single suffix. its amortized cost for dealing with all $3l$ suffixes is constant. When the d-ptr moves up one node. and over one suffix link. it loses depth at most two nodes in the suffix tree. This is because the node depth of the suffix $c\alpha$ is at most one more that the node-depth of α and a pointer can travel from one node to the next in constant time [23]. So in order to look for $3l$ suffixes. it loses depth up to $6l$ nodes due to following suffix links. moves forward at most $5l$ characters to cover all the overlapping substrings on $2l$ characters on either side of LNP. Thus GTAC moves forward $O(l)$ nodes to remove $3l$ suffixes. □

For a single iteration of step 3. say an LNP of length l with n occurrences is found. It requires $O(l)$ time to remove each one of them. After $n - 1$ of them are removed. a new rule is created and so the remaining occurrence of the LNP is converted to correspond to this rule. So to reduce the size of the original input by $O(nl)$ characters takes time $O(nl)$. and the amount that gets removed can never exceed the original size of the input m. so this phase is $O(m)$ as well. Thus the theorem is proved. □

3.2 Implementation and Experimental Results

Other works in the area of estimating the entropy of genetic sequences have used the same benchmark sequences to compare their estimates. These standard sequences (available at [51]) come from a variety of sources and include the complete genomes.of two mitochondria and two chloroplasts. the complete chromosome III from yeast. five sequences from humans. and finally the complete genome from two

Sequence name	Sequence length	UNIX compress	Bio-compress-2	CDNA compress	GTAC
PANMTPACGA	100314	2.12	1.88	1.85	1.74
MPOMTCG	186609	2.20	1.94	1.87	1.78
CHNTXX	155844	2.19	1.62	1.65	1.53
CHMPXX	121124	2.09	1.68	–	1.58
SCCHRIII	315339	2.18	1.92	1.94	1.82
HUMGHCSA	66495	2.19	1.31	0.95	1.10
HUMHBB	73308	2.20	1.88	1.77	1.73
HUMHDABCD	58864	2.21	1.88	1.67	1.70
HUMDYSTROP	38770	2.23	1.93	1.93	1.81
HUMHPRTB	56737	2.20	1.91	1.72	1.72
VACCG	191737	2.14	1.76	1.81	1.67
HEHCMVCG	229354	2.20	1.85	–	1.74

Table 3.1: Comparison of Entropy Values in Bits Per Symbol

viruses. The complete descriptions are as follows:

- **PANMTPACGA** the complete mitochondrial genome of *Podospora anserina*. a fungus (also called MIPACGA)
- **MPOMTCG** the complete mitochondrial genome of *Marchantia polymorpha*. commonly known as liverwort. which is related to mosses
- **CHNTXX** the chloroplast from *Nicotiana tabacum*. commonly known as the tobacco plant
- **CHMPXX** the chloroplast from *Marchantia polymorpha*. (also called MPOCPCG)
- **SCCHRIII** the complete third chromosome from *Saccharomyces cerevisiae*. commonly known as baker's yeast (also called YSCCHRIII)

- **HUMGHCSA** the human growth hormone (GH-1 and GH-2) and chorionic somatomammotropin (CS-1. CS-2 and CS-5) genes
- **HUMHBB** the human beta globin region on chromosome 11
- **HUMHDABCD** the human DNA sequence of a contig comprising 3 cosmids
- **HUMDYSTROP** the human dystrophin gene
- **HUMHPRTB** the human hypoxanthine phosphoribosyltransferase gene
- **VACCG** the complete genome of the Vaccinia virus
- **HEHCMVCG**the complete genome from cytomegalovirus strain AD169. commonly known as herpes (also called HS5HCMVCG)

On these test sequences. GTAC always beats Biocompress-2. As well. GTAC beats CDNA on seven out of the ten sequence results (and ties on one) that are available for CDNA. The entropy estimates of all four algorithms are presented in Table 3.1.

Sequence name	Sequence length	without reverse complements	with reverse complements
CHNTXX	155844	1.83	1.53
CHMPXX	121124	1.75	1.58
VACCG	191737	1.76	1.67
HEHCMVCG	229354	1.85	1.74

Table 3.2: Recognizing Reverse Complements

When GTAC completely ignores reverse complements, the values are only slightly worse (about 0.01-0.02 bits/symbol) for eight of the twelve sequences. but.

as Table 3.2 illustrates. dramatically different for four sequences: the two chloroplasts. CHNTXX and CHMPXX. and the two viruses. VACCG and HEHCMVCG. The results get worse by between 0.09 - 0.30 bits per symbol. This difference happens because these sequences are known to contain long reverse complements. So for most sequences the overhead of recognizing reverse complements lowers the entropy estimate slightly. but for others it has a more dramatic difference. The ability to recognize reverse complements may also be an issue for CDNA. Two of the sequences in Table 3.2 are ones for which Biocompress-2 beats CDNA. and CDNA's entropy estimates for the other two were not available.

3.3 Conclusion

While the idea of using Context Free Grammars in compression algorithms has been around for a while. the recent results have shown that if these grammars have the addition property that they are asymptotically compact then they are universal. This result has created a whole new family of approaches. One such algorithm in this family. namely GTAC. beats all known competitors for estimating the entropy of a set of standard genetic sequences. and has the additional property that it has linear running time. and has been proven to be universal without assuming an Markov source.

Chapter 4

Distinguishing String Problems

While one of the key problems of the previous chapter was to identify the longest pattern that occurs in a set of strings, the next set of problems concerns finding a pattern that occurs in one set of strings but does not occur in another set. The motivation for this problem is based on the idea of discovering and using genetic information that distinguishes one set of closely related species from another set of species. For example, one application would be to create a drug that would kill several closely related pathogenic bacteria yet would be relatively harmless to humans. A way of approaching this problem is to consider the genes that encode essential proteins. Of these essential genes, try to discover ones that have sequences that are very similar among the bacteria but different from that of humans. This distinguishing feature could then become a potential target for drug design. It is the goal of this and the next chapter to formulate and systematically study the set of optimization problems that underlie this task.

4.1 Hybridization

To understand this approach, one must first understand, in a bit more detail, the ability of nucleic acids to hybridize. As mentioned in Section 1.2.2, when the bases in two strands are the exact reverse complement of each other, the strands can hybridize, or twist together, to form a structure that is more stable than just a single strand. This form of hybridization is called Watson-Crick base pairing and the conditions under which it occurs have been well studied [67]. In fact, under many conditions the two strands do not have to be the exact reverse complement of each other for the double-stranded form to occur. These inexact pairings have been classified into two basic categories whose properties have been investigated. One category is substitutions [6, 42] (also called mismatches), where a base in one strand is not the Watson-Crick pair of the base on the other strand; another category is gaps [43] (also called bulges), where there is at least one extra base in one strand that is not paired with any bases in the other strand. For example, the exact complement of $5' - aaacaaa - 3'$ is $5' - tttgttt - 3'$. A mismatch would be $5' - tttattt - 3'$, while both $5' - tttttt - 3'$, caused by a deletion, and $5' - tttgccttt - 3'$, caused by two insertions, would be bulges.

The destabilizing effect of gaps and substitutions have been tabulated and are quantified in terms of Gibbs free energy. For example, at body temperature and in a 1 M NaCl solution [62] a substitution, the free energy, G°_{s}, is about 0.8 kcal/mol, and for a bulge G°_{b}, it is about 3.3 kcal/mol. However, this does not mean that substitutions are four times more likely to occur than bulges. In fact, substitutions are 58 times more likely to occur in hybridization than are bulges. This difference is because Gibbs free energy is related exponentially to the difference in probability of occurrence of possible structures. Given two short sequences in equal concentrations that could either hybridize with one bulge or with one substitution (assuming that the rest of the base pairing would be the same). then the ratio of substitutions to bulges. $K_{s:b}$. relates to free energy in the following manner [47]:

$$K_{s:b} = e^{\Delta G^{\circ}/RT}$$

where T is the temperature and R is a constant.

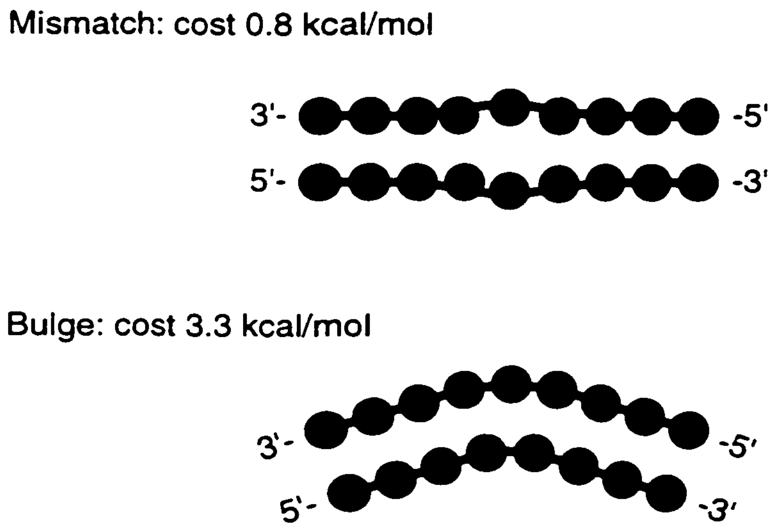

Figure 4.1: The Geometry of Mismatches and Bulges

Just looking at the geometry of the situation adds intuitive support to the larger thermodynamic penalty for bulges as compared to mismatches presented above. Figure 4.1 illustrates that a bulge actually results in a crook in the helix whereas a mismatch results in less of a distortion in the geometry of the sequences. Because gaps are destabilizing and cause greater distortions in the double stranded helix. if the goal is to design an oligomer (short strand of DNA) to bind tightly to another strand. one may use the Hamming distance metric. which considers only substitutions. rather than edit distance. which considers both gaps and substitutions. The use of the Hamming distance metric to estimate the strength of hybridization will be formalized as follows: Given two sequences x and y. with lengths l_x and l_y respectively. if $l_x = l_y$. then the Hamming distance $d(x.y)$ is the number of times $x[i] \neq y[i]$. for all $1 \leq i \leq l_x$. If $l_x < l_y$ then the Hamming distance is the minimum of the Hamming distance between x and each substring of y of length l_x. Additionally. x and y are said to hybridize or bind together if $d(x.y) \leq t$ for some threshold $t \geq 0$. Obviously. the smaller the Hamming distance is between two sequences. the better the match. resulting in a stronger binding between the two strands.

4.1.1 Justification for Hamming Distance

The rationale behind using Hamming distance to model binding is threefold. First. the goal is to create either a drug or a probe that is highly specific. tolerating as few mismatches as possible. By the nature of the way nucleic acids hybridize. an insertion or a deletion is much more destabilizing to the double stranded structure than mismatches [62].

Second. the strategy to find targets in related species is based on the fact that there is a similarity between the coding regions for the same protein in different species. Typically two closely related organisms have similar DNA sequences for similar proteins. For example. the percentage of sequence identity between some mammals is as follows: two humans are 99.63% identical in their DNA. the coding regions between humans and great apes share about 98 – 100%. and the coding region identity between man and mice is between 70 – 90% [59]. There are some exceptions. such as the highly conserved protein ubiquitin. which is identical in humans. mice and even *Drosophila* [59]. Because of the sequence similarity between orthologous proteins (the same protein in different species). there are potentially many candidates for targets in closely related species. Hence Hamming distance is not so restrictive that solutions are unlikely.

Third. Hamming distance also has a role in describing mutations in the protein coding regions of DNA: edit distance is more suitable to measure the *amount* of change that has happened. whereas Hamming distance is more suitable to measure the *effect* of that change. To illustrate this consider the frameshift mutation. As explained in Section 1.4.1. since both DNA and mRNA strings are comprised of four different bases. and protein molecules are comprised of 20 different acids. it takes three consecutive bases in mRNA to code for each amino acid. Although more than one codon can specify the same amino acid. an insertion or deletion of one or two bases in the DNA sequence drastically changes the resulting protein because it shifts the reading frame of translation. For example. the sequence $5' - aaa\ ccc\ ggg\ ttt - 3'$ (which codes for lysine. proline. glycine and phenylalanine) is completely different from $5' - ata\ acc\ cgg\ gtt - 3'$ (which codes for isoleucine. threonine. arginine and valine). yet the latter only has an additional t in the second position (as reflected by an edit distance of one) while the amino acids it codes for has changed completely (as reflected by the Hamming distance).

4.1.2 Additional Considerations

In addition to mismatches and bulges. other factors can influence the likelihood that two strands will hybridize together. many of which can be explained by simple geometric considerations. There are two aspects to the change in hybridization potential due to individual differences in the base pairs: size and hydrogen bonding. The bases c and t are single-ringed structures. and so are less bulky than a and g. which are two-ringed structures. So the mismatch between an a and a g creates a larger deviation in structure than a mismatch between a c and a t. Another factor is the number of hydrogen bonds between base pairs. $a - t$ pairs associate with less strength than do $g - c$ pairs [62] because $a - t$ pairings share two hydrogen bonds whereas $g - c$ base pairings share three. so more accurate modeling would take the differing bonding strengths into consideration. Additionally. more accurate modeling can consider doublets of base pairs because the strength of the hydrogen bond is very sensitive to the distance between the Watson-Crick pairs: neighbouring bases can affect this distance by moving the bases closer or farther away from the ideal distance. For example. $5' - gc - 3'$ binds to $5' - gc - 3'$ more tightly than $5' - gg - 3'$ binds to $5' - cc - 3'$ even though they both consist of two g-c bonds [62. 4. 18]. Other factors that influence the binding include the location of a mismatch: being near the middle has a different effect than being near the end [24]. which allows the ends to float freely apart. Finally the ability of a sequence to hybridize with itself. forming a stem-loop structure (as illustrated in Figure 4.2).

can decrease the likelihood of anything else binding to that area of the genetic sequence [24. 39].

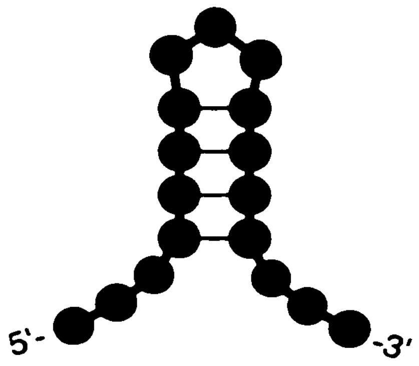

Figure 4.2: The Geometry of a Stem Loop Structure

4.1.3 Some Simplifications

While Section 4.1.2 introduced additional factors that could be included for a more accurate model. from a computational point of view. most (except self-hybridization) can be calculated in constant time. since these differences are functions of each base's individual position. identity. and the identity of its two immediate neighbours. So in the analysis that follows. they will be ignored, with the understanding that it would be simple to add them if the application warrants the additional accuracy.

Another simplification concerns the variety of formats the answer can be provided in. Biological applications occur in two varieties: some require that a region of similarity be discovered. for example consensus sequences. and other applications use the reverse complement of that region. such as designing probes or primers. The algorithms described in this analysis report the region directly with the understanding that the reverse complement of the region can be easily calculated if required. With this in mind, a formulation of the problems to be solved plus several applications using this formulation will be discussed.

4.2 Formulation

The applications that will follow in Section 4.3 all have as a common theme the following two constraints. A string must be found that is both:

1. close or similar to one set of strings. $\mathcal{S}_c$
2. far or different from another set of strings $\mathcal{S}_f$.

Since there are two constraints. it is natural to break the problem up into two subproblems: first. the Closest String Problem. where the distance between the set and the target is less than some value. and second. the Farthest String Problem where the distance between the target and the set is greater than some value. More formally these two problems are as follows:

Closest String Problem

INSTANCE: Given a set $\mathcal{S}_c$ of strings of length n over an alphabet A.

OBJECTIVE: Find a string x of length n over A minimizing k_c such that for every string s in $\mathcal{S}_c$. the Hamming distance $d(x.s) \leq k_c$.

Farthest String Problem

INSTANCE: Given a set S_f of strings of length n over an alphabet A.

OBJECTIVE: Find a string x of length n over A maximizing k_f such that for any s in S_f, the Hamming distance $d(x, s) \geq k_f$.

While sometimes the task is to find a string x that is close to or far from a string s, another possibility is that x must be close to or far from a *substring* of s. This observation leads to two more problems:

Closest Substring Problem

INSTANCE: Given a set S_c of strings of length at least n over an alphabet A.

OBJECTIVE: Find a string x of length n minimizing k_c such that for every string s in S_c, the Hamming distance $d(x, y) \leq k_c$, where y is some length n substring of s.

Farthest Substring Problem

INSTANCE: Given a set S_f of strings of length at least n over an alphabet A.

OBJECTIVE: Find a string x of length n maximizing k_f such that for every string s in S_f, and every length n substring y of s, the Hamming distance $d(x, y) \geq k_f$.

An additional consideration is that sometimes it is impossible to find a string that is far or close to every member in a set, so the next best constraint is to be as far or close to as many members of that set as possible. These leads to two more problems:

Close to Most String Problem

INSTANCE: Given a set S_c of strings of length n over an alphabet A and a threshold $k_c > 0$.

OBJECTIVE: Find a string x of length n maximizing the number of strings s in $\mathcal{S}_c$ satisfying the constraint that the Hamming distance $d(x,s) \leq k_c$.

Far from Most String Problem

INSTANCE: Given a set $\mathcal{S}_f$ of strings of length n over an alphabet A and a threshold $k_f \geq 0$.

OBJECTIVE: Find a string x of length n maximizing the number of strings s in $\mathcal{S}_f$ satisfying the constraint that the Hamming distance $d(x,s) \geq k_f$.

Finally, the Distinguishing String Selection Problem (DSSP) can be formalized as the following.

Distinguishing String Selection Problem

INSTANCE: Given two sets of strings $\mathcal{S}_c$ and $\mathcal{S}_f$, all of length at least n, and two positive integers k_c and k_f.

OBJECTIVE: Find a string x such that for each string s_c in $\mathcal{S}_c$, there exists a substring y_c of s_c such that $d(x,y_c) \leq k_c$ and the length of y_c is n, and for any string s_f in $\mathcal{S}_f$, $d(x,s_f) \geq k_f$.

Certainly there are other combinations possible, such as the Closest to Most Substring Problem and Farthest from Most Substring Problem, as well as various combinations of string and substring constraints on the DSSP, but the problems listed above provide the framework for solving the DSSP, which is the ultimate goal of the next two chapters.

4.3 Applications

4.3.1 Finding Targets for Therapeutic Drug Design

Distinguishing String Selection Problems have the potential to help out in therapeutic drug target selection. Given a dataset of sequences of orthologous genes (the same gene from different species) from a group of closely related pathogens. and a host (typically humans. livestock. or crops). the goal is to find a substring that is fairly well conserved in all or most of the pathogens' sequences (these are the set $\mathcal{S}_c$) but not as conserved in the host's (which would be the set $\mathcal{S}_f$). Information encoded by this fragment then can be used for novel antibiotic development. For example. the conserved region suggests biological importance. since the region is resisting the natural tendency to mutate. This resistance could possibly imply that any mutation is fatal. so chemicals could be screened to identify those that bind to the peptide encoded by the conserved regions of the pathogenic DNA. These chemicals could then be tested as potential broad-range antibiotics. Wareham *et al.* have also looked at this problem. however they use Gibbs sampling to identify the drug targets [64]. Their approach was based on an approach by Rocke and Tompa [56] which in turn was based on an approach by Lawrence *et al.* [36].

4.3.2 Finding Targets for Antisense Drug Design

Using the same information as is used to discover targets for therapeutic drug design. another strategy would be to attack the mRNA that encodes the protein. rather than the protein itself. This approach is called antisense drug design and it represents a new strategy in designing highly specific drugs that inhibit the production of the targeted disease-related proteins while not interfering with other proteins [13]. They work by impeding the translation of targeted genes when they hybridize to the portion of the mRNA corresponding to that gene after the mRNA has been transcribed but before it can be translated into a protein. The mRNA to which they bind is called sense mRNA because it is the template for making protein. hence it makes sense: the drug that binds to the sense mRNA is the reverse complement of the sense mRNA and is called antisense. The composition and structure of antisense drugs is similar to mRNA and they hybridize to the target mRNA using the same base pairing mechanism: however. they have been modified to resist degradation in the cell in order to increase potency and durability. In general. these drugs must be at least 15 nucleotides long in order to bind tightly to target sites and avoid associating with non-target sites [12]. Like therapeutic drug approaches. by identifying a sequence fragment that distinguishes the pathogens from the host. the potential exists to create a drug that harms several pathogens with minimal effect on the host.

4.3.3 Creating Diagnostic Probes for Bacterial Infection

Besides having a potential application in the treatment of disease. the problem of finding a substring that is close to one set of strings and far from another set can also be used for diagnosis. such as the task of creating diagnostic probes. Probes are strands of either DNA or RNA that have been modified (such as being made radioactive or fluorescent) so that their presence can be easily detected. These probes can be used to determine if specific sequences occur in a sample of DNA. For example, a probe has been developed to see if a patient has the sickle cell mutation, which causes them to have sickle cell anemia [59].

Another application is to use a probe for the diagnosis of bacterial infections [7]. A methodology for using a bacterial probe works as follows [46]. First a sequence fragment is found which occurs in the bacterial genome but does not occur in the host's genome. Next a radioactive reverse complement of that substring is synthesized and mixed with genetic material from the sample which is being diagnosed. The sample is then washed so that any probe that has not hybridized to the sample is removed. Finally the sample is tested to see if it contains a significant amount of the radioactivity. A significant amount provides an indication that the bacteria is present in the host.

While probes currently exist for single species of bacteria [7], since many bacteria are treated by the same antibiotic, a useful extension would be to create a probe that would recognize a family of bacteria that would all be treated with the same antibiotic. This is exactly the DSSP. The problem of designing DNA probes to recognize a family of bacteria was introduced into the computational biology literature by Ito *et al.* [27] under the name the characteristic string problem (CSP). They formalized the CSP in a slightly different manner than DSSP: Given a set of strings S and a subset $T \subseteq S$, the k-characteristic string of T under S is a substring occurring in all the strings in T and that is at least distance k away (using edit distance) from any substrings of strings in $S - T$. More formally, using the $x \sqsubset t$ to signify that x is a substring of t, and d_e to signify edit distance, then x is a

k-characteristic string of T under S, if for all $t \in T$ and all $s \in S - T$, $x \sqsubset t$ and $d_c(x, s) > k$. The DSSP set S_c is represented by T in this problem and the set S_f corresponds to $S - T$.

CSP Compared to DSSP

Since the characteristic string is a substring of each string in T, there is a polynomial time solution to this problem [27]. The DSSP studied here is computationally more difficult than the CSP, because whereas CSP requires that all strings in T contain a common substring, the DSSP only requires that all strings in T contain a substring that is within a constant Hamming distance from the characteristic string (which is called the distinguishing string in the DSSP terminology).

4.3.4 Creating Universal PCR Primers

Polymerase chain reaction (PCR) primer design is another area related to the DSSP. PCR is a laboratory technique for amplifying, that is creating many copies of, a portion of DNA [24]. For example, given a small sample of DNA at a crime scene, using PCR this sample can be amplified a million-fold and sequenced to determine from whom it came.

An algorithmic challenge associated with PCR concerns primer design [24]. For the PCR to initiate, two small fragments of DNA, called primers, must be synthesized with the property that each of these primers hybridizes somewhere within a very specific region. Given a region x of DNA to be amplified, the first primer must hybridize somewhere on the 5' side of x on one strand (call this side p) and the

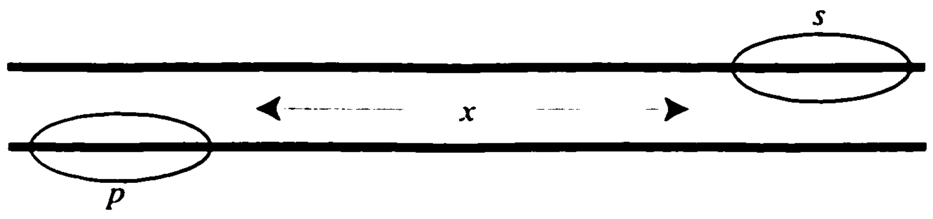

Figure 4.3: The Location of PCR Binding Sites

second primer must hybridize on the 5' side of the opposite strand (call this region s). The PCR process then amplifies the DNA fragment between the two areas where the primers hybridized. If the first primer binds anywhere else in addition to binding in p, or if the second primer binds anywhere else in addition to s, then this secondary binding reduces both the amount and purity of x that is produced.

The binding requirements can be formulated as two separate target selection problems where, in one case, p is the target region and everything else is to be avoided, and in the second case, s is the target region and everything else is to be avoided. This constraint is called the *specificity* of the primer. The DSSP occurs when trying to design a single pair of primers to amplify several different regions simultaneously. More formally, given n different regions, $x_1, x_2, \ldots, x_n$ find a pair of primers, called *universal primers* [31], such that the first primer will bind in each of the prefix regions $p_1, p_2, \ldots p_n$ and nowhere else (in DSSP terminology the prefix regions represent S_c and everything else is S_f), and the second primer will bind in each of the suffix regions $s_1, s_2, \ldots s_n$ and nowhere else. In each case, given a region x_i, the prefix regions, p_i, and the suffix regions, s_i, occur on opposite sides, and opposite strings of x_i. An example for the case $n = 3$ is illustrated in Figure 4.4.

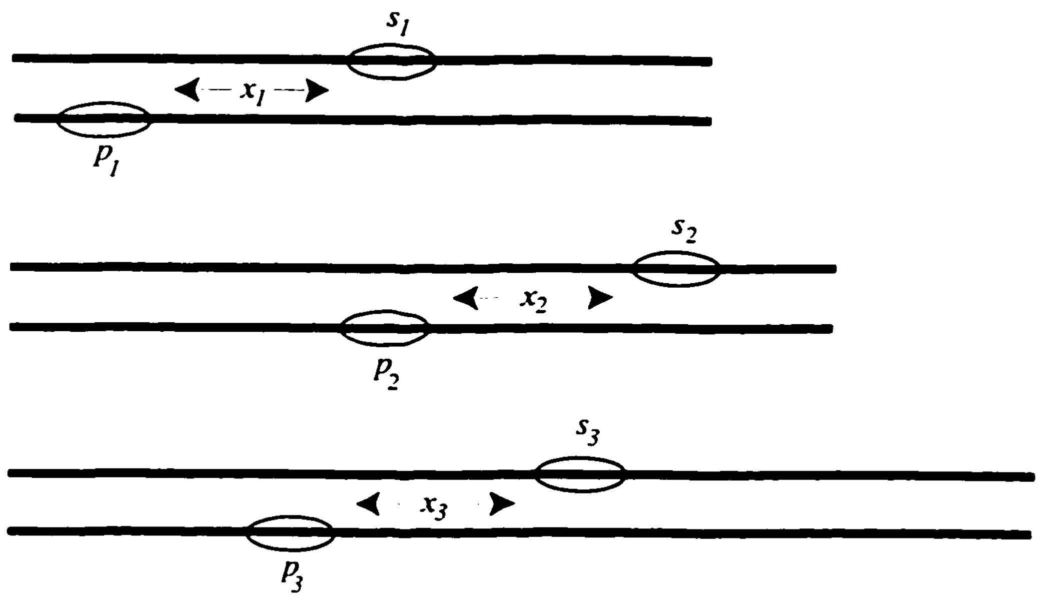

Figure 4.4: The Universal Primer Binding Problem

Of the existing programs that deal with primer design. three of them only consider the relatively simple case where there is only one region to be amplified [5. 44. 57]. Other programs. such as Primer [45], PCRPROF [14] and Prime Master [55] also deal with the more challenging case of universal primers.

Of the programs that search for universal primers: the Primer paper does not state the algorithm by which the targets are selected [45]: PCRPROF assumes that the sequences have already been aligned by another program [14]; and Primer Master uses an unspecified search to find variable and conserved regions and then picks the primer candidates from the conserved regions [55]. None of the programs surveyed do an analysis of the complexity of their algorithms. As with hybridization in general. there are other factors, such as GC content and self-hybridization. that are part of primer design; but again, all of these can be calculated in linear time with respect to the size of the primer.

4.3.5 Creating Unbiased Consensus Sequences

Another application of DSSP is determining consensus sequences. Given a set of related sequences, a consensus sequence is the single sequence that best represents the set. A consensus sequence is determined by aligning the group of closely related sequences, such as a promoter site, and making the i^{th} base in the consensus sequence the majority element in the i^{th} column of the related sequences. As such, the consensus sequence is the sequence which minimizes the total distance from the consensus string to each of the other strings. More formally, given a set of strings S, find a string x minimizing the following sum:

$$\sum_{s \in S} d(x, s)$$

Figure 4.5 illustrates a consensus sequence generated from six related sequences. Note that $5' - tataat - 3'$ does not actually occur in any of the sequences, but is the consensus sequence of the set.

A challenge associated with creating consensus sequences is sample bias. For example, given a dataset of sequences of orthologous genes from many closely related species and a few more distantly related ones, the resulting consensus sequence could be biased towards sequences from the over-represented species. A common way of dealing with this situation is by assigning weights to the sample, so that over-represented sequences would be given a lower weight. Ben-Dor *et al.* [7] re-

	5'-TATATT-3'
	5'-TCTAAT-3'
Related Sequences	5'-GATCAT-3'
	5'-TAATAT-3'
	5'-TATAAC-3'
	5'-TACACT-3'
Consensus Sequence	5'-TATAAT-3'

Figure 4.5: An Example of a Consensus Sequence

view six of these weighting schemes. and then propose a method that minimizes the bias by creating a consensus sequence that minimizes the maximum distance from x to any sequence rather than minimizing the total distance from x to each of the sequences. Ben-Dor *et al.*'s approach of minimizing the maximum distance is a special case of the DSSP where there are no strings to be distant from. that is S_f is empty. which is the Closest String Problem.

Ben-Dor *et al.* [7] also investigated a problem that is very similar to the DSSP. but with a different formulation: Given a set of strings S and a subset $T \subseteq S$. using Hamming distance. $d(\ )$. find a target x that maximizes a distance k such that

$$\min_{v \in S-T} d(x,v) - \max_{w \in T} d(x,w) = k$$

That is. that every element in $S - T$ is at least k further away from the target t than any element in T. This contrasts with the DSSP formulation. presented in Section 4.2. which uses two parameters, k_c and k_f. in place of k. The DSSP case is more general. and simplifies to Ben-Dor *et al.*'s formulation by setting $k_f = k_c + k$.

As well. Ben-Dor *et al.* provide an approximation to the Closest String Problem which with probability less than ϵ they get

$$k + \sqrt{3k \log \frac{|m|}{\epsilon}}$$

where k is the optimal distance and m is the number of strings. However. a small k is critical for practical applications and the straightforward linear programming relaxation method as used in [7] does not work well for small k.

4.4 Related Work

The Closest String Problem also occurs in non-biological applications such as coding theory. and has been proven NP-hard for binary codes [17]. Again in the context of coding theory. Gąsieniec *et al.* [20] independently claimed a $(\frac{4}{3} + \epsilon)$-approximation to the Closest String Problem.

Another non-biological problem. similar to the Closest String Problem is the hitting string problem: Given a set S of strings of length n over $\{0. 1. \star\}$. the hitting string problem is to find a string over $\{0. 1\}$ that has at least one match with each string in S [19]. Such a problem was proved to be NP-hard by Fagin [15]. This problem is a special case of the Closest String Problem in which the Hamming distance bound is $n - 1$ and the sought string lies over $\{0. 1\}$ rather than over the original alphabet $\{0. 1. \star\}$.

Chapter 5

Distinguishing String Properties

5.1 The Complexity of Farthest String Problem

This section characterizes the complexity of the Farthest String Problem. which is NP-hard. The proof is broken up into two cases. The first case deals with alphabet sizes greater than two and the second case considers alphabet sizes exactly equal to two. From a biological point of view. the cases where the alphabet size is four (for nucleic acid sequences) or twenty (for peptide sequences) are the most relevant: however. this chapter will present a complete characterization of the problem.

Theorem 6 *The Farthest String Problem is NP-hard for strings over any alphabet whose size is greater than two.*

Proof. The proof of NP-hardness is approached by reducing the strong 3-SAT problem [19] (that is, it cannot consist of trivial clauses like: x or x or x) to the Farthest String Problem. First consider the case where the alphabet size is three.

that is $|A| = 3$.

Let $I_{m,n}$ be an arbitrary instance of the 3-SAT problem with m clauses $\{C_1, C_2, \cdots, C_m\}$ and n variables $\{v_1, v_2, \ldots, v_n\}$.

Construct $m + 9$ strings each of length $n + 2$. The first n characters of the first m strings will encode the clauses of $I_{m,n}$ and the rest will encode constraints to ensure that the answer is a valid solution to the 3-SAT problem. As well, the i^{th} string will be referred to as s_i and the j^{th} character of the i^{th} string will be called s_{ij}.

Let the first m strings be formulated as follows:

$$s_{ij} = \begin{cases} 0 & v_j \in C_i. \\ 1 & \overline{v}_j \in C_i. \\ \star & v_j \text{ does not appear in } C_i. \\ \star & j = n + 1 or j = n + 2. \end{cases}$$

For the last nine strings, the first n characters will always be $\star$ and the last two characters will be a different one of the nine possible strings of length two on three characters, namely $\{0, 1, \star\}$, that is $\star^9\{0, 1, \star\}^2$. See Figure 5.1 for an example of the encoding of $I_{2,4} = (v_1 \vee \overline{v}_2 \vee v_3) \wedge (v_2 \vee \overline{v}_3 \vee v_4)$. Here, the upper left 2×4 sub-matrix represents the two clauses of $I_{2,4}$ and the rest ensures that the answer will be in $\{0, 1\}$.

With the reduction set up, what remains to be shown is that the instance $I_{m,n}$ is satisfiable if and only if there is a string x of length $n + 2$ on $\{0, 1, \star\}$ such that $d(x, s_i) \geq n$ for every $1 \leq i \leq m + 9$.

```
0 1 0 * * *
* 0 1 0 * *
* * * * 0 0
* * * * 0 1
* * * * 0 *
* * * * 1 0
* * * * 1 1
* * * * 1 *
* * * * * 0
* * * * * 1
* * * * * *
```

Figure 5.1: Encoding of $(v_1 \vee \overline{v}_2 \vee v_3) \wedge (v_2 \vee \overline{v}_3 \vee v_4)$

First, suppose $I_{m,n}$ is satisfied by an assignment $x \in \{0,1\}^n$. Define $x00 = x_1x_2\cdots x_n00$. For the first m strings, if x makes C_i true, then the following three points hold.

1. There is at least one mismatch between $x00$ and s_i where a 1 mismatches a 0 in the first n characters.
2. There are $n-3$ mismatches in the first n symbols where either a zero or a one in $x00$ mismatches a $\star$ in s_i.
3. There are two more mismatches because the last two characters of $x00$ and s_i are different.

For the last nine strings, since x does not contain a $\star$ anywhere, $d(x00, s_{m+j}) \geq n$ for every $1 \leq j \leq 9$. Hence, $d(x00, s_i) \geq n$ for all i such that $1 \leq i \leq n$.

Conversely, suppose there is a string $x = x_1x_2\cdots x_{n+1}x_{n+2}$ such that $d(x, s_i) \geq n$ for every $1 \leq i \leq m+9$. Then x does not contain any $\star$'s in the first n positions, otherwise it would match at least three characters in at least one of the last nine strings, making the distance between that string and x less than or equal to $n-1$. Since x does not contain any $\star$'s in the first n positions, it induces an assignment to the variables v_i. Such an assignment is a satisfying assignment for $I_{m,n}$ since at least one variable in each clause is true because it mismatches.

This is the case for an alphabet of size three. For alphabet sizes p, where $p > 2$, $(p-2)p^2$ extra strings must be added to the m strings, which can be grouped into $p-2$ groups of p^2 characters, where each group consists of the first n characters being one of the characters in the alphabet other than $\{0, 1\}$, and the last two characters being every possible combination of two characters in the alphabet.

Hence the theorem is proved. □

The binary case of the Farthest String Problem requires careful encoding. Since at least three different characters are needed for the proof to work, the basic idea behind the next theorem is to encode these three characters in pairs of binary characters, namely $\{0, 1\}$, with '00' representing '0', '11' representing '1', and '01' representing '$\star$.'

Theorem 7 *The Farthest String Problem is NP-hard even for binary strings.*

Proof. Again, the 3-SAT problem will be reduced to the Farthest String Problem. Let $I_{m,n}$ be an arbitrary instance of the 3-SAT problem with m clauses $\{C_1, C_2, \cdots, C_m\}$ and n variables $\{v_1, v_2, \ldots, v_n\}$.

For each clause C_i, construct a string $s_i = s_{i1}s_{i2}\cdots s_{in}$ of length $2n$ from the set $\{00, 01, 11\}^n$, where s_{ij} is in the following format:

$$s_{ij} = \begin{cases} 00 & v_j \in C_i, \\ 11 & \overline{v}_j \in C_i, \\ 01 & v_j \text{ does not appear in } C_i. \end{cases}$$

Let $p(x, i)$ represent the string $(10)^{i-1}x(10)^{n-i}$, and let $P_n(x)$ represent the set of n strings $\{p(x, i) | 1 \leq i \leq n\}$. As well, let $q(x, i)$ represent the string $(01)^{i-1}x(01)^{n-i}$ and let $Q_n(x)$ represent the set of strings $\{q(x, i) | 1 \leq i \leq n\}$. Then the corresponding instance of the Farthest String Problem is

$$\begin{aligned} S_{I_{n,m}} &= P_n(00) \cup P_n(11) \cup P_n(01) \cup \{(10)^n\} \cup \\ &\quad Q_n(00) \cup Q_n(11) \cup Q_n(10) \cup \{(01)^n\} \cup \{s_i | 1 \leq i \leq m\} \end{aligned}$$

with the distance bound as $n - 1$. Clearly the instance $S_{I_{n,m}}$ is computable in polynomial time, and like the non-binary instance, the purpose of including the various $P_n(), Q_n(), (10)^n, (01)^n$ sets is to force the solution of $S_{I_{n,m}}$ to be a string in $\{00, 11\}^n$.

With the reduction set up, it remains to be shown that the instance $I_{m,n}$ is satisfiable if and only if there exists a string x that is at least $n - 1$ away from every string in the set $S_{I_{n,m}}$.

First assume that there is a solution to the 3-SAT problem. Then a string $x = x_1x_2\cdots x_n$ over the alphabet $\{00, 11\}^n$ can be constructed so that there is at least one mismatch between x and each string s_i causing the Hamming distance to be at least $n-1$, in the following manner:

$$x_i = \begin{cases} 11 & \text{if } v_j \text{ is true.} \\ 00 & \text{if } v_j \text{ is false.} \end{cases}$$

Since x is an element of $\{00, 11\}^n$, then by construction it is at least $n-1$ away from any other string in $S_{I_{n,m}}$.

Next let x be a solution to the Farthest String Problem. In order to prove that x is in $\{00, 11\}^n$, assume the contrary. Factor x into the concatenation of pairs of binary symbols, that is, $x = x_1 x_2 \cdots x_n$ where $x_i \in \{00, 11, 10, 01\}$. Since x is not in $\{00, 11\}^n$, x is in the form $\{00, 11\}^*(10|01)\{01, 11, 10, 01\}^*$.

Let n_{10} and n_{01} denote the numbers of 10s and 01s in the factorization. Without loss of generality, assume that $n_{10} \leq n_{01}$. There are two cases to consider.

Case 1: If $0 < n_{10} \leq n_{01}$, then let the first 10 occur in the i^{th} position in the factorization, that is, $x_i = 10$. Then $d(x, q(10, i)) = (n - n_{10} - n_{01}) + 2(n_{10} - 1) \leq n - 2$, which contradicts the requirement that x is at least $n-1$ units away from all strings.

Case 2: If $n_{10} = 0$ and $n_{01} > 0$, then, $d(x, (01)^n) = n - n_{01} \geq n - 1$. Thus, $n_{01} = 1$. Let the i^{th} position be a location where 01 does not occur. If $x_i = 11$ then $d(x, q(11, i)) = n - 2$, and if $x_i = 00$ then $d(x, q(00, i)) = n - 2$.

Hence x is a string in the alphabet $\{11, 00\}^n$ that is at least $n-1$ away from any string in $\{s_i | 1 \leq i \leq n\}$, and this string can be used to find a satisfying truth assignment for the 3-SAT problem because a mismatch of 00 to 11 in each string will correspond to a satisfying assignment for at least one term in each clause.

□

5.2 The Farthest String Problem

Although the Farthest String Problem is NP-hard. through the use of a linear programming relaxation technique. a polynomial time approximation scheme (PTAS) for the Farthest String Problem can be presented. The first step towards this goal is to prove a lower bound.

Lemma 4 *Let S be a set of m strings. $S = \{s_1, s_2, \ldots, s_m\}$. each of length n over an alphabet A of t symbols. If $n > 6\ln(2m/\beta)/\beta^2$ where β is a constant such that $0 < \beta \leq 1$. then there exists a string x such that the Hamming distance between s_i and x is at least $(1-\beta)\frac{n(t-1)}{t}$ for every $s_i \in S$.*

Proof. The theorem can be proved by a probabilistic argument. Let s_i be a string in S. Given a random string $x \in A^n$. the expected value of the Hamming distance between s_i and x is $\frac{n(t-1)}{t}$. Thus. by a Chernoff bound [50].

$$\Pr(|d(x, s_i) - \tfrac{n(t-1)}{t}| > \beta\tfrac{n(t-1)}{t}) \leq \frac{2}{e^{0.38\beta^2 \frac{n(t-1)}{t}}}$$

By requiring $n > 6\ln(2m/\beta)/\beta^2$, and substituting this value for n in the above inequality. the results yield that the right hand side is bounded from above by β/m. Hence,

$$\Pr(\cup_{1\leq i\leq m}|d(x, s_i) - \frac{n(t-1)}{t}| > \beta\frac{n(t-1)}{t}) < \beta$$

Thus, there exists a string x such that $d(x, s_i) \geq (1-\beta)\frac{n(t-1)}{t}$ for all strings s_i.

□

5.2.1 Formulation of Hamming Distance

With the lower bound established, the next step is to set up a formulation of the Farthest String Problem. Towards this end, let $S = \{s_1, s_2, \ldots, s_m\}$ be a set of m strings, each of length n over an alphabet $A = \{a_1, a_2, \ldots a_t\}$ of t symbols.

Let $\beta > 0$ be any small number. If $n < 6\ln(2m/\beta)/\beta^2$, perform an exhaustive search to find an optimal solution. Otherwise, set up the following zero-one integer programming formulation of the problem. Let $s = s_1s_2\cdots s_n \in S$ and let $x = x_1x_2\cdots x_n$ be an arbitrary string over the alphabet A. The Hamming distance between s and any x can be calculated as follows. For each character position x_i, introduce t variables x_{ij} with $1 \leq j \leq t$, so the distance between s and x is of the following form.

$$d(s, x) = \frac{1}{2}\sum_{i=1}^{n}(\sum_{j=1}^{t} c_{hij}x_{ij} + 1)$$

such that if the i^{th} symbol in s_h is a_j, that is $s_hi = a_j$, then

$$c_{hij} = \begin{cases} -1 & \text{if } s_i = a_j \\ +1 & \text{otherwise.} \end{cases}$$

As well. x is of the following form:

$$x_{ij} = \begin{cases} 1 & \text{if } x_i = a_j \\ 0 & \text{otherwise.} \end{cases}$$

When solving for each x_{ij}. there are two additional constraints. First. require that $x_{ij} \in \{0.1\}$ for $1 \leq i \leq n$ and $1 \leq j \leq t$. and also that

$$\sum_{j=1}^{t} x_{ij} = 1 \qquad \forall\, 1 \leq i \leq n.$$

These two constraints capture the fact that if the i^{th} character in x is the q^{th} symbol in the alphabet. (that is. $x_i = a_q$). then $x_{iq} = 1$ and $x_{ij} = 0$ for $j \neq q$.

5.2.2 Formulation of Farthest String Problem

With this formulation. the Farthest String Problem is stated as:

$$\begin{array}{ll} \text{find } x \text{ that maximizes } k_f & \\ \text{subject to the constraints :} & \\ \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{t} c_{hij}x_{ij} + n \geq k_f & \forall\, 1 \leq h \leq m; \\ \sum_{j=1}^{t} x_{ij} = 1 & \forall\, 1 \leq i \leq n; \\ x_{ij} \in \{0.1\} & \forall\, 1 \leq i \leq n, 1 \leq j \leq t. \end{array} \tag{5.1}$$

For example. if the set $S = \{ca. ga\}$ in the alphabet $A = \{a. c. g\}$ then the linear program would be the following.

$$\begin{aligned}
&\text{find } x \text{ that maximizes} \quad k_f \\
&\text{subject to the constraints :} \\
&\tfrac{1}{2}(x_{11} + 1 - x_{12} + x_{13} + 1 - x_{21} + x_{22} + x_{23}) \geq k_f \\
&\tfrac{1}{2}(x_{11} + x_{12} + 1 - x_{13} + 1 - x_{21} + x_{22} + x_{23}) \geq k_f \\
&x_{11} + x_{12} + x_{13} = 1 \\
&x_{21} + x_{22} + x_{23} = 1 \\
&x_{ij} \in \{0, 1\} \quad \forall\ 1 \leq i \leq n, 1 \leq j \leq t;
\end{aligned} \tag{5.2}$$

5.2.3 Solving the Integer Program

To solve this set of equations, first relax the integrality constraints on each x_{ij}, replacing them with the constraints $0 \leq \overline{x}_{ij} \leq 1$ and solve the resulting linear program. Let $\overline{x} = (\overline{x}_{ij})$ be the solution vector of this resulting linear program and let the objective function have the value $\overline{k}_f$.

Now apply randomized rounding [49] to restore integrality: for each i, independently set each character place holder to a specific symbol in the alphabet, say $x_{ip} = 1$ and set the remaining place holders to zero, $x_{ij} = 0$ for $j \neq p$, according to the probability distribution $\overline{x}_{ij}$ for $1 \leq j \leq t$. Thus, for each i,

$$\Pr(x_{ij} = 1) = \overline{x}_{ij}$$

The randomized rounding process has the property that the integer solution,

k_f, is related to the continuous solution, $\overline{k}_f$, in the following manner [49]:

$$k_f > \overline{k}_f - \epsilon O(\sqrt{n \log n})$$

where $\epsilon > 0$ is an arbitrary constant (related to β).

Letting $k_{f,opt}$ be the unknown optimal distance, by Lemma 4,

$$k_{f,opt} > (1-\beta)\frac{n(t-1)}{t}$$

hence $k_{f,opt}$ is $O(n)$.

Consider the ratio between k_f and $k_{f,opt}$, for large n.

$$\frac{k_f}{k_{f,opt}} \geq \frac{\overline{k}_f}{k_{f,opt}} - \frac{\epsilon O(\sqrt{n \log n})}{k_{f,opt}} \geq 1 - \epsilon O(\sqrt{(\log n)/n}) \geq 1 - \epsilon.$$

The third inequality was obtained from the second in two steps:

1. Replace $k_{f,opt}$ by $\overline{k}_f$ in the first denominator. Since $\overline{k}_f$ was obtained using real values and $k_{f,opt}$ was obtained using integer values, the inequality is preserved because $\overline{k}_f \geq k_{f,opt}$.

2. Replace the $k_{f,opt}$ by $O(n)$ in the second denominator.

With this bound achieved, the above algorithm can result in a PTAS after de-randomization. One approach to de-randomizing would be with conditional probabilities, (see [1] or [50]). These results can be generalized into the following result.

Theorem 8 *There is a PTAS for the Farthest String Problem.*

With this theorem. the next corollary is straightforward.

Corollary 1 *The Farthest Substring Problem is NP-hard and has a PTAS.*

Proof. The Farthest Substring Problem can be broken down to the Farthest String Problem in the following manner. Let $S = \{s_1. s_2. \ldots. s_m\}$ be a set of strings. each of length greater than or equal to n over an alphabet A of t symbols. Given the i^{th} string. s_i. has length $p \geq n$. break s_i up into $p - n + 1$ overlapping strings each of length n. Use the algorithm for the Farthest String Problem on these substrings.

□

5.3 The Closest String and Substring Problems

With the foundation laid by the investigation of the Farthest Strings Problems. much of the work of characterizing the Closest String Problems is fairly straightforward.

5.3.1 Hardness Results

Theorem 9 *The Closest String Problem is NP-hard.*

Proof. Reduce the Farthest String Problem to this problem. First consider the case of $|A| = 2$. Given an integer k_c and a set of m strings $S = \{s_1. s_2. \ldots s_m\}$ each of length n. the following statements are all equivalent for $1 \leq i \leq m$.

- There is a string x such that $d(x, s_i) \geq k_c$.
- There are at least k_c mismatches between x and s_i.
- There are strictly less than $k_f = n - k_c$ mismatches between s_i and $\overline{x}$, the complement of x.
- $d(\overline{x}, s_i) \leq k_f$.

The binary case can be reduced to the case $|A| > 2$ in the following manner. Given an instance of the binary Closest Substring Problem, $I_{m,n}$ over the alphabet A_2, use the non-binary algorithm to solve $I_{m,n}$ to the distance bound k_c. Then take that non-binary solution, convert each letter that is not in A_2 to one of the symbols in A_2 arbitrarily. This step will only improve the quality of the solution, yielding a binary solution within distance k_c. By this reduction, the case $|A| > 2$ is also NP-hard. □

Since the Closest String Problem is a special case of the Closest Substring Problem where the length of all strings in S is n, Theorem 9 leads to the following corollary.

Corollary 2 *The Closest Substring Problem is NP-hard.*

With the NP-hardness of these problems determined, the next step is to establish the hardness of approximation. First, consider the following.

Lemma 5 *Both the Closest String Problem and the Closest Substring Problem can be approximated within a ratio of two in polynomial time.*

Proof. Consider the following special case of the star alignment approximation algorithm[23]. Let $I_{n,m}$ be an instance of the Closest Substring Problem, consisting of a set $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ each of length at least n over an alphabet A. Let x be an optimal solution such that for each string $s_i \in \mathcal{S}$, including s_1, there exists a substring p_i of s_i such that $d(x, p_i) \leq k_c$. For every other string in $\mathcal{S}$, say s_j, by the triangle inequality, there exists a substring p_j of s_j such that $d(p_1, p_j) \leq 2k_c$. To find this string p_1, just try each substring of length n in s_1.

Since the Closest String Problem is just a special case of the Substring Problem, the results apply to both. □

The next theorem improves the previous approximation.

Theorem 10 *The Closest String Problem can be approximated within* $\frac{4}{3} + \epsilon$ *in polynomial time for any small constant* $\epsilon > 0$ *and for any alphabet* A *whose size is two or more.*

Proof. Let $\mathcal{S} = \{s_1, s_2, \ldots s_m\}$ be a set of strings each of length n over an alphabet A. Without loss of generality, assume the following:

1. $d(s_1, s_2) = k \geq d(s_i, s_j)$ for any $s_i, s_j \in \mathcal{S}$.
2. The k mismatches between s_1 and s_2 occur at the first k positions.

These assumptions are valid because the characters and strings of $\mathcal{S}$ could be permuted to this format, the problem solved, and then the characters permuted back to their original positions without affecting the validity of the solution.

To simplify things. consider the following notation. Given a string x of length n. let x' represent the first k characters and let x'' represent the rest of the string. For example. $s_i = s_i's_i''$ where $s_i' = s_{i1}s_{i2}\cdots s_{ik}$ and $s_i'' = s_{i(k+1)}s_{i(k+2)}\cdots s_{in}$.

With this notation. consider the following integer program for s_i'

find x that minimizes k'

subject to the constraints :

$$\begin{array}{ll} \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{t} c_{hij}x_{ij}' + k \le k' - k_h'' & \forall\ 1 \le h \le m; \\ \sum_{j=1}^{t} x_{ij}' = 1 & \forall\ 1 \le i \le k; \\ x_{ij}' \in \{0,1\} & \forall\ 1 \le i \le k. 1 \le j \le t. \end{array} \tag{5.3}$$

where x' is an unknown string of length k. and for each i. k_i'' is the constant $d(s_1''.s_i'')$ and the c_{hij}'s encapsulate the Hamming distance as specified in Subsection 5.2.1.

The constraints are similar to Theorem 8 except the length is now k instead of n. the goal is to minimize rather than maximize. and the bounds are $k' - k_h''$ instead of k_f

Let the optimal bound for integer program 5.3 be k_{opt}' with solution x_{opt}'. It is impossible to compute k_{opt}' efficiently. Therefore. approximate it by considering two cases.

1. If $k' \le 6\ln(2m/\beta)/\beta^2$ where β is a constant such that $0 < \beta \le 1$. then use an exhaustive search to find the optimal solution.

2. Otherwise. since $k' > 6\ln(2m/\beta)/\beta^2$. follow the same process as outlined in Theorem 8: linear relaxation. solving for $\overline{x}'$ and $\overline{k}'$. followed by randomized

rounding. letting the resulting solution be x'. Finally. output either s_1 or $x's_1''$ depending on which has the minimal distance from the rest of the strings in $\mathcal{S}$.

Analysis of Algorithm

For case one above. the solution is exact: for case two. some analysis needs to be performed.

Recall from Subsection 5.2.3 that the randomized rounding process has the property that the integer solution. k'. is related to the continuous solution. $\overline{k}'$. in the following manner [49]:

$$k' > \overline{k}' + \epsilon O(\sqrt{k \log k})$$

Where the error is now an overestimate because we are minimizing rather than maximizing.

In Theorem 8. since the solution was $O(n)$ and the error was $O(\sqrt{n \log n})$ the result was a PTAS because. in the limit. the error is small compared to the answer. In this case. the two strings s_1' and s_2' are k apart and so the final answer k' must be greater than or equal to $k/2$ since this is the best that can be done with these two strings. So with this formulation the solution is also $O(k)$. Now consider the ratio of k' to k'_{opt}.

$$\frac{k'}{k'_{opt}} \geq \frac{\overline{k}'}{k'_{opt}} + \frac{\epsilon O(\sqrt{k \log k})}{k'_{opt}} \geq 1 - \epsilon O(\sqrt{(\log k)/k}) \geq 1 - \epsilon.$$

Hence, again a PTAS is obtained for x' and k'.

However, the relationship between the solution for the original problem (with parameters x and n) and this formulation (with parameters x' and k) has not been determined and determining this relationship is the goal of the rest of the proof.

Let x_{opt} be the solution string with optimal distance bound k_{opt} for the original instance of S.

If $d(s_1, s_2) \leq \frac{4}{3}k_{opt}$, then the solution given by the algorithm satisfies the required distance bound, $\frac{4}{3}k_{opt}$.

Otherwise, $d(s_1', s_2') = d(s_1, s_2) > \frac{4}{3}k_{opt}$. By the triangle inequality,

$$d(s_1', x_{opt}') + d(x_{opt}', s_2') \geq \frac{4}{3}k_{opt}$$

Thus, either $d(s_1', x_{opt}') > \frac{2}{3}k_{opt}$ or $d(s_2', x_{opt}') > \frac{2}{3}k_{opt}$.

Without loss of generality, assume the first holds. Then $d(s_1'', x_{opt}'') \leq \frac{1}{3}k_{opt}$.

Next, consider the following. For any $i \leq n$,

$$d(x_{opt}', s_i') = d(x_{opt}, s_i) - d(x_{opt}'', s_i'').$$

Since $d(x_{opt}, s_i) \leq k_{opt}$,

$$d(x_{opt}', s_i') \leq k_{opt} - d(x_{opt}'', s_i'').$$

By rearranging the triangle inequality, $d(s_1'', s_i'') \leq d(s_1'', x_{opt}'') + d(x_{opt}'', s_i'')$ the following is obtained:

$$d(x'_{opt}, s'_i) \leq k_{opt} - (d(s''_1, s''_i) - d(s''_{opt}, s''_1))$$

which finally yields

$$d(x'_{opt}, s'_i) \leq \frac{4}{3}k_{opt} - d(s''_1, s''_i).$$

Thus, this results in $k'_{opt} \leq \frac{4}{3}k_{opt}$, which implies

$$d(x', s_i) \leq (\frac{4}{3} + \epsilon)k_{opt} - d(s''_1, s''_i).$$

Thus,

$$d(x's''_1, s_i) = d(x', s'_i) + d(s''_1, s''_i) \leq (\frac{4}{3} + \epsilon)k_{opt}.$$

This implies that the algorithm outputs a solution with distance bound at most $(\frac{4}{3} + \epsilon)k_{opt}$. □

5.3.2 Heuristic for Closest Substring Problem

Theorem 10 can be used to design an efficient heuristic for the Closest Substring Problem. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be a set of strings where the length of each string is at least n. Without loss of generality, assume s_1 is either shorter than, or the same length as any other string in $\mathcal{S}$. The heuristic takes two steps.

1. For each p_i. a substring of s_1. and for each s_j. a string in $\mathcal{S}$ such that $j \neq 1$. find a substring of s_j with length n that is closest to p_i. call it q_j. such that

$$\max_{p_i, s_j} \quad d(p_i, q_j)$$

is minimized.

2. Apply the algorithm in Theorem 10 to improve the solution.

5.4 Distinguishing String Selection Problem

Let $(A, n, \mathcal{S}_c, k_c, \mathcal{S}_f, k_f)$ be an instance of the Distinguishing String Selection Problem (DSSP). The DSSP requires a string x of length n such that for each $s_c \in \mathcal{S}_c$. $d(x, p) \leq k_c$ for some substring p of s_c. and for each $q \in S_f$. $d(x, q) \geq k_f$. By setting either $k_c = n$ or $k_f = 0$ respectively. both the Farthest String and Closest Substring Problems are special cases of the Distinguishing String Selection Problem (DSSP). Hence the DSSP is NP-hard.

When k_c is a small constant. a solution can be found by exhaustive search. This is because. if s_c is the shortest string in $\mathcal{S}_c$. there are at most

$$O((|s_c| - n + 1)n^{k_c}(|A| - 1)^{k_b})$$

candidates and each candidate can be tested to determine if it is a solution in time that is polynomial in terms of the size of tlie instance. When k_c is relatively large. the following approximation result is obtained.

Theorem 11 *Let $(A, n, \mathcal{S}_c, k_c, \mathcal{S}_f, k_f)$ be an instance of the Distinguishing String Selection Problem. If there is a solution for the instance, an approximate solution x can be obtained in polynomial time such that for each $s_c \in \mathcal{S}_c$, $d(x, p) \leq 2k_c$ for some substring p of s_c, and for each $s_f \in \mathcal{S}_f$, $d(x, s_f) \geq k_f - k_c$.*

Proof. Suppose x_o is a solution, and x is a substring of a sequence in $\mathcal{S}_c$ such that $d(x_o, x) \leq k_c$. For any $s_c \in \mathcal{S}_c$, there is a substring p of s_c such that by the triangle inequality $d(x, p) \leq 2k_c$. For any $s_f \in \mathcal{S}_f$, since $d(x_o, s_f) \geq k_f$, then

$$d(x, s_f) \geq d(x_o, s_f) - d(x_o, x) \geq k_f - k_c$$

Hence the theorem is proved. $\square$

Chapter 6

Conclusion

6.1 GTAC

It has been established that the GTAC entropy estimator converges under relatively modest assumptions. and experimental work suggests that it converges quickly relative to other algorithms. However. the difficult task of establishing its theoretical convergence rate remains open.

As well. GTAC's greedy approach is to take the longest pattern at any given time. rather than the pattern that has the largest product of length and frequency. For example. given a choice between a pattern of length 20 that occurs twice. and a pattern of length 18 that occurs 20 times. picking the second pattern may result in lower entropy estimates.

Finally. a challenging modification of GTAC is to include some of the approaches that other methods use. such as recognizing inexact matches. which its competitor CDNA does. Such work has recently begun with the paper by Chen *et al.* [11].

6.2 Distinguishing String

There is still work remaining with the Distinguishing String Selection family of problems. The Distinguishing String Selection Problem. the Far From Most Problem. and Close To Most Problem have String and Substring variations which have not been fully explored. Additionally. the Close To Most String Problem is an open problem. Others are continuing in the area. such as the recent paper by Li *et al.* which presents a PTAS for the Closest String Problem.

As well. although these problems fit well with the one-dimensional genetic sequence framework. perhaps they could be extended into three-dimensional protein-ligand interactions which are a fundamental part of the therapeutic drug mechanism.

6.3 Computational Biology

Computational biology is a young and vibrant area of research with many promising avenues of investigation. More than any other time in the history of science. biologists are in the enviable position of being faced with a wealth of biological information. more than any single researcher can hope to understand. Clearly. a next step is to use existing computational tools—and to create new ones—in order to use this information effectively. Hopefully this thesis will serve as a step in that direction.

Bibliography

[1] N. Alon. J. Spencer and P. Erdös. *The Probabilistic Method*. Wiley Interscience. New York. 1992.

[2] E. Amaldi and V. Kann. The complexity and approximability of finding maximum feasible subsystems of linear relations. *Theoret. Comput. Science* 147 (1995) 181-210.

[3] S. Arora. C. Lund. R. Motwani. M. Sudan and M. Szegedy. Proof verification and intractability of approximation problems. In *Proc. of 33rd FOCS*. 13-22. 1992.

[4] K. J. Breslauer. R. Frank, H. Blöcker. L. A. Marky. Predicting DNA duplex stability from the base sequence, *Proc. Natl. Acad. Sci. USA*. 83 (1986) 3746-3750.

[5] C.G. Bridges, Olga—oligonucleotide primer design program for the Atari ST. *CABIOS* 6 (1990) 124-125.

[6] T. Brown. G. A. Leonard. E. D. Booth and G. Kneale. Influence of pH on the Conformation and Stability of Mismatch Base-pairs in DNA. *J. Mol. Biol.* 212 (1990) 437-440.

[7] A. Ben-Dor, G. Lancia, J. Perone. and R. Ravi. Banishing Bias from Consensus Sequences. *Combinatorial Pattern Matching. 8th Annual Symposium* pp. 247-261. 1997.

[8] BioInformatics Group Homepage. `http://wh.math.uwaterloo.ca`.

[9] R. Cameron. Source Encoding Using Syntactic Information Source Models. *IEEE Trans. Inform. Theory* 34 (1988) 843-850.

[10] P Carmosino. From Darwin to the Human Genome Project. *The Chico Anthropological Society Papers* 16 (1996) 46-55.

[11] X. Chen. S. Kwong. and M. Li A Compression Algorithm for DNA Sequences and its Application in Genome Comparison. in *Tenth Workshop on Genome Informatics*. 1999.

[12] J. S. Cohen and M. E. Hogan. The New Genetic Medicines. *Scientific American* 171 (1994) 76-82.

[13] S. Crooke and B. Lebleu (editors), *Antisense Research and Applications*. CRC Press. Boca Raton. 1993.

[14] J. Dopazo. A. Rodríguez, J. C. Sáiz and F. Sobrino. Design of primers for PCR amplification of highly variable genomes, *CABIOS* 9 (1993) 123-125.

[15] R. Fagin. Generalized first-order spectra and polynomial time recognizable sets. *Complexity of Computation* (R. Karp ed.). America Mathematical Society. Providence. 43-73. 1974.

[16] M. Farach. M. Noordewier. S. Savari. L. Shepp. A. Wyner. and A. Ziv. On the entropy of DNA: Algorithms and measurements based on memory and rapid convergence. *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* pp. 48-57. 1994.

[17] M. Frances and A. Litman. On Covering Problems of Codes. *Theory of Computing Systems* 30 (1997) 113-119.

[18] S. M. Freier. R. Kierzek. J. A. Jaeger. N Sugimoto. M. H. Caruthers. T. Neilson and D. H. Turner. Improved free-energy parameters for predictions of RNA duplex stability. *Proc. Natl. Acad. Sci. USA*. 83 (1986) 9373-9377.

[19] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. W.H. Freeman. San Francisco. 1979.

[20] L. Gąsieniec. J. Jansson. and A. Lingas. Efficient Approximation Algorithms for the Hamming Center Problem. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* pp. 905-906. 1999.

[21] S. Grumbach and F. Tahi. Compression of DNA sequences. *Proceedings of the IEEE Symposium on Data Compression*. 340-350. 1993

[22] S. Grumbach and F. Tahi. A New Challenge for Compression Algorithms: Genetic Sequences. *Information Processing & Management* 30 (1994) 875-886.

[23] D. Gusfield. *Algorithms on Strings. Trees. and Sequences*. Cambridge University Press. Cambridge. 1997.

[24] E. Hillis. C. Moritz. and B. Mable. *Molecular Systematics*. 2nd ed.. Sinauer Associates Inc.. Sunderland. 1996.

[25] J. Hopcroft and J. Ullman. *Introduction to Automata Theory. Languages. and Computation*. Addison-Wesley. Reading. 1979.

[26] IEEE Computer Society. Timeline of Computing History. `http://computer.org/computer/timeline/index.html`.

[27] M. Ito. K. Shimizu. M. Nakanishi and A. Hashimoto. Polynomial-time algorithms for computing characteristic strings. *Proc. of Fifth Symposium on Combinatorial Pattern Matching*. 274-288. 1994.

[28] E. Kawaguchi and T. Endo. On a Method of Binary-Picture Representation and Its Application to Data Compression. *IEEE Trans on Pattern Analysis and Machine Intelligence* 2 (1980) 27-35.

[29] J. Kieffer and E. Yang. Ergodic Behavior of Graph Entropy. *ERA Amer. Math. Society*. 3 (1997) 11-16.

[30] J. Kieffer and E. Yang, Grammar Based Codes: A New Class of Universal Lossless Source Codes. submitted for journal publication.

[31] T. D. Kocher and T. J. White. Evolutionary Analysis via PCR in H. A. Erlick (ed.) *PCR Technology: Principles and Applications for DNA Amplification* Stockten Press. New York. 1995.

[32] E. Kourapova and B. Ryabko, Application of Formal Grammars for Encoding Information Sources, *Problems of Information Transmission*, 31 (1995) 23-26.

[33] J. Lanctot, M. Li, B. Ma, S. Wang, L. Zhang Distinguishing String Selection Problems, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* pp. 633-642, 1999.

[34] J. Lanctot, M. Li, B. Ma, S. Wang, L. Zhang Distinguishing String Selection Problems, accepted for publication in *Information and Computation*.

[35] J. Lanctot, M. Li, and E. Yang, Estimating DNA Sequence Entropy, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms* pp. 409-418, 2000.

[36] C. Lawrence, S. Altschul, M. Boguski, J. Liu, A Neuwald, and J. Wootton, Detecting Subtle Sequence Signals: a Gibbs Sampling Strategy for Multiple Alignment, *Science*, 262 (1993) 208-214.

[37] B. Lewin, *Genes VI*, Oxford University Press, Oxford, 1997.

[38] M. Li, B. Ma, and L. Wang, Finding Similar Regions in Many Sequences, *Proceedings of the 31st ACM Symposium on the Theory of Computing* pp. 473-482, 1999.

[39] W. F. Lima, B. P. Monia, D. J. Ecker, S. M. Freier, Implications of RNA Structure on Antisense Oligonucleotide Hybridization Kinetics, *Biochemistry*, 31 (1992) 12055-12061

[40] P. Liò, A. Politi, M. Buiatti, and S. Ruffo. High Statistics Block Entropy Measures of DNA Sequences, *Journal of Theoretical Biology* 180 (1996) 151-160.

[41] D. Loewenstern and P. Yianilos. Significantly Lower Entropy Estimates for Natural DNA Sequences, *Journal of Computational Biology* 6 (1999) 125-142.

[42] A. Lomant and J. Fresco. Structural and Energetic Consequences of Noncomplementary Base Oppositions in Nucleic Acid Helices *Nucleic Acid Res. and Mol. Bio.* 15 (1975) 185-218.

[43] C. Longfellow, R. Kierzek, D. Turner. Thermodynamic and Spectroscopic Study of Bulge Loops in Oligoribonucleotides. *Biochemistry* 29 (1990) 278-285.

[44] T. Lowe, J. Sharefkin, S. Yang and C. Dieffenbach. A computer program for selection of oligonucleotide primers for polymerase chain reactions. *Nucleic Acids Res.* 18 (1990) 1757-1761.

[45] K. Lucas, M. Busch, S. Mössinger and J. Thompson. An improved microcomputer program for finding gene- or gene family-specific oligonucleotides suitable as primers for polymerase chain reactions or as probes, *CABIOS* 7 (1991) 525-529.

[46] A. Macario and E. Macario, *Gene Probes for Bacteria*, Academic Press, San Diego, 1990.

[47] C. Mathews and K. van Holde. *Biochemistry*, Benjamin/Cummings Publishing Company, Inc., Redwood City, 1990.

[48] P. S. Miller, Development of Antisense and Antigene Oligonucleotide Analogs. *Nucleic Acid Research and Molecular Biology* 52 (1996) 261-291.

[49] R. Motwani, J. Naor, and P. Raghavan. Randomized Approximation Algorithms in Combinatorial Optimization *Approximation Algorithms for NP-Hard Problems* (D. Hochbaum, Ed.), pp. 447-481, PWS Publishing Company, Boston, pp. 447-481, 1995.

[50] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[51] National Center for Biotechnology Information. Entrez Nucleotide Query. `http://www.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=n_s`.

[52] C. Nevill-Manning and I. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intell. Res.* 7 (1997) 67-82.

[53] M. Noordewier. CDNA Source Code. `http://paul.rutgers.edu/~loewenst/cdna.html`.

[54] C. H. Papadimitriou and M. Yannakakis, Optimization, approximation and complexity classes, *JCSS* 43 (1991) 425-440.

[55] V. Proutski and E. Holme, Primer Master: a new program for the design and analysis of PCR primers, *CABIOS* 12 (1996) 253-255.

[56] E. Rocke and M. Tompa. An Algorithm for Finding Novel Gapped Motifs in DNA Sequences. *Proceedings of the Second Annual International Conference on Computational Molecular Biology*. 228-233. 1998.

[57] W. Rychik and R. Rhoads. A computer program for choosing optimal oligonucleotides for filter hybridization. sequencing and *in vitro* amplification of DNA. *Nucleic Acids Res*. 17 (1989) 8543-8551.

[58] A. Schmitt and H. Herzel. Estimating the Entropy of DNA Sequences. *Journal of Theoretical Biology* 188 (1997) 369-377.

[59] T. Strachan and A. Read. *Human Molecular Genetics* Wiley-Liss. New York. 1996

[60] J. Storer and T. Szymanski. Data Compression via Textual Substitution. *Journal of the Association Computing Machinery* 29 (1982) 928-951.

[61] L. Stryer. *Biochemistry*. 3rd ed.. W.H. Freeman. New York. 1988.

[62] D. Turner. N. Sugimoto and S. Freier. RNA Structure Prediction. *Ann. Rev. Biophys. Chem*. 17 (1988) 167-192.

[63] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica* 14 (1995) 249-260.

[64] H. Wareham. T. Jiang. X. Zhang. and C. Trendall. Stochastic Heuristic Algorithms for Target Motif Identification. *Pacific Symposium on Biocomputing*. 389-400. 2000.

[65] T. A. Welch, A technique for high performance data compression. *Computer* 17. (1984) 8-19.

[66] R. Werner *Essential Biochemistry and Molecular Biology*. 2nd ed.. Appleton & Lange. Norwalk. 1992.

[67] J. G. Wetmur. DNA Probes: Applications of the Principles of Nucleic Acid Hybridization. *Critical Rev. in Biochem. and Mol. Bio.* 26 (1991) 227-259.

[68] J. G. Wetmur. DNA Probes: Applications of the Principles of Nucleic Acid Hybridization. *Critical Rev. in Biochem. and Mol. Bio.* 26. (1991) 227-259.

[69] S. Woodson and D. Crothers. Proton Nuclear Magnetic Resonance Studies on Bulge-Containing DNA Oligonucleotides from a Mutational Hot-Spot Sequence. *Biochemistry* 26 (1987) 904-912.

[70] E. Yang and J. Kieffer. Efficient universal lossless compression algorithms based on a greedy sequential grammar transform–Part one: Without context models. to appear in *IEEE Trans. Inform. Theory.*

[71] E. Yang and J. Kieffer. Universal source coding theory based on grammar transforms. *Proc. of the 1999 IEEE Information Theory and Communications Workshop*. Kruger National Park. South Africa. June 20-25. pp. 75–77.

[72] P. Yianilos. CDNA source code. `http://www.neci.nj.nec.com/homepages/pny/software/cdna/main.html`.

[73] J. Ziv and A. Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*. 23 (1977) 337-343.

[74] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*. 24 (1978) 530-536.