2010

# Modeling Biological Structures via Abstract Grammars to Solve Common Problems in Computational Biology

David J. Russell
*University of Nebraska - LIncoln*, drussell@engr.unl.edu

# MODELING BIOLOGICAL STRUCTURES VIA ABSTRACT GRAMMARS TO SOLVE COMMON PROBLEMS IN COMPUTATIONAL BIOLOGY

by

David James Russell

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Engineering (Electrical Engineering)

Under the Supervision of Professor Khalid Sayood

Lincoln, Nebraska

October, 2010

# MODELING BIOLOGICAL STRUCTURES VIA ABSTRACT GRAMMARS TO SOLVE COMMON PROBLEMS IN COMPUTATIONAL BIOLOGY

David James Russell, Ph.D.

University of Nebraska, 2010

Advisor: Khalid Sayood

Grammars are generally understood to be the set of rules that define the relationships between elements of a language. However, grammars can also be used to elucidate structural relationships within sequences constructed from any finite alphabet. In this work abstract grammars are used to model the primary and secondary structures present in biological data. These grammar models are inferred and applied to efficiently solve various sequence analysis problems in computational biology, including multiple sequence alignment, fragment assembly, database redundancy removal, and structural prediction.

The primary structures, or sequential ordering of symbols, of biological data are first modeled with Lempel-Ziv (LZ) grammars. The results are used to construct a grammar based sequence distance metric which can be used to compare biological sequences by comparing their inferred grammars. This concept is applied to solve several problems involving biological sequence analysis including multiple sequence alignment and phylogenetic clustering. The higher-level secondary structures of biological sequences are then modeled via two novel grammar inference methods. The resulting context-free grammars are used to estimate structural pieces within biological sequences, which can in-turn be used as supplemental information to help guide various sequence analysis algorithms. The use of this approach to develop algorithms for various sequence analysis tasks demonstrates the viability and versatility of using abstract grammars to model biological data.

# Acknowledgements

It is finally over. Nineteen years after my first class in the 1991 fall semester, I have finally completed the ultimate student achievement. Regarding the dissertation itself, it could not have been completed if it were not for the support provided by the following. First, the University of Nebraska-Lincoln subsidized nearly the entire cost of doctoral tuition. In conjunction, Dr. Jerry Hudgins, Chairman of the Electrical Engineering Department, was gracious in providing me with employment as Lecturer which has been extremely beneficial; in particular, I have maintained full-time employment while on campus with a very flexible schedule. Second, graduate student peer Ufuk Nalbantoğlu provided support through research topic discussions and research paper collaborations, and undergraduate research assistant Sam Way provided technical support in the form of Perl scripts and the GUI for GramContig. Next, the committee members, Dr. Michael Hoffman, Dr. Brian Harbourne, and Dr. Mustafa Gursoy, took the time and effort to review this research and provide many valuable comments and corrections. Acting as advisor, Dr. Khalid Sayood has certainly invested the most time in overseeing my research, followed by thoroughly reviewing this dissertation. No doubt this work would not contain the quality nor quantity if it were not for the demanding expectations of Dr. Sayood.

While this is certainly not a miracle, there have been significant personal events over the past nineteen years that have added to the challenge. The three most important being: marriage to my wife Jamie in 1993, the birth of our first daughter Gates in 1999, and the birth of our twins Gracen and Gavin in 2003. Only Jamie has been by my side for the duration to witness the total effort which has culminated in this dissertation. She has provided unlimited support to me, not the least of which by being a fantastic full-time mother to our kids; and perhaps most important of which by being my best friend.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Imagine Bob, a student of biology. Suppose Bob obtains an organic sample of unknown origin–perhaps he simply went into his backyard and scooped up a handful of dirt filled with many bacteria. Bob is interested in learning more about the sample. Perhaps he would like to know from where each organism came. That is, he would like to determine their ancestry by building a family tree. In doing so, Bob will have identified many known organisms, but there is also a good chance that he will have discovered something new. It turns out there are many bacterial organisms on Earth that have never been classified. These discoveries are important for the general health of other populations, not least of which is our own human species.

Consider the research methods that Bob uses to learn more about his sample. His study begins by using tools to obtain his primary target, which ultimately depends on what kind of biologist Bob is. Assuming Bob is a microbiologist, then his target is the information contained within each living organism or colony of organisms. While the information he is after is quite enormous, his site under study is actually very tiny. The illustration in Figure 1.1 depicts one possible site on the far left-hand-side being a macromolecule called a chromosome. A microbiological target is generally a microscopic piece in a cell within a living organism such as a rose bush, or a chihuahua, or a human. Bob uses a variety of methods and tools to initially clear the

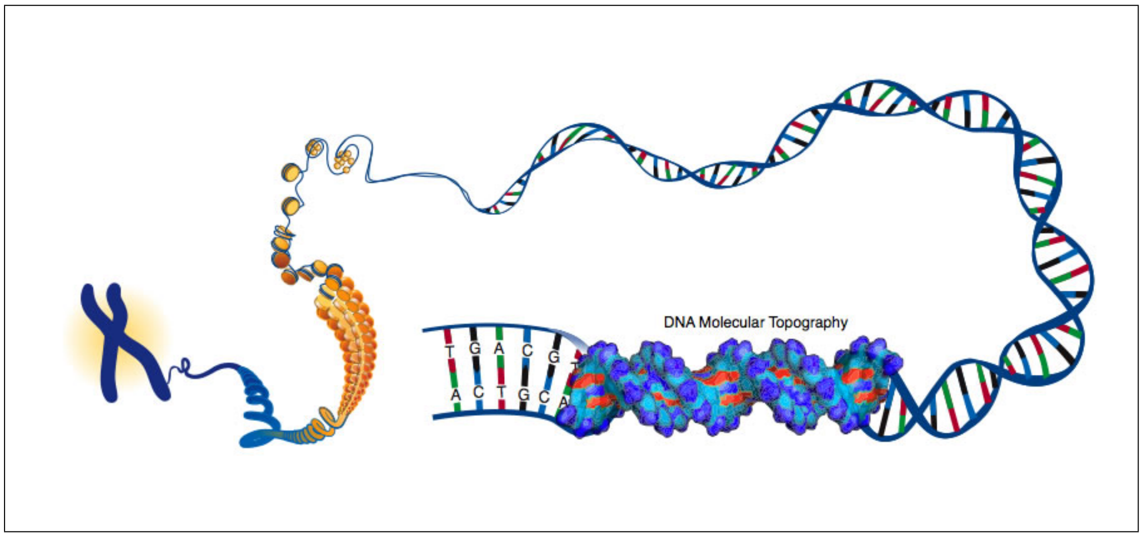


Figure 1.1: An illustration depicting DNA packed tightly into chromosomes, as well as a DNA molecule unwound to reveal its double helix structure. Image freely available from the National Human Genome Research Institute (http://www.genome.gov).

area, making it possible to focus on a specific target. Such tools might include swabs, petri dishes, microscopes, scalpels, etc. Then, more methods and tools are used to refine the target gaining the necessary molecular information; these tools are generally referred to as sequencing tools and they allow Bob the ability to acquire pieces of the target site. Referring to the illustration in Figure 1.1, the pieces that Bob has access to are fragments of the DNA double helix depicted more on the right-hand-side. In many analyses, Bob does not study the organism directly, but uses a schematic representation as depicted at the very end of the illustration in Figure 1.1. Notice how the image changes from a cartoon view of chemical molecules into sequences of letters for the set $\{A, C, G, T\}$. It turns out that DNA macromolecules are composed of only a few specific smaller molecules, and it is the order in which they are chained together and the way in which they form three-dimensional shapes that represents so much interesting information. The sequencing tools mentioned earlier take in biological samples and produce enormous listings of sequences representing small sections of the organism. These fragments are then like puzzle pieces that need to be reassembled in order to have a complete representation of the organism.

This leads to the next stage for Bob, which is analysis of his organism. After organisms are initially sequenced, they are studied to gain as much information as possible. The typical first step in analysis is to perform fragment reassembly, which introduces the first problem of analysis. Bob always faces this issue, as the only current methods available to Bob are tools that generate small pieces representing the DNA molecules of his target site. Once the DNA has been properly reassembled, Bob is able to catalogue and compare his organism to previously published collections in order to classify it phylogenetically. This would allow him to identify other hosts with similar DNA sequences. After which, a wide range of analytical techniques are available in bioinformatic science to analyze organism composition.

One analysis technique presents Bob with an interesting problem. Imagine the work necessary to be the first person to study a never before seen text. It represents the ultimate puzzle in which the only clues are the visual symbols and their physical arrangement upon the artifact. It turns out the sequences of molecules represented by the four letters actually form a language called the genetic code. Within the strands of DNA are regions of words and phrases that may appear alien to Bob, but ultimately spell out the sentences and paragraphs of information necessary for an organism to produce its life-giving proteins.

Bob is faced with millions or billions of symbols. To make the situation more complicated, there are many complex and long-distant relationships within an organism's microbiological functioning. To understand this further consider the two nearly identical english phrases: 1) Time flies like an arrow. 2) Fruit flies like an apple. Now imagine not understanding english at all. The very first problem is recognizing word boundaries. In other words, how would someone that has never before seen an english alphabet identify where words begin and end? Beyond this fundamental issue, consider how the words interact with each other in order to generate the semantic meaning of each sentence. The middle sequential fragment " flies like an a" occurs in both phrases, and so a person who has never seen the english alphabet might focus on this similarity. However, anyone understanding english recognizes that the word "flies" changes meaning due to the surrounding words–its context. DNA sequences exhibit similar behavior, including the initial problem of identifying word boundaries. However, Bob's case is made more difficult because there are no special "space" characters to explicitly identify where one word ends and the next one begins. Imagine trying to read the text on this page with all the spaces removed. Beyond this problem, the genetic words and phrases interact with each other to change their meaning, analogous to the word "flies" in the english phrases above.

Many applications involving sequence analysis are based on understanding the source mechanism from which the sequence was generated. Grade school students often learn how to diagram a sentence to gain greater understanding of english grammar, which are the collective rules governing the english language. For example, in the case of the two english phrases, students are able to categorize the words as in:

- Time flies like an arrow. (noun, verb, preposition, indefinite article, noun)
- Fruit flies like an apple. (plural noun, verb, indefinite article, noun)

Once diagrammed, it is a little easier to see how the words behave together to form a larger meaning. If Bob knew the grammar governing the genetic language of his organism, he could diagram the DNA sequence in order to find out exactly what is being said. Unfortunately, the genetic language and its underlying grammar are generally unknown. If there were some way to derive an estimation of the grammar given a small sample of Bob's sequence, then perhaps the resulting approximation would be useful in subsequent analysis for the rest of the DNA sequence, kind of like fitting a curve to a histogram of data samples in order to better predict some unknown source behavior.

## 1.1 Contribution

The previous discussion represents the primary objective of this work. In this dissertation we utilize the information-theoretic tool called an abstract grammar to model biological data. These grammar models are inferred and applied to efficiently solve various sequence analysis problems present in computational biology.

We begin with a pre-existing method for estimating a grammar based upon a classic text-based dictionary compression scheme. The resulting grammar models for each sequence are used to create a relative distance metric. Then, comparing the similarity of two sequences amounts to comparing their grammar rules resulting in efficient tools for performing sequence analysis, including multiple sequence alignment, relative fragment assembly, and sequence clustering for the purpose of removing redundancy within a dataset. Second, we turn our attention from the grammar models that operate on the sequential ordering of elements to grammars that model longer distance relationships within DNA sequences. We describe two novel grammar inference algorithms that are able to estimate a more complicated level of grammar called a context-free grammar. The first method is a polynomial-time framework capable of modeling the three-dimensional molecular shapes that result due to mechanical folding of the sequential strands. The second method improves upon the first by reducing the order of execution time from polynomial to linear. Again, the result is able to model the secondary structures responsible for the complex folding interactions. The circle is completed by applying the structural models of the final grammar inference algorithm to the initial multiple sequence alignment application. The preliminary results validate the overall alignment quality improvement after using the higher-level grammar-based model information.

## 1.2 Organization

This dissertation is organized as follows. Chapter 2 provides general background information covering computational biology, problems in bioinformatics, and an introduction to grammars. Chapters 3 and 4 contain applications of LZ grammars on several bioinformatics problems. In Chapter 3, an inferred LZ grammar is used to form a distance metric that determines the order in which sequences are progressively aligned. The chapter concludes with a modification that allows for fragment assembly against a known reference sequence. In Chapter 4, the problem of efficient data clustering is described. In particular, an inferred LZ grammar is applied on large sets of sequence fragments with the intent of classifying similar sequences within clusters that are represented by a single sequence. Chapter 5 and 6 propose two new methods for inferring CFGs for DNA or RNA sequences making use of the Chargaff rule domain knowledge. Both methods are designed to infer the secondary structure present in the corpus of data, thus capturing information not available in the LZ grammars. Chapter 5 details a polynomial-time algorithm based on a classic string classification method, CYK. Chapter 6 presents a linear-time algorithm based on the recent Sequiter algorithm. The chapter finishes by modifying the multiple sequence alignment application in Chapter 3 with the application of the secondary structural information during the alignment process. Chapter 7 concludes this dissertation with remarks on future research.

# Chapter 2

# Fundamentals of Computational Biology

The relatively new field of *bioinformatics* tends to have a nebulous coverage of topics. In fact, the two seemingly distinct branches of science, chemistry and biology, have sub-categories dedicated to their own version of studying problems in bioinformatics. Crudely speaking, chemists tend to focus on the very low-level, chemical structures and functions of various biologically-important macromolecules, including sugar, fatty acid, nucleotides, and amino acids [17], while biologists focus on the interactions and regulation of proteins and underlying nucleic acids. This distinction is not very sharp and there is considerable overlap between the two disciplines. Various members belonging to the fields of mathematics, statistics, computer science and electrical engineering have also become interested in solving many problems that affect the ability of chemists and biologists to perform their research. The reason for this seemingly divergent meshing of groups stems from the gigantic amounts of information stored in the macromolecules of interest. That is, the genetic code. It turns out the blue-print to each living organism is held within itself in the form of chemistry-based macromolecules. The schematic used to model the information contained in the genetic code tends to be enormous–on the order of millions of text symbols. Born from these concepts is *computational biology*, synonymous with bioinformatics, which is concerned with all the problems that occur after genetic information has been obtained.

This chapter begins with a brief introduction to the fundamental components and processes underlying the bioinformatics applications discussed in subsequent chapters. This is a presentation of the core vocabulary in addition to the basic principles in going from low-level chemistry to the higher-level computational realm–operating on sequences of symbols. A balanced approach is taken to present the required concepts without delving too deeply into details that are beyond the scope of this work. Once navigation from molecules to sequences is complete, a summary of typical bioinformatics problems is reviewed followed by an introduction to the necessary terms and operations that define a concept from information theory called abstract grammar. The subsequent chapters are dedicated to solving some of these problems using grammars.

## 2.1 Biochemistry Background

Biochemistry is a large topic to cover, and is really beyond the scope of this text. However, there are some fundamental terms, concepts and processes that deserve an introduction. The material presented in this section was culled from [17]; more details could be found there or in other texts on biochemistry.

All living organisms on Earth are principally made from Carbon and its characteristic strong covalent bonds. In fact, Carbon has four electrons in its outer shell that allows four very strong covalent bonds with various other atoms. As a result many molecules that form the basic components of living organisms are composed largely out of Carbon atoms covalently bonded with combinations of Nitrogen, Oxygen, Hydrogen and Phosphorus. It turns out the study of Carbon compounds is so important that it forms a branch of Chemistry called *Organic Chemistry.* One of the most relevant Carbon compounds is a *sugar* which is a carbohydrate with the generic formula $C_m(H_2O)_n$ containing either an aldehyde group (CHO) or a ketone group ($C = O$). The primary sugar molecule of interest, schematically shown in Figure 2.1, is a five Carbon sugar with a ketone group, called *ribose.* If the Oxygen atom is removed from

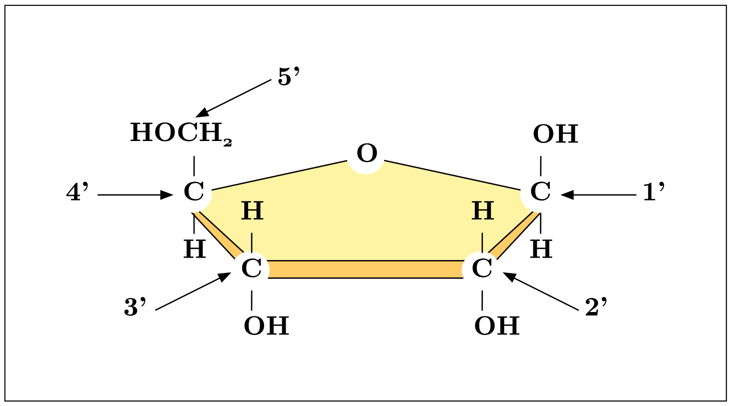


Figure 2.1: A schematic of the cyclic form of the ribose sugar molecule. The five Carbon atoms present in the ribose molecule are numbered from 1' to 5' beginning with the Carbon belonging to the ketone group.

the 2' Carbon, the resulting sugar molecule shown in Figure 2.2 is called *deoxyribose.* The ribose and deoxyribose sugar molecules provide the 'R' and 'D' to the portion of the monomer called a nucleotide. A *monomer* is an atom or small molecule that may chemically bind with other monomers to form a larger molecule called a *polymer.*

### 2.1.1 Nucleotides

*Nucleotides* are the molecular building blocks of the larger polymers deoxyribonucleic acid (DNA) and ribonucleic acid (RNA). That is, nucleotides are themselves polymers chained together forming the much larger polymers of DNA and RNA, also called *oligonucleotides.* As seen in Figure 2.3, a *nucleoside* is formed by bonding one of a group of *nucleobases*, or just *bases*, to the 1' Carbon of a sugar molecule (either

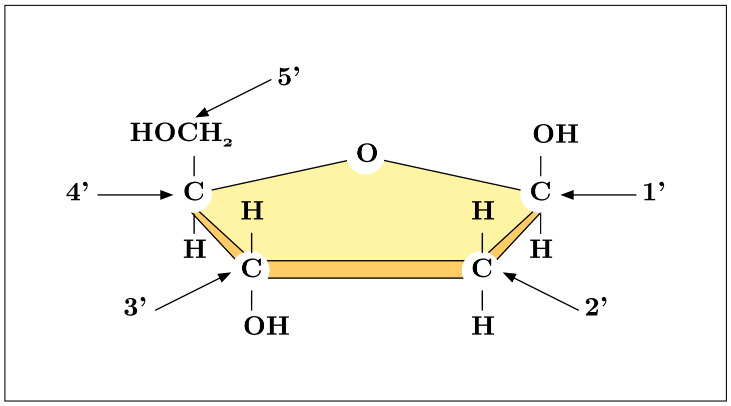


Figure 2.2: A schematic of the deoxyribose sugar molecule.

ribose or deoxyribose). A nucleotide is formed by taking the nucleoside and bonding a phosphate group to the 5' Carbon of the sugar.

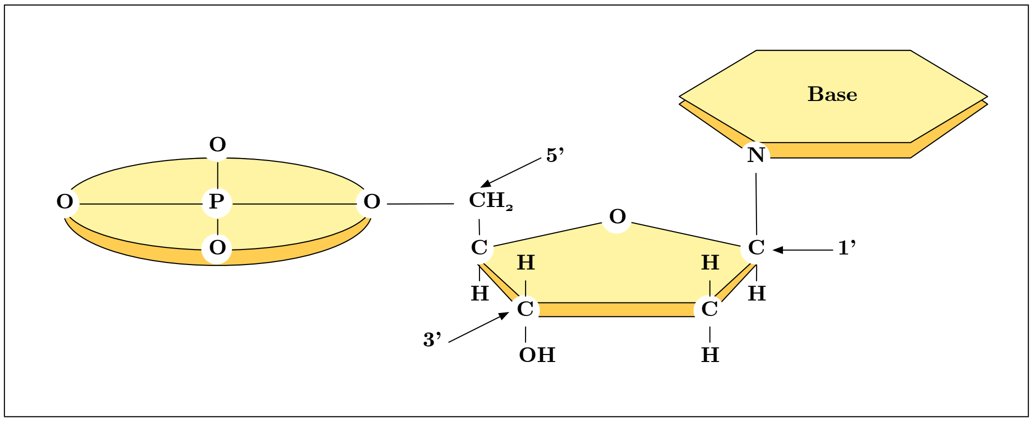


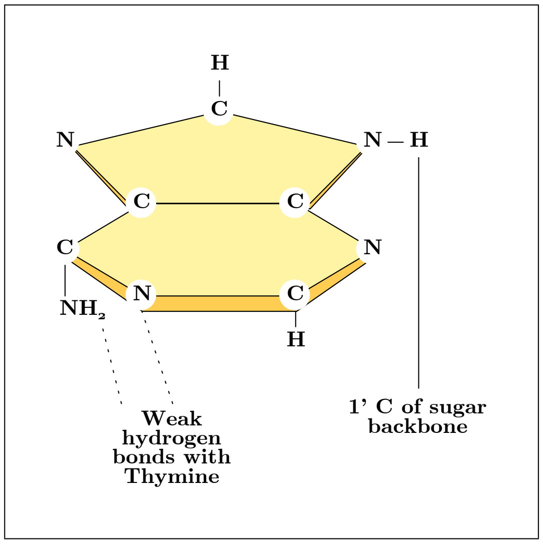
Figure 2.3: A schematic of the nucleotide polymer. The nucleotide is formed by bonding a phosphate group to the 5' Carbon and a base to the 1' Carbon of the sugar molecule.

### 2.1.2 Bases

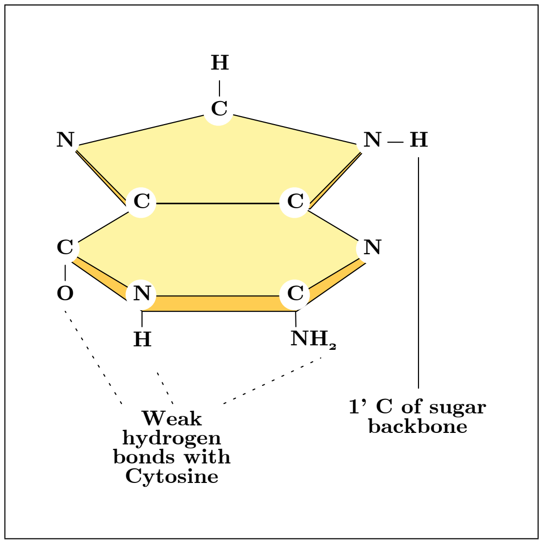
*Nucleobases*, or just *bases*, are Nitrogen/Carbon ring molecules and act as a key component to the functionality of DNA. They are the primary source of structure and functioning behind the famous DNA double helix shape and the method by which

DNA is replicated. However, they serve other interesting purposes. Perhaps most important is their role as lexicon to the entire construction of their host organism. That is, the four different base molecules of DNA act as an alphabet of symbols that, when combined, form larger phrases used to define many larger components of an organism. While there are several layers of phrases that one could consider, the most recognizable characterization is that of a gene which will be discussed shortly.

Four of the five base molecules are shown in Figures 2.4 and 2.5. These four bases



(a) Adenine

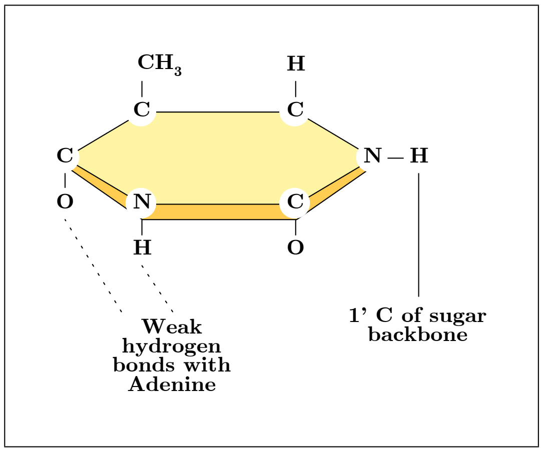


(b) Guanine

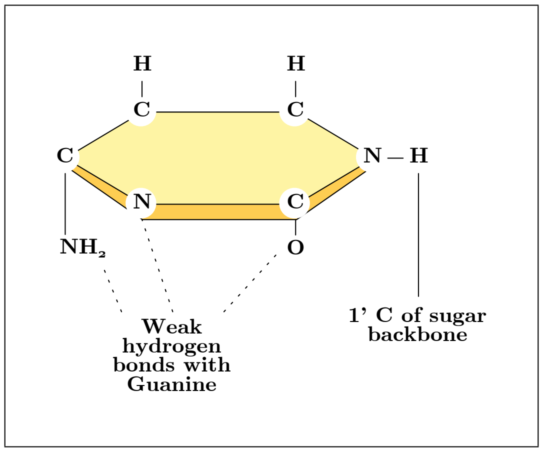
Figure 2.4: Schematics of the two different DNA base molecules belonging to the purine family.

are those used to bond with the deoxyribose sugar in order to build a DNA nucleotide. In RNA, a fifth base, Uracil, replaces Thymine although it acts structurally similarly. As a result, in many high-level bioinformatics problems, Uracil and Thymine are treated as the same element corresponding to the respective alphabet.

Referring to Figure 2.4, Adenine and Guanine having two ring structures belong to the family of *purines* while Cytosine and Thymine shown in Figure 2.5 have single ring structures, and belong to the family of *pyrimidines*. These bases are what keep the two strands of the DNA together via weak hydrogen bonding. In particular, the polarity of the bases relative to how they bond with the 1' Carbon of the sugar leads to only two pairings that really bond well. The *Chargaff rules* refer to these two pairings whereby Adenine forms two hydrogen bonds with Thymine and Cytosine forms three hydrogen bonds with Guanine.



(a) Thymine



(b) Cytosine

Figure 2.5: Schematics of the two different DNA base molecules belonging to the pyrimidines. In RNA, the base Uracil replaces Thymine. Uracil is able to act structurally similar to Thymine because its schematic is a modification of Thymine in which the $CH_3$ group is replaced with H.

### 2.1.3 Phosphate Group

A chain of three phosphate groups, one of which is shown in Figure 2.6, provide the necessary energy to connect nucleotides together to form an RNA or DNA polymer. In general, two nucleosides are chained together via a single phosphate group attached at the 5' Carbon of one sugar and the hydroxyl group at the 3' location of the second sugar. The entire RNA or DNA molecule begins with a single nucleotide in which a small chain of three phosphate groups is bonded to the 5' location. Two of the

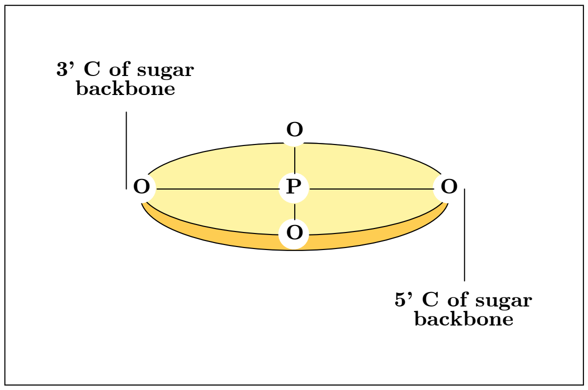


Figure 2.6: A schematic of the phosphate group. Three of these are chained together to provide the necessary energy for DNA and RNA molecules to grow. Once two of the three phosphate groups break off, the remaining phosphate group acts as an interlink between subsequent sugar molecules in the sugar-based backbone of DNA and RNA.

phosphate monomers break away from the polyphosphate group in order to release enough energy so the remaining phosphate group can create a bond with the hydroxyl group at the 3' location of another nucleotide. DNA and RNA are therefore grown from a 5' end towards a 3' end. Thus, DNA and RNA have directionality, much like written text.

### 2.1.4 DNA

A single strand of DNA grows in the 5'-3' direction with the addition of each nucleotide. Because of the mechanical structure of the nucleotides, each time another is added to the chain a slight rotation occurs in physical space. The result is the characteristic helix spiral of DNA. Two strands of DNA join with each other through weak hydrogen bonding between their bases, as shown in Figure 2.7. One half of a DNA molecule is a very long strand of alternating deoxyribose sugars and phosphate groups. Often this strand is referred to as the sugar-based backbone, and it is the foundation upon which the host organism's genome is written. The *genome* contains

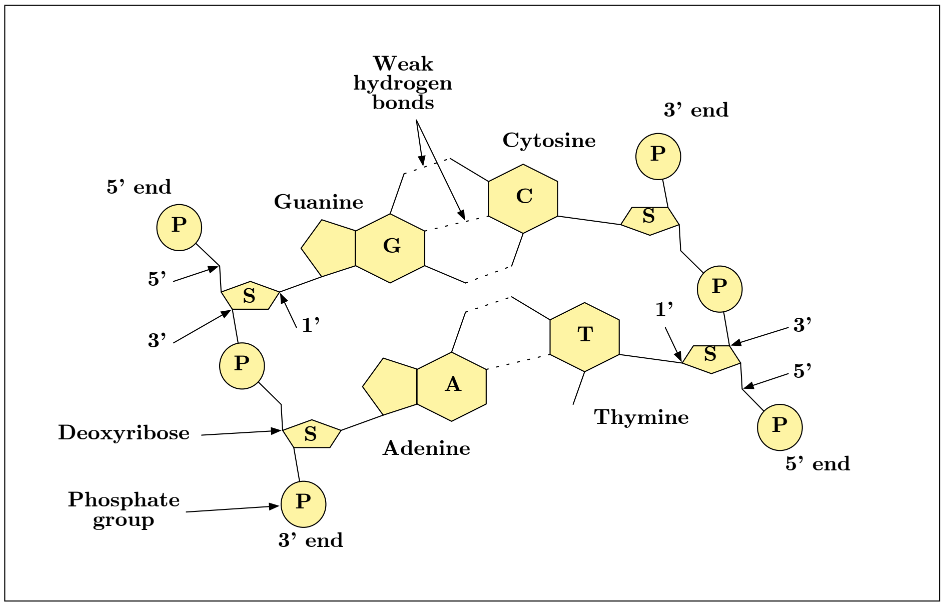


Figure 2.7: A schematic of a DNA strand. Two sugar-phosphate backbones provide the foundation necessary to hold the base molecules which store the genetic code of the host organism. The two backbones are weakly attached together via the Adenine-Thymine and Cytosine-Guanine hydrogen bonds.

the entire set of instructions used by the organism's biological machinery in order to produce all the pieces necessary for survival, including the machinery itself! This genetic code is stored via different sequential patterns of base molecules attached to the 1' Carbons of each sugar molecule on the backbone.

The structure of a DNA molecule is reinforced by the presence of a second sugar-based backbone containing what appears to be completely different sequential patterns of base molecules hydrogen bonded with the bases on the first backbone. However, due to the Chargaff rules, the sequence on the second backbone needs to be exactly paired to the first backbone. Additionally, as can be seen in Figure 2.7, the 5' end of the second backbone is located at the 3' end of the first backbone. As a result, the second set of sequential patterns is said to be the *reverse complement* of the first set. More accurately, each set is the reverse complement of the other.

Notice that the information of an entire DNA molecule is specified so long as mutual exclusive segments of each strand are known. Thus, the double strands not only provide increased robustness in physical structure, but also an information-based error correction system as well.

### Replication

It turns out that many microscopic components of all living organisms are present for the purpose of *replication*, the process of copying existing DNA molecules into new DNA molecules. Because DNA molecules are physically wound into helices, the first step in performing replication is to unwind and separate the strands. An enzyme called *Helicase* is responsible for separating the two strands by temporarily breaking hydrogen bonds, the result of which is depicted in Figure 2.8. As it takes

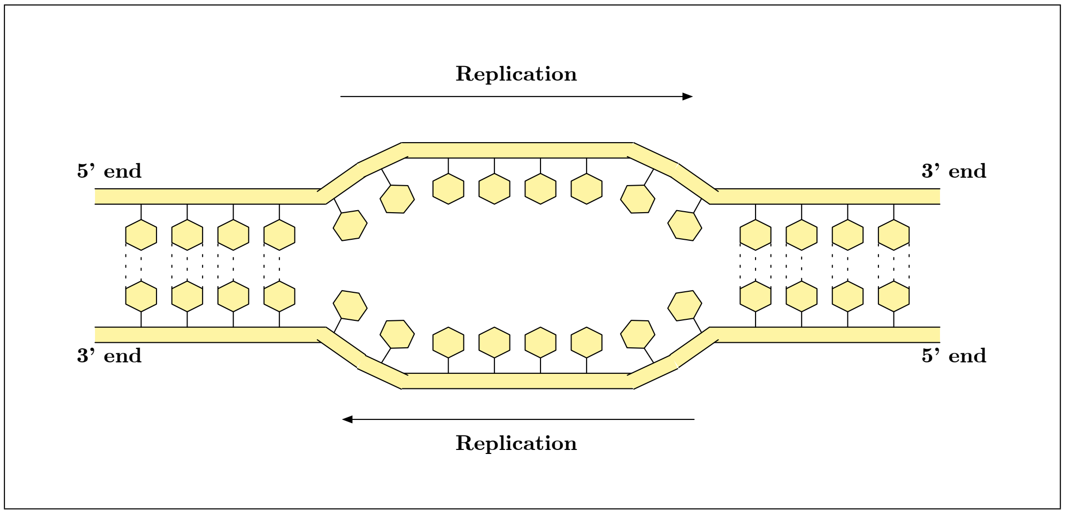


Figure 2.8: A schematic of a DNA strand prepared for replication whereby Helicase has unwound and separated the two strands.

energy to break the bonds, weaker regions often define locations where replication begins. Because Adenine-Thymine pair with only two bonds, compared to Cytosine-Guanine that pair with three bonds, the origin of replication is often "AT" rich, meaning the sequence of bases on either strand will have an increased proportion of Adenine or Thymine. After the separation is accomplished, a physical wedge of single strand binding protein is shoved between the strands at the replication fork. Once stabilized, either strand may be copied after an *RNA Polymerase*, or just *Primase*, enzyme attaches a *primer* RNA fragment to some specific complementary sequence of bases. Next, *DNA Polymerase* enzymes begin replication by first attaching to the primer RNA fragment and then moving along the bases in the 5'-3' direction, outputting two copies of the DNA stand. Eventually, the primer is stripped away from the original DNA bases by another enzyme called an exonuclease.

**Structure**

Increasing in organizational hierarchy, DNA is generally organized into *chromosomes*. In *prokaryotes*, organisms without a cell nucleus, there is usually only a single chromosome with a circular structure. On the other hand, *eukaryotes*, organisms with a definite cell nucleus, contain multiple linear chromosomes inside each nucleus.

As stated in the previous section, organisms contain the necessary machinery (e.g., enzymes) to perform DNA replication naturally. It turns out that several processes have been developed in order to perform DNA replication artificially. For example, in a process called Polymerase Chain Reaction (PCR), DNA is repeatedly heated and cooled within a mixture containing both specific primers and individual nucleotides. After the DNA is heated the strands separate, followed by cooling at which time the bases attempt to reestablish their previous bonds. However, the presence of the primers and nucleotides and the addition of DNA Polymerase allow artificial replication to take place. This is just one example of many different processes available that allow researchers the ability to discover the order in which bases occur on either strand. Other common techniques include *shotgun sequencing* and *pyrosequencing*.

In general, chemists and biologists have the ability to analyze organisms with the goal of recording the entire sequential order of base molecules–its genome. This is one of the primary transitioning points between low-level chemist/biologist and high-level modeler, as genomes can be many millions of bases in length. The result is often a need for efficient data searching and storage as well as a desire for accurate models to represent or predict the information.

### 2.1.5 Genes

The genome, or collection of genes, contains the entire enumeration of bases along the chromosome or set of chromosomes; much like an encyclopedia contains a specific enumeration of alphabetic characters along each volume. While the order of letters ultimately matters to the meaning of text, it is difficult to understand passages without zooming out a little, where words and sentences can be analyzed from a higher-level context. This is true of a genome as well; bases are analogous to letters of an alphabet, and genes are analogous to sentences or paragraphs of prose. The following sections introduce vocabulary and concepts important to understanding genes.

#### Proteins

*Proteins* are another class of polymers and composed of a sequence of *amino acid* molecules. Amino acids are Carbon compounds with the generic structure shown in Figure 2.9. A central Carbon atom called the $\alpha$Carbon is connected to an amino group ($NH_2$), a carboxyl group (COOH), a hydrogen atom, and a *residue*. The residue is one of several significantly different molecular chains, and is responsible for the chemical behaviors between various other amino acids. Based on the structure of the residues, amino acids can fall into different categories including polar/non-polar, acidic/basic, hydrophilic/hydrophobic, etc. While there are more than 100 amino

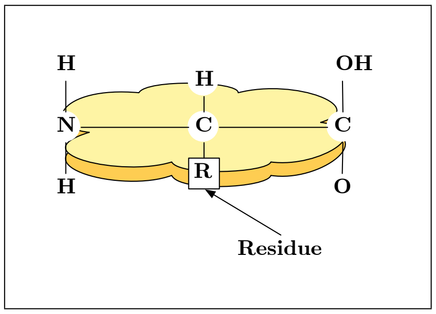


Figure 2.9: A schematic of the general structure for an alpha amino acid. The central $\alpha$Carbon bonds with a residue which is one of several different molecular groups that define the chemical behavior of an amino acid.

acids in nature, only 20 are used by DNA to form proteins. Some of the 20 amino acids can by synthesized by the host organism while others cannot and so need to be introduced externally.

Chemical reaction between the amino group of one amino acid with the carboxyl group of another amino acid forms a *peptide bond*. Proteins are formed when a sequence of many amino acids come together via peptide bonds. Thus proteins are also known as *polypeptides*. Interestingly, DNA molecules and protein molecules present similar sequential models. That is, DNA molecules are composed of a chain of monomers along a sugar-based backbone, each of which can be one of four different molecules. Similarly, protein molecules are composed of a chain of amino acid polymers along a peptide backbone, each of which can be one of twenty different molecules. Both macromolecules are thus recorded in databases as a sequence of alphabetic characters, where each letter is meant to represent a specific base or residue.

Proteins are essential to the metabolic and structural functions of organisms. For example, the enzymes involved with replicating DNA are all proteins. Proteins are generally responsible for all aspects of how organisms are able to function at a molecular level. Additionally, proteins form the structural elements of an organism's body, and so are responsible for how a life-form appears. Proteins are truly the building blocks of life.

Proteins are complex, three-dimensional structures. As a result, the sequential order of amino acids is not the only significant aspect to proteins. Many times, long distance relationships are important resulting in the polypeptide mechanically folding many times, forming interesting shapes that are useful in allowing the protein to function. Just as words in a sentence can affect phrases that are far apart from each other, so can amino acids in one area affect those in another area. That is, not only are subsequent relationships important, but more complex local and even long-distance interactions can be equally important. Because of the relationship between the structure and function of proteins, a great deal of effort has gone into the study of their organization. The sequential order due to the covalent bonding of amino acids along the peptide backbone is referred to as the *primary structure*, while the local and long-distance hydrogen bonding is referred to as the *secondary structure* and *tertiary structure*, respectively. It turns out that proteins with related functionality tend to have similar tertiary structures. An even higher-level structure exists called *quaternary structure* in which multiple proteins combine into a multi-polypeptide unit.

### Genetic Code

As has been alluded to, proteins and DNA are related through the genetic code. In particular, DNA may be thought of as an ordered list of instructions for building proteins out of the amino acid components. In fact, a *gene* is defined as the portion of DNA that contains the information necessary for the generation of a particular protein. Note however, because there are only four letters in the DNA alphabet compared to 20 different letters in the relevant amino acid alphabet, it is necessary that runs of DNA bases be used to determine the addition of a single amino acid. A *codon* is a three-base DNA segment leading to a many-to-one relationship between DNA codons and amino acids. In fact, other than Methionine and Tryptophan, all other amino acids are represented by multiple codons. If $n$ codons correspond to the same amino acid, it is said to be $n$-fold degenerate. Interestingly, the presence of a many-to-one relationship introduces yet another form of error control between the genome and the produced proteins. That is, if a mistake were to occur during the production of a protein, there is a possibility that it will not be fatal.

**Transcription**

The first step in producing a protein from DNA is called *transcription* or *gene expression.* The RNA Polymerase enzyme performs the work of constructing an appropriate messenger RNA (*mRNA*) which ultimately forms the template upon which the protein is constructed. Transcription begins with an initiation stage in which the RNA Polymerase attaches to a *promoter* region that is sequentially prior to the gene being expressed. That is, the promoter region is closer to the 5' end of the strand and is said to be upstream so that it can flow towards the 3' end whereby it will move across the gene of interest. While promoter sites vary in terms of content and location from one gene to another, there are two highly common, six-base subsequences in promoters. These occur around 10 bases and 35 bases upstream of the gene being expressed. The RNA Polymerase works through an elongation process in which it flows toward the 3' end while unwinding the DNA downstream and rewinding the region upstream behind it. As it flows along, it constructs an mRNA strand by attaching complementary nucleotides to the currently unwound region. Eventually the process terminates due to mechanical stopping conditions. In one condition, a physical *hairpin* forms in the mRNA where a local segment of mRNA folds over and bonds with a reverse complement region nearby. The resulting structure is such that it physically forces the RNA Polymerase to halt. Another condition is due to a second enzyme that has the ability to interfere with the RNA Polymerase if it slows down too much.

### Translation

Since the RNA Polymerase constructed a complementary strand of mRNA by flowing across a region of DNA containing the coding information for a gene, the mRNA strand contains the same information necessary for constructing the corresponding protein. The mRNA fragment is carried to the ribosome which consists of many proteins and RNA molecules. As shown in Figure 2.10, once the translation pro-

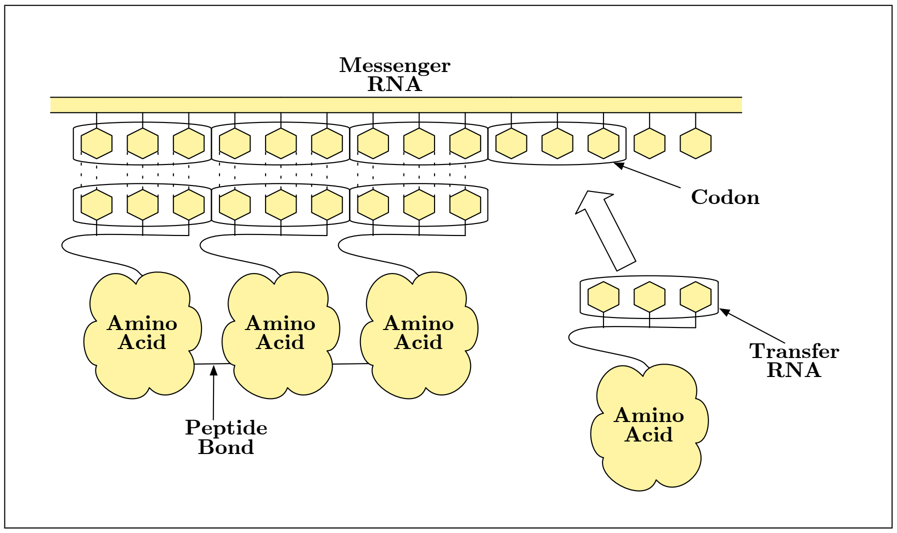


Figure 2.10: Translation of messenger RNA into protein takes place in the ribosome. The mRNA acts as the template upon which transfer RNA anti-codon tags are attracted and attached to their corresponding codon location. The tags are connected to an amino acid. So a chain of amino acids are placed next to each other, at which time peptide bonds form resulting in the final polypeptide.

cess begins, the ribosome moves along the mRNA to manufacture the protein. The translation of genetic information presented in the mRNA to a protein is carried out with the help of adaptor molecules called transfer RNA (*tRNA*). These molecules are about 75 nucleotides long with an amino acid attached to the 3' end. The tRNA fragment contains multiple complementary sections that cause it to take on a cloverleaf configuration. The three bases complementary to the codon for the particular amino acid attached to the tRNA are located at the tip of one of the leaves. This triplet is often referred to as the anti-codon. Once the amino acids are placed, a peptide bond forms between neighbors ultimately resulting in the final polypeptide.

### 2.1.6 Genomic Variation

The preceding sections have presented a streamlined version of many fundamental processes that take place between DNA molecules and the proteins for which they code. Perhaps in a perfect environment processes such as replication, transcription and translation would all occur without any errors. However, it is clear that these tasks are performed at the molecular level without any discernible sentient control to govern the results. In fact, this is quite remarkable and wonderful. Yet any of these processes is prone to occasional mistakes that may lead to stable genomic variation, which in turn drives the process of evolution. Actually, these variations might not be thought of as mistakes, but adaptations to the organism's environment, the result of which allows the organism to thrive. On the other hand, some stable mistakes are also the source of several debilitating diseases. Genomic variation can occur through mutation, recombination and horizontal transfer of genetic material.

Genomic mutations are changes that occur to bases in DNA strands. These changes may occur at a single site, where one base gets changed to another in a *point mutation*, or a base gets introduced or deleted, the result of which is generically referred to as an *indel*. Changes may also occur as a result from the transfer of a segment of genetic material. Sometimes mutations are either neutral or beneficial, the result of which will generally be passed on to future generations. Mutations that persist in more than one percent of the population are called *polymorphisms*.

Genomic recombination occurs when a fragment of DNA or RNA erroneously joins in an incorrect location or with an incorrect complementary piece. Similarly, in bacteria, horizontal gene transfer may take place in which a segment from one genome is erroneously copied into some other circular genome. In particular, one organism can receive genetic information from another organism without being its descendant.

Now that many of the fundamental concepts and vocabulary terms have been presented, the next section introduces several current problems that face the field of computational biology.

## 2.2 Common Problems in Computational Biology

As bioinformatics is a relatively new field of research, the list of typical problems is somewhat dynamic. However, there are several core topics that continue to deserve attention, or have recently developed due to past research. Perhaps in the coarsest sense, there are two categories of computational tools used by bioinformatics researchers: modeling tools and workload tools. The latter class of software tools are developed with the intent of simplifying or reducing the amount of rote work that exists due to the nature of the massive amounts of biological data present in databases. These tools often provide modern implementations of classic computer science solutions for various database problems, and include work done in the recently developed field of data-mining. In contrast, the effort put forth in developing modeling tools involves much more focus on successful analysis of a system in order to create realistic representations that allow researchers the ability to predict behavior. A brief introduction to a few sub-categories of problems is presented in the following sections.

### 2.2.1 Homology Search

Homology searching is the process of finding similar regions between multiple sequences. In particular, *homologs* are regions within proteins related via evolution from a common ancestor. The importance of this task lies in the theory that known information, either structural or functional, about one sequence implies information about the other sequence *by homology* [23]. Much work is put forth discovering homologous sequences in order to determine the function of a new gene, identify additional members of an existing family of proteins, or locate the position of similar genes in different, yet related, organisms.

One of the earliest methods for homology searching is the dynamic programming algorithm presented in [70] which forms a global alignment between a pair of sequences. The goal of a global alignment algorithm is lining up two sequences parallel with each other in such a way that the position of each sequence matches that of the other. Spaces are inserted in order to account for indel mutations. An early modification to the algorithm created an overall alignment based on the combination of several local alignments of sub-sequences. The result allows for higher accuracy in regards to identifying homologs. The first version of this modification was presented in [93], followed by heuristic methods including [81] which defines the very common FASTA file format, and BLAST (Basic Local Alignment Search Tool) in [3] which is probably one of the most prevalent tools used in bioinformatics. As of 2004, the National Center for Biotechnology Information (NCBI) BLAST server for homology searching was queried over 100,000 times a day [11]. Additionally, tools have been developed to form various models of sequences in order to aid in the speed and accuracy of homology searches. One popular method forms a profile hidden Markov model (HMM) as discussed in [24], in which position-specific scores are used for database searching.

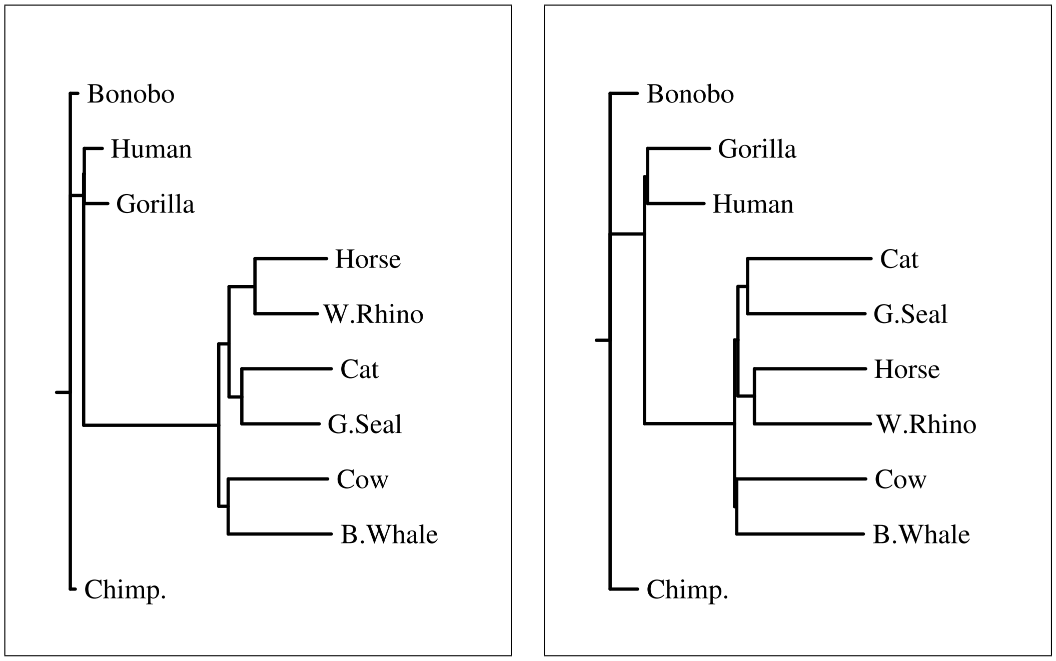
Most recently, research has been focused on managing homology searches over whole genomes, which will necessarily present an enormous computational problem. Thus, work such as [63], [57], and [48] presents novel methods for faster database searching to counteract the growing size.

### 2.2.2 Genomic Annotation

Genomic annotation is the process of identifying and labeling genes and their position within a DNA sequence. This category of software includes tools used to perform sequence assembly, genetic mapping, as well as genetic annotation. When an organism needs to be analyzed, its initially unknown DNA is processed by sequencing techniques that, by present-day technology, are unable to output the entire sequence as a whole. Instead, the various sequencing methods provide small sub-sequences called *fragments*. Computational tools that perform sequence assembly take the set of fragments as input and attempt to recreate the original DNA strand as an output. Challenges occur in sequence assembly due to gaps occurring in the set of fragments where some portion of the original DNA strand was not successfully represented, or multiple fragments of the same region will occur. Genetic mapping and annotation algorithms attempt to identify the position and function of important regions within the DNA strand. These regions include genes and promoter sites, among other interesting genetic features. The results of these tools is typically stored in various databases, usually made available to the research community (e.g., NCBI database).

### 2.2.3 Computational Evolutionary Biology

Computational evolutionary biology is the study of *phylogeny*, the evolutionary relatedness among groups of organisms. Since descendent organisms necessarily evolve from a common ancestor, evolution is naturally seen as a branching process. Thus, phylogenies are often visualized with the aid of a *phylogenetic tree*. As an example, the evolutionary relationship depicted in Figure 2.11(a) contains ten different organisms whose relationship was estimated by generating a multiple sequence alignment of the 12s RNA gene from mitochondria found in the respective cells of the following host organisms: D38113 Chimpanzee, D38114 Gorilla, D38116 Bonobo (Pygmy Chimpanzee), U20753 Cat, V00654 Cow, V00662 Human, X72004 Grey Seal, X72204 Blue Whale, X79547 Horse, and Y07726 White Rhino. Figure 2.11 contains two



(a) Neighbor-Joining (b) Grammar

Figure 2.11: Example phylogenetic trees of the mitochondria found in various eukaryotes. The distances among the mitochondria found in the various host organisms generated using `DRAWGRAM` [31]. The distances were discovered using (a) neighbor-joining clustering on a multiple-sequence alignment and (b) grammar-based distance.

phylogentic trees both linking the various organisms together. The path-length from any organism to any other organism is meant to indicate the evolutionary difference between those organisms. Thus, the trees depicted in Figure 2.11 implies that Gorilla and Human are similar, while Gorilla and Cat are much more distant.

The first tree in Figure 2.11(a) is typical in that the distances were estimated by first creating a *multiple sequence alignment* (MSA) followed by a clustering algorithm called Neighbor-Joining [86]. An MSA is similar to the global and local *pair-wise alignments* used in the homology searching applications with the exception that more than two sequences are aligned along each position. MSA algorithms are the topic of Chapter 3, in which a grammar-based distance metric from [7] and [78] is introduced as the distance metric for a new MSA algorithm. The second tree in Figure 2.11(b), is nearly identical to the first, demonstrating the viability of the grammar-based distance metric from [78] used to estimate the phylogeny.

### 2.2.4 Protein Structure Prediction

The 3-dimensional shapes of proteins often affect their behavior and functionality. That is, the secondary and tertiary structures are often just as important as their primary structure. Of course the higher-layers of structure are ultimately dependent upon the low-level primary structure. As a result, a great deal of research effort continues to go into the task of modeling the various folding and chemical bonding that takes place as a result of the primary structure of the protein sequences. For example, [49] utilizes position specific scoring matrices to guide the protein prediction modeling which then feeds a neural network. These matrices are often used in many aspects of protein analysis. They often represent the log-likelihood of two amino acids being aligned with each other in an effort to apply statistical likelihood of point mutations occurring. Neural networks are often used in current protein structure prediction research, for example [66] and [65].

### 2.2.5 Genetic Regulatory Networks

Genetic regulation is the entire process by which an organism's biochemical machinery scans DNA for promoter sites, performs transcription followed by translation. Genetic regulation is made more complicated by the fact that the machinery performing promotor-site identification, transcription and translation are themselves various proteins that are typically formed via genetic regulation. Further, regulation is a spatial process where proteins floating in the soup of the organism need to chemically bind themselves to the DNA molecule at or near a genetic region, and then move along the DNA strand generating mRNA molecules of the gene which are the pieces necessary for the translation into the intended proteins. However, this ideal description is based on a vacuum of interference. That is, organisms at the molecular level are often dense with material such that the proteins required to bind to the DNA molecule may not do so because other objects physically block their path. In some cases, this is unintentional behavior, and may be thought of as natural noise within the process of the system. However, current research is showing more and more evidence of intentional behavior of the system to either impede or amplify the generation of the proteins described by genes in the DNA. These intentional behaviors are usually realized via other proteins found elsewhere in the organism and also generated by genetic regulation. This fascinating paradox of protein-and-the-gene is controlled by what is now referred to as a Genetic Regulatory Network (*GRN*).

### 2.2.6 Functional Genomics

Identification of function and/or meaning of segments of biological sequences remains an ongoing and active area of research called functional genomics. This work is accomplished primarily by analyzing the genes, the resultant proteins, their individual functionality, and their interaction with other proteins, called protein-protein interaction (PPI). Information is gathered and interpreted by studying the proteins expressed by an organism in addition to the mRNAs produced due to transcription. The typical technique used to acquire these sets of information is via DNA *microarray* experiments. A *chip* is a DNA microarray consisting of a matrix of thousands of *probes* which are known specific DNA fragments. Because of the GRN, the organism to be studied can be applied to the chip under various environmental conditions resulting in various amounts of protein expression. The microarray experiment actually captures the amount of mRNA which occurs during transcription and will bond with the probes on the chip. After the experiment, the microarray contains amounts of material that correspond to the levels of transcription, which then implies the amount of respective protein production. Like many other problems in bioinformatics, the output of these experiments can be enormous in their amount of data. Further, analyzing the results can be a challenging task due to the size of data as well as the fact that complex GRNs may ultimately affect whole systems of protein behaviors.

## 2.3 Grammar

As well be shown, the contributions made by this work are centered upon a family of modeling tools from information theory called grammars. Necessary concepts for understanding how a grammar model is specified are briefly reviewed in this section. In general, standard mathematical notation as found in a typical text on automata theory is followed (see, for example, [46]).

### 2.3.1 Language Terminology

An *alphabet* $\Sigma$ is a finite, nonempty set of symbols from which finite-length sequences, or *strings*, are formed. Strings are constructed via the binary operation of *concatenation* which begins with a copy of the left string and appends a copy of the right string. Notationally, the formal symbol $\cdot$ is often omitted in favor of using string juxtaposition to indicate concatenation. The *power* of an alphabet is the set of all strings of a certain length from an alphabet. For example, $\Sigma^k$ is the set of all strings of length $k$ whose symbols are from $\Sigma$, with $\epsilon = \Sigma^0$ being used to indicate the case of a zero-length string, or the empty string. Two symbolic powers are used to indicate the sets of all strings over an alphabet $\Sigma^+ = \bigcup_{k=1}^{\infty} \Sigma^k$ and $\Sigma^* = \Sigma^+ \cup \epsilon$, the latter extending the former by including the empty string. A *language* $L$ is then defined as a set of strings selected from some $\Sigma^*$, and a *problem* is defined as the question of deciding whether a given string is a member of some particular language. That is, given a string $w \in \Sigma^*$ and $L$ a language over $\Sigma$, perform classification to decide if $w \in L$.

As $L$ may be infinite, it is useful to have a compact description of the strings in $L$. Such an abstract model is called a *grammar* $G$, and it is said that $L = L(G)$. Typically, a grammar is specified by the 4-tuple $G = (V, T, P, S)$, where $V$ is the set of *variables* and $T$ is the set of *terminals* which are symbols that form the strings of $L$. $P$ is the set of *productions*, each of which represent the recursive definition of $L$, and $S \in V$ is the *start symbol*, which is the variable that defines $L$. Each production consists of a *head* variable followed by the production operator $\rightarrow$ and a *body* string of zero or more terminals and variables. Each production represents one way to form strings in $L$ from the head variable. Note that more than one body may be defined for each head variable, resulting in a nondeterministic model. One way of using a grammar is "top-down" in which each head variable is replaced with a body, the result of which is scanned for other variables, which are then replaced by one of their bodies, and so on. During this process, terminals are usually left unchanged, although there is a class of unrestricted grammars, in which terminals may be altered.

Defining a string in $L$ via head-to-body recursive expansion of $S$ is referred to as a *derivation* of the string. The relation symbols $\Rightarrow$ and $\overset{*}{\Rightarrow}$ are used to indicate one derivation step from variable to string, and zero or more steps, respectively. Aside from a derivational sentence, some classes of grammars have graphical representations to aid in understanding their linguistic structure and/or derivations. Grammars may be analyzed via their directed *derivation graphs* [53] in which each node is uniquely labeled by elements of $(V \cup T)$ such that all elements are used exactly once. Similarly, derivations may be specified via a different directed graph, called a *parse tree*, in which each interior node is labeled by some $V$, and each leaf labeled by some $(V \cup T \cup \epsilon)$. If an interior node is $A$, and its children are $X_1, ..., X_k$, then $A \rightarrow X_1 \cdots X_k$ is a production in $P$. Children of a node are ordered from left-to-right.

Given $G = (V, T, P, S)$, the language $L$ is defined by

$$L(G) = \{w \mid w \in T^*, \text{and } S \overset{*}{\Rightarrow} w\}.$$

That is, $L(G)$ is the set of all strings derived from $S$.

## 2.3.2 Chomsky Hierarchy

Among many other linguistic innovations, Noam Chomsky defined four categories for types of grammars in [15] and elaborated upon in [16]. The language levels, summarized in Figure 2.12, are contained in terms of complexity as $3 \subset 2 \subset 1 \subset 0$, where type-3 or regular grammars generate regular languages, type-2 or context-free grammars (CFGs) generate context-free languages (CFLs), type-1 or context-sensitive grammars (CSGs) generate context-sensitive languages (CSLs), and type-0 or unrestricted grammars generate recursively enumerable languages. All other languages can be classified between type-3 and type-0. Knowing the type-containment of a certain grammar is important in understanding the computational complexity necessary in solving linguistic problems. Problems in regular grammars, whose productions rewrite a variable as a terminal followed by at most one variable (e.g., $A \rightarrow aB$), may be solved in linear time $\mathcal{O}(N)$. A CFG is defined as any grammar whose productions allow any arrangement of terminals and variables in the body (e.g., $A \rightarrow aBbC$). Especially within the context of studying the secondary structure of sequences, it is

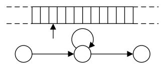
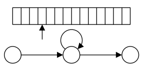
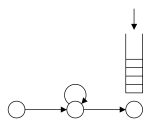
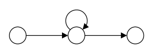
| | | | | |
|---|---|---|---|---|
| Language: | Recursively enumerable | Context sensitive | Context free | Regular |
| Grammar: | Unrestricted | Context sensitive | Context free | Regular |
| Type: | 0 | 1 | 2 | 3 |
| Example Grammar Rule: | $Baa \rightarrow A$ | $At \rightarrow aA$ | $S \rightarrow gSc$ | $A \rightarrow cA$ |
| Automaton: | Turing Machine  | Linear Bounded  | Pushdown (stack)  | Finite-State Automaton  |

Figure 2.12: The Chomsky hierarchy and formal language theory (adapted from [91]). From upper-left to lower-right is the most- to least-complicated grammar.

worth mentioning that strings in $L$(CFG) can exhibit self-embedding [16], resulting in non-crossing dependencies; i.e., palindromes necessary for properly modeling hairpin structures. Problems of a CFG may be solved in polynomial time $\mathcal{O}(N^k)$, which is generally accepted as representing the limiting bound for a practical algorithm. The next grammar type, CSG, is any grammar whose productions have additional symbols in their head, but never more than in the body (e.g., $Aa \rightarrow bB$). Again, regarding sequential secondary structure, it is worth noting that strings in $L$(CSG) can exhibit crossing dependencies–necessary for modeling pseudoknots in RNA. Unfortunately, CSG problems are classified as computationally decidable [46], guaranteed to complete in finite time but the end time is unknown, so algorithms working with CSGs may be difficult to use realistically. Finally, an unrestricted grammar is any grammar whose productions are unrestricted such that the head may have more elements than the body (e.g., $Aab \rightarrow B$). Problems are undecidable, i.e. not guaranteed to end, and so there is little hope of working in this case with unrestricted grammars, in spite of their complex modeling power.

It has been suggested [91] that the natural grammar of DNA/RNA sequences is at least context-sensitive in order to account for some of the folding that takes place for example in pseudoknot structures. Unfortunately, working with context-sensitive or unrestricted grammars is impractical due to the computational time necessary for problem solving–exponential or worse. Hence, the work presented herein operates on type-2 or type-3 grammars. While this limitation precludes representation of pseudoknot structures, it allows the representation of most other features present in biological data.

### 2.3.3 Notation

The work presented in Chapters 5 and 6 generally adopts standard linguistic notations. Beginning with [53] and given a grammar $G$ for language $L(G)$, then $V(G)$, $T(G)$ and $P(G)$ shall be the convention used for the sets of variables, terminal symbols and production rules of grammar $G$, respectively. Following [46], which uses conventions in [16], the common conventions used to indicate symbol functionality are

- early lower-case letters, $a, b, ... \in T(G)$, represent individual terminals;
- early capital letters, $A, B, ... \in V(G)$, represent individual variables;
- late lower-case letters, $..., y, z \in T(G)^+$, represent strings of terminals;
- late capital letters, $..., Y, Z \in (V(G) \cup T(G))$, represent either individual terminals or individual variables;
- lower-case Greek letters, $\alpha, \beta, ... \in (V(G) \cup T(G))^+$, represent strings consisting of variables and/or terminals.

### 2.3.4 Lempel-Ziv Compression as a Grammar

The material presented in Chapters 3 and 4 detail applications of grammars to solve several bioinformatics problems. As will be presented, the grammar of biological data is typically not assumed to be known *a priori*, and so needs to be inferred from the corpus of data. It was observed in [72], that a grammar $G$ used to model a string can be converted to an LZ77 representation in a simple way. The term LZ77 refers to Lempel-Ziv dictionary-based lossless compression detailed in [55] and [105]. Subsequently, an algorithm was presented in [12] to use an inverted process to map an LZ77-compressed sequence into a grammar. While the inverted process is more involved, it demonstrates the fact that Lempel-Ziv compression can be thought of as inferring a Regular grammar from the sequence it compresses.

### 2.3.5 Grammar Applications in Bioinformatics

The Lempel-Ziv algorithms, though usually not thought of that way, are examples of the use of a grammar for compression. The original concept behind abstract grammars is that a grammar $G$ is meant to completely describe the underlying structure of a corpus of sequences. Because most naturally occurring sequences contain repetition and redundancy, grammars are often able to describe sequences efficiently. Hence, the usage of grammars can be thought of as a means to provide compression.

A block diagram describing how grammars can be used for compression is shown in Figure 2.13. The input to the encoder is a sequence $w$. The encoder first infers a grammar $G$ specific to $w$. It should be noted that an orthodox linguist may not approve of the term “grammar” in the sense provided here, as $G$ will derive the single string $w$ and nothing else. However, time and engineering often find ways of modifying and applying existing ideas to new applications. Once the grammar is estimated, it is encoded, first into symbols then into bits, followed by storage or transmission. Upon

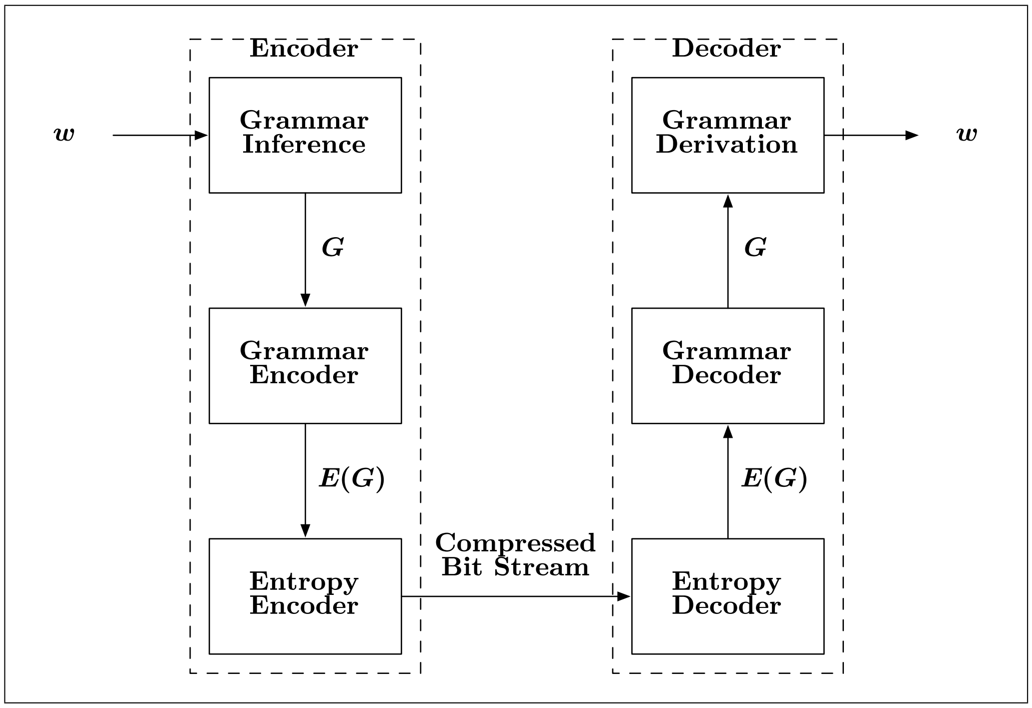


Figure 2.13: A block diagram depicting the basic steps involved with a grammar-based compression scheme.

reception, the bits are decoded into symbols and then into the inferred grammar $G$. Given this kind of grammar, it is a simple matter to recover $w$ from $G$ by beginning with the start symbol $S$, which is part of $G$. In a seminal paper Kieffer and Yang [53] showed that a grammar based source code is a universal code with respect to finite state sources over a finite alphabet.

Identification of function and/or meaning of segments of biological sequences remains an ongoing and active area of research. This implies studying primary and secondary structure of sequences. A somewhat uncommon method for predicting RNA secondary structure focuses only on the information contained within the sequences. For example, [14] reviews many ways in which linguistics, specifically abstract grammars, may be used to model and analyze secondary structures found in

RNA and protein sequences. Another example [88] includes RNA secondary structure prediction using stochastic context-free grammars (SCFGs).

Abstract grammars have been shown to be useful models of biological sequences at various levels of detail. Surveys presented in [91] and [35] describe correlations between linguistic structures and biological function. In particular, linguistic models of macromolecules [10, 41], have been used to model nucleic acid structure [90, 89, 47], protein linguistics [1, 82], and gene regulation [18, 84, 56]. Much of the work available in the literature assumes the underlying grammar is known *a priori*. Hence, there is a need for general methods to infer grammars efficiently from biological structures.

In [71] and [73] a general algorithm is presented for inferring sequential structure in the form of CFGs for generic inputs including biological data. Two other algorithms in which sets of arbitrary sequential data are categorized to generate a CFG are presented in [87] and [68]. One drawback with these algorithms, is the inability to make use of domain knowledge, although [71] discusses the improvement available when domain knowledge is applied. In fact, the algorithm was modified in [13] to operate specifically on DNA and makes use of the Chargaff base pairing rules to generate a more compact model.

The most commonly known and recognized application of grammars to computational biology are in the form of SCFGs used to search for the most likely secondary structures in RNA leading to the identification of mechanistic elements that control various aspects of regulation [88, 69, 26, 25, 23]. The remaining primary usage of grammars are in a data-mining paradigm, where grammars are used to efficiently scan databases full of experimental data from the literature (e.g., RegulonDB). Some work has briefly been done in regards to modeling GRNs using a subclass of CSGs [18, 91, 89, 47, 5] called definite clause grammars (DCGs) developed in the efficient computer language, Prolog. This was further developed into Basic Gene Grammars in

[56]. The end result is a very high-level model description with a database approach to determining the classification of sequences of data *in silico*.

In this dissertation we utilize abstract grammars to model the primary and secondary structures present in biological data. These grammar models are inferred and applied to efficiently solve various sequence analysis problems present in computational biology, including multiple sequence alignment, fragment assembly, database redundancy removal, and structural prediction. In doing so, we demonstrate the viability and versatility of using abstract grammars to model biological data. The next two chapters introduce applications of a grammar based sequence distance metric which is useful in comparing the primary structure of biological sequences. The similarity of two sequences can be estimated by comparing their inferred grammars. This concept is applied to solve three common problems involving sequence analysis. Chapters 5 and 6 introduce two novel grammar inference methods capable of capturing not only the primary structure, but the higher-level secondary structure of biological sequences. The resulting context-free grammars are used to estimate structural pieces within sequences, which can in-turn be used as supplemental information to help guide various sequence analysis algorithms. A preliminary example is provided with an MSA application that uses the inferred structural information to improve upon its alignment quality.

# Chapter 3

# Biological Sequence Alignment

The first application of grammar-based models is on the problem of sequence alignment. This often-studied topic involves identifying common subsequences among biological sequences. When matches are found, the associated pieces are shifted so that when sequences are presented as successive rows–one sequence per row–each nucleotide or amino acid residue lines-up with all others in its column. Two specific example applications were developed to solve problems involving sequence alignment issues. The first, GramAlign, is a progressive alignment algorithm that uses a grammar-based distance metric to determine the order in which biological sequences are to be pairwise aligned. The second, GramContig, is a fragment-to-reference alignment algorithm that uses the same grammar-based distance metric to identify sequence fragment, or contig, locations relative to a reference sequence. The result allows for a fragment assembly and identifies possible regions that require additional sequencing.

## 3.1 Multiple Sequence Alignment Background

Generation of meaningful multiple sequence alignments (MSAs) of biological sequences is a well-studied NP-complete problem, which has significant implications for a wide spectrum of applications [17, 23]. In general, the challenge is aligning $N$

sequences of varying lengths by inserting gaps in the sequences so that in the end all sequences have the same length. Of particular interest to computational biology are DNA/RNA sequences and amino acid sequences, which are comprised of nucleotide and amino acid residues, respectively.

MSAs are generally used in studying phylogeny of organisms, structure prediction, and identifying segments of interest among many other applications in computational biology [30]. Regarding phylogeny, $N$ sequences containing the same functionality for different organisms are aligned. Assuming the organisms evolved from the same ancestor, alignments can show how the original functionality changed for each organism. The resulting MSA may imply how closely the organisms are related to each other. In identifying segments of interest, the functionality of at least one out of $N$ sequences is unknown. Here, the assumption is that the relationship between the organisms is well understood. Consequently, the resulting MSA may imply the underlying functionality of unknown segments based upon the location relative to known segments of other organisms.

Given a scoring scheme to evaluate the fitness of an MSA, calculating the best MSA is an NP-complete problem [17]. Differences in scoring schemes, need for expert-hand analysis in most applications, and many-to-one mapping governing elements-to-functionality (codon mapping and function) make MSA a more challenging problem when considered from a biological context as well [67].

Generally, three approaches are used to automate the generation of MSAs. The first offers a brute-force method of multidimensional dynamic programming [62], which may find a good alignment but is generally computationally expensive and, therefore, unusable beyond a small $N$. Another method uses a probabilistic approach where Hidden Markov Models (HMMs) are approximated from unaligned sequences

[74]. The final method, progressive alignment, is possibly the most commonly used approach when obtaining MSAs [75].

A progressive alignment algorithm begins with an optimal alignment of two of the $N$ sequences. Then, each of the remaining $N-2$ sequences are aligned to the current MSA, either via a consensus sequence or one of the sequences already in the MSA. Variations on the progressive alignment method include PRALINE [92], ProbCons [22], MAFFT [51, 50], MUSCLE [28, 27], T-Coffee [76], Kalign [54], PSalign [96], and the most commonly used ClustalW [97]. In most cases, the algorithms attempt to generate accurate alignments while minimizing computational time or space. Advances in DNA sequencing technology with next generation sequencers such as ABI's SOLID and Roche's GC FLX provide vast amounts of data in need of multiple alignment. In the case of large sequencing projects, a high number of fragments that lead to longer contigs to be combined are generated with much less time and money [95]. In addition, as more organisms' genomes are sequenced, approaches that require MSA of the same gene in different organisms now find a more populated data set. In both cases computational time in MSA is becoming an important issue that needs to be addressed.

The next sections present GramAlign, a progressive alignment method with improvements in computational time. In particular, the natural grammar inherent in biological sequences is estimated to determine the order in which sequences are progressively merged into the ongoing MSA. The following sections describe the algorithm and present initial results as compared with other alignment algorithms.

## 3.2 Multiple Sequence Alignment Algorithm

A general overview of the GramAlign algorithm is depicted in Figure 3.1. The set of sequences to be aligned, $S$, are regarded as input to the algorithm with $S = \{s_1, ..., s_N\}$, where $s_i$ is the $i$th sequence and $i \in \{1, ..., N\}$.

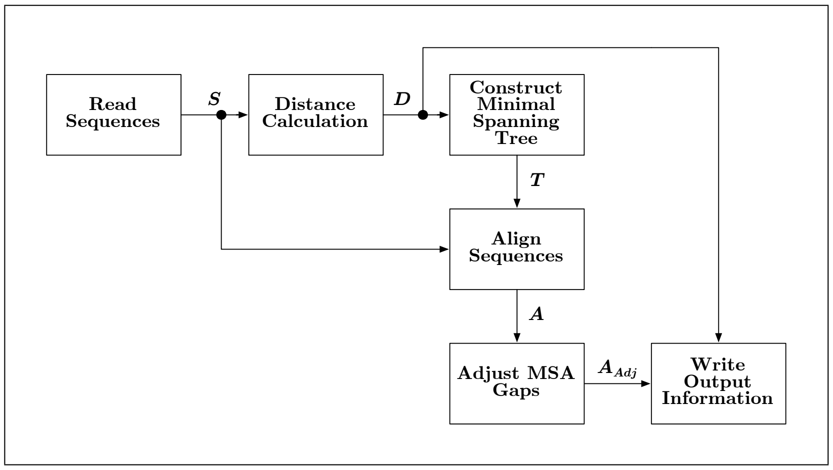


Figure 3.1: The algorithm operates on a set of sequences $S$ originally read in FASTA format. After a grammar-based distance matrix $D$ is estimated, a minimal spanning tree $T$ is constructed. The tree is used as a map for determining the order in which the sequence set is progressively aligned in $A$. Gaps in the alignment are grouped together using a sliding window resulting in $A_{Adj}$. Several outputs are available, including the distance matrix and various sequence alignment formats.

### 3.2.1 Distance Estimation

The first step in the procedure involves the formation of an estimate of the distance between each sequence $s_m$ and all other sequences $s_n \; \forall \; n \neq m$. The distance used in GramAlign is based on the natural grammar inherent to all information-containing sequences. Unfortunately, the complete grammar for biological sequences is unknown, and so cannot be used when comparing sequences. However, we do know that biological sequences have structures which correspond to functions. This in turn implies that biological sequences which correspond to proteins with similar functions will have similarities in their structure. Therefore, we use a grammar based on Lempel-Ziv (LZ) compression [105, 106] used in [79] for phylogeny reconstruction. This measure uses the fact that sequences with similar biological properties share commonalities in their sequence structure. It is also known that biological sequences contain repeats, especially in the regulatory regions [39]. When comparing sequences with functional similarity, non-uniform distribution of repeats among the sequences poses a problem for assessing sequence similarity. As shown below, the proposed distance naturally handles such cases, which are difficult to account for by alignment or sequence edit based measures.

An overview of the grammar-based distance calculation is shown in Figure 3.2 where a dictionary of grammar rules for each sequence is calculated. Initially, the dictionary $G_m^1 = \emptyset$ is empty, a fragment $f^1 = s_m(1)$ is set to the first residue of the corresponding sequence, and only the first element $s_m(1)$ is visible to the algorithm. At the $k$th iteration of the procedure, the $k$th residue is appended to the $k-1$ fragment and the visible sequence is checked. If $f^k \notin s_m(1, ..., k-1)$ then $f^k$ is considered a new rule, and so added to the dictionary $G_m^k = G_m^{k-1} \cup \left\{f^k\right\}$, and the fragment is reset for the start of the next iteration, $f^k = \emptyset$. However, if $f^k \in s_m(1, ..., k-1)$, then the current dictionary contains enough rules to produce the current fragment, i.e., $G_m^k = G_m^{k-1}$. In either case, the iteration completes by appending the $k$th residue to the visible sequence. This procedure continues until the visible sequence is equal to the entire sequence, at which time the size of the dictionary is recorded along the diagonal of the grammar elements matrix, $E_{m,m} = \left|G_m\right|$. As will be shown, calculating the distance between sequences requires only the number of entries in the dictionary.

In the next step shown in Figure 3.2, each sequence is compared with all other sequences. In particular, consider the process of comparing sequences $m$ and $n$. Initially, the dictionary $G_{m,n}^1 = G_m$ is set to that of sequence $m$, a fragment $f^1 = s_n(1)$ is set to the first residue of the $n$th sequence, and the visible sequence is all of $s_m$. The algorithm operates as described previously, resulting in a new dictionary size $E_{m,n} = \left|G_{m,n}\right|$. When complete, more grammatically-similar sequences will have a new dictionary size with fewer additional entries as compared to sequences that are

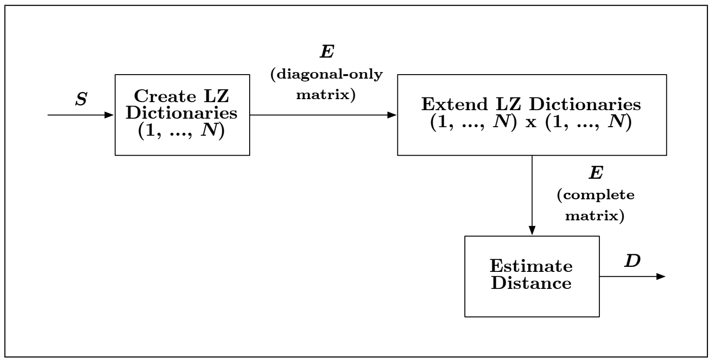

S
Create LZ Dictionaries (1, ..., N)
E
(diagonal-only matrix)
Extend LZ Dictionaries (1, ..., N) x (1, ..., N)
E
(complete matrix)
Estimate Distance
D

Figure 3.2: An $N \times N$ grammar-based distance matrix $D$ is estimated from the set of $N$ input sequences $S$. The first step in generating $D$ is to approximate the original number of elements in each sequence's dictionary based on an LZ complexity. Each dictionary is extended using all other sequences resulting in new numbers of elements. The grammar-based distance between sequences $m$ and $n$ is determined by considering the amount by which dictionaries change.

less grammatically-similar. Therefore, the size of the new dictionary $E_{m,n}$ will be close to the size of the original dictionary $E_{m,m}$.

In the final step, the distance between the sequences is estimated using the dictionary sizes. Five different distance measures were suggested in [79]. This work used the distance measure

$$d_{m,n} = \frac{E_{m,n} - E_{m,m} + E_{n,m} - E_{n,n}}{\frac{E_{m,n}+E_{n,m}}{2}}, \tag{3.1}$$

where $m, n \in \{1, ..., N\}$ are indices of two sequences being compared. This particular metric accounts for differences in sequence lengths, and normalizes accordingly. Thus, the final distance matrix $D$ is composed of grammar-based distance entries given by (3.1). Smaller entries in $D$ indicate a stronger similarity, at least in terms of the LZ-based grammar estimate. Intuitively, sequences with a similar grammar should be pairwise aligned with each other in order for progressive combining into an MSA.

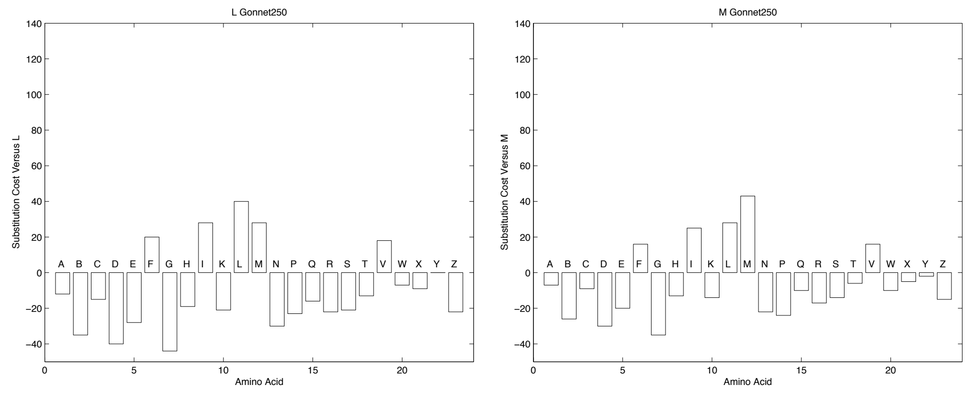
To further improve the execution time, $D$ is only partially calculated as follows. An initial sequence is selected and compared with all other sequences. The resulting distances are used to split the sequences evenly into two groups, one containing the smallest distances, and the other containing the largest distances. The process is repeated recursively on each group until the number of sequences in a group is two. The benefit is that only $N\log(N)$ distances need to be calculated. The validity of only calculating these sets of distances stems from the transitivity of the LZ grammars being inferred. That is, if the grammar-based distances $d_{i,j}$ and $d_{j,k}$ are small, it is likely that $d_{i,k}$ is also small. By recursively dividing groups of extreme distances, only those distances which would likely be used in the spanning-tree creation process will actually be calculated.

**Sequence Alphabet**

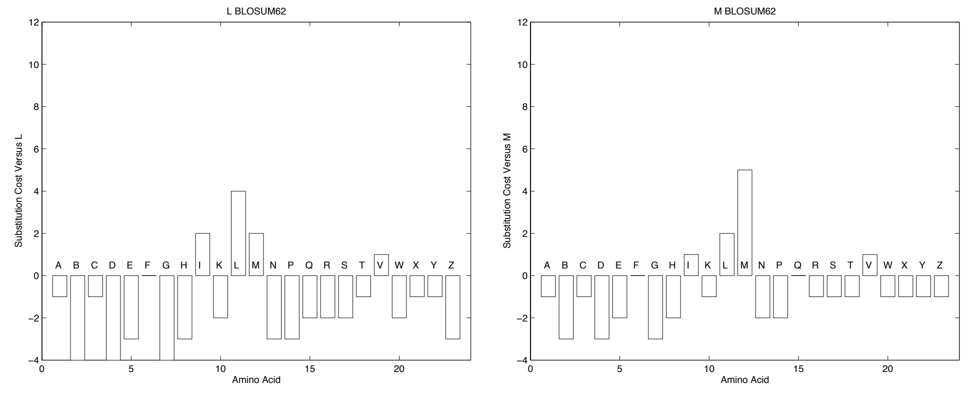
The distance between sequences $m$ and $n$ as determined by (3.1) is based on how many additional rules need to be added to each grammar in order to generate both $s_m$ and $s_n$. Because the real grammars are unknown, $G_m$ and $G_n$ are approximated by scanning the only observations available (i.e., $s_m$ and $s_n$). The grammar approximation improves as the length of the observed sequences increases. And so, the distance calculations are a function of sequence lengths, becoming more accurate as the sequences increase in length. In practice, this calculation works well for DNA sequences, even of shorter lengths, because the approximated grammar of a DNA sequence can only contain rules involving words composed of combinations of elements from the alphabet {'A','C','G','T'}. This small alphabet allows for a rapid generation of a reasonable grammar since there are a relatively small number of permutations of letters.

From a grammar perspective, amino acid sequences are generally much more difficult to process correctly using (3.1). The reason being the alphabet contains 23 letters, where each element is not equally different from all other elements. Due to the relatively large alphabet size, much longer sequences are necessary to generate a reasonable grammar approximation. Thus, the accuracy of distances calculated for sets of short amino acid sequences are diminished. Additionally, consider the substitution scores of ‘L’ and ‘M’ as taken from the GONNET250 and BLOSUM62 substitution matrices in Figure 3.3. Notice in (a) and (c), that ‘L’ receives a relatively high positive value when aligned with any of {‘I’,‘L’,‘M’,‘V’}. Similarly, in (b) and (d), ‘M’ receives a relatively high positive value when aligned with any of the same set. Additionally, both ‘L’ and ‘M’ generally receive high negative values when compared to letters other than {‘I’,‘L’,‘M’,‘V’}. When taking this type of scoring into account, the elements ‘L’ and ‘M’ could be considered the same letter in a grammatical sense. Note that these positive and negative scores were originally derived for an evolutionary model parameterized in terms of residue pair probabilities. Two sequences known to be homologous were used to estimate the likelihood of amino acid residue alignments. The pairwise scores were calculated using a log odds ratio $\log p_{AB}/q_A q_B$, where $q_i$ is the relative frequency of residue $i$ and $p_{jk}$ is the probability that residues $j$ and $k$ are aligned [17].

Thus, GramAlign offers the option to use a “Merged Amino Acid Alphabet” when calculating the distance matrix. The merged alphabet contains 11 elements corresponding to the 23 amino acid letters grouped into the sets {‘A’,‘S’,‘T’,‘X’}, {‘B’,‘D’,‘N’}, {‘C’}, {‘E’,‘K’,‘Q’,‘R’,‘Z’}, {‘F’}, {‘G’}, {‘H’}, {‘I’,‘L’,‘M’,‘V’}, {‘P’}, {‘W’}, and {‘Y’}. These groupings were determined by considering all 23 rows of the BLOSUM45, BLOSUM62, BLOSUM80 and GONNET250 substitution matrices, and only grouping elements that had a strong similarity across the entire row in all four matrices. The merged alphabet has the benefit of containing fewer elements allowing for more accurate distance estimates based upon shorter observed sequences. Also, the



(a) GONNET250 Row 'L' (b) GONNET250 Row 'M'



(c) BLOSUM62 Row 'L' (d) BLOSUM62 Row 'M'

Figure 3.3: Bar graphs of the substitution scores for amino acid 'L' and 'M' as taken from the Gonnet250 and BLOSUM62 substitution matrices. The scores are shown based on an alphabetical ordering of amino acid letters from the leftmost 'A' to rightmost 'Z'.

resultant merged-alphabet substitution matrices are more consistent in that a merged-letter score is high only when compared to itself. In practice, the average alignment scores increased when aligning the same data sets using the merged alphabet within the distance calculation, as compared to using the actual alphabet (results not shown). In either case, once the distances have been calculated, a tree based on these distances is used to determine which sequences should be pairwise aligned.

### 3.2.2 Tree Construction

The next step in the algorithm consists of constructing a minimal spanning tree $T$ based on the distance matrix $D$. In particular, consider a completely connected graph of $N$ vertices and $\frac{N(N-1)}{2}$ edges, where the weight of an edge between vertices $i$ and $j$ is given by the $(i,j)$th element of the distance matrix, $D_{i,j}$. This work uses Prim's Algorithm [2] to determine a minimal spanning tree $T$ which may be used as a guide in determining the order for progressively aligning the set of sequences $S$.

### 3.2.3 Align Sequences

The minimal spanning tree $T$ along with the set of sequences $S$, are processed by the "Align Sequences" block in Figure 3.1. This block is presented in more detail in Figure 3.4. The first two sequences from $S$ to be aligned are given by $T$ as the root sequence of $T$ and the nearest sequence in terms of the LZ grammar distance. At the conclusion of the pairwise alignment process, the resulting alignment is stored in an ensemble of sequences.

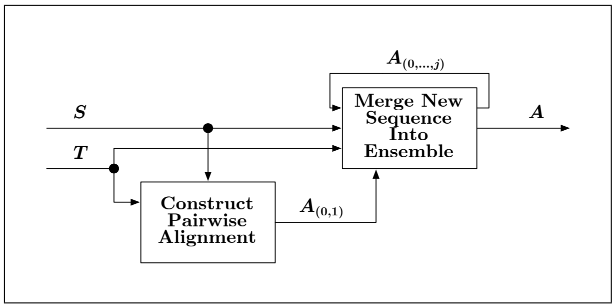

$A_{(0,\ldots,j)}$
$S$
$T$
Merge New Sequence Into Ensemble
$A$
Construct Pairwise Alignment
$A_{(0,1)}$

Figure 3.4: From the spanning tree $T$ and the set of sequences $S$, a progressive alignment is generated and stored in an ensemble. When no more sequences remain, the final alignment $A$ is available for post-processing gap adjustments.

In the following we describe the pairwise alignment procedure, the scoring system and the method for progressive alignment.

## Dynamic Programming

At the core of most progressive MSA algorithms is some method for performing pairwise alignments between two sequences. The implementation in GramAlign uses a version of the Needleman-Wunsch [70] dynamic programming algorithm with affine gap scores as discussed in [23] to generate each pairwise alignment; it requires the six matrices partially shown in Figure 3.5. A cell in $M$ indicates the maximum score achievable given that for the $i$th column and $j$th row, elements $x_i$ and $y_j$ are aligned with each other. A cell in $I_x$ indicates the maximum score achievable given that for the $i$th column and $j$th row, element $x_i$ is aligned with a gap. A cell in $I_y$ indicates the maximum score achievable given that for the $i$th column and $j$th row, element $y_j$ is aligned with a gap. The trace-back matrices, $TB_M$, $TB_{I_x}$, and $TB_{I_y}$, are used in the backward pass to determine how the residues are aligned. Specifically, the $(i, j)$ cell in $TB_M$ will point to cell $(i-1, j-1)$ in either matrix $M$, $I_x$ or $I_y$. The significance of which matrix is pointed to determines if the $i-1$ and $j-1$ residues are aligned with each other, or if one is aligned with a gap. On the other hand, the $(i, j)$ cell in $TB_{I_x}$ will point to cell $(i-1, j)$ in either matrix $M$ or $I_x$, the difference being a gap is opened or a gap is extended, respectively. Similarly, the $(i, j)$ cell in $TB_{I_y}$ will point to cell $(i, j-1)$ in either matrix $M$ or $I_y$. The trace-back procedure is followed until the final cell $f$ is reached in any of the $TB$ matrices. Note, the matrix notation used here follows that of [23] where coordinates in the matrices are given as $(i, j)$, in which the column is the first coordinate. However, we deviate from [23] with the addition of the left-most "gap column" and top-most "gap row" as depicted in Figure 3.5.

To demonstrate the pairwise alignment procedure, consider a toy example of aligning $s_0 = ACGGT$ and $s_1 = AGGT$. As with all dynamic programming algorithms, the first half of the procedure involves calculating all path distances from the finish

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | 0 | 0 | 0 |
| $y_1$ | 0 | $c(x_1, y_1)$ | |
| $y_2$ | 0 | | |

$M$

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | $f$ | $\cdot$ | $\cdot$ |
| $y_1$ | $\cdot$ | $M$ | |
| $y_2$ | $\cdot$ | | |

$TB_M$

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | 0 | $-d'$ | $-d' - e'$ |
| $y_1$ | 0 | 0 | |
| $y_2$ | 0 | 0 | |

$I_x$

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | $f$ | $I_x$ | $I_x$ |
| $y_1$ | $\cdot$ | $\cdot$ | |
| $y_2$ | $\cdot$ | $\cdot$ | |

$TB_{I_x}$

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | 0 | 0 | 0 |
| $y_1$ | $-d'$ | 0 | 0 |
| $y_2$ | $-d' - e'$ | | |

$I_y$

| | $-$ | $x_1$ | $x_2$ |
|---|---|---|---|
| $-$ | $f$ | $\cdot$ | $\cdot$ |
| $y_1$ | $I_y$ | $\cdot$ | $\cdot$ |
| $y_2$ | $I_y$ | | |

$TB_{I_y}$

Figure 3.5: The Needleman-Wunsch dynamic programming implementation in GramAlign uses three trace-back matrices $TB_M$, $TB_{I_x}$ and $TB_{I_y}$, and three scoring matrices $M$, $I_x$ and $I_y$.

to the start via a forward pass. The first step initializes the score matrices as follows:

$$\begin{aligned}
M(0,j) &= 0 && \text{for} \quad 0 \le j \le |s_1| \\
M(i,0) &= 0 && \text{for} \quad 0 \le i \le |s_0| \\
M(1,1) &= c(s_0(1), s_1(1)) && \\
I_x(0,j) &= 0 && \text{for} \quad 0 \le j \le |s_1| \\
I_x(1,j) &= 0 && \text{for} \quad 1 \le j \le |s_1| \\
I_x(i,0) &= -d' - \big((i-1) \times e'\big) && \text{for} \quad 1 \le i \le |s_0| \\
I_y(i,0) &= 0 && \text{for} \quad 0 \le i \le |s_0| \\
I_y(i,1) &= 0 && \text{for} \quad 1 \le i \le |s_0| \\
I_y(0,j) &= -d' - \big((j-1) \times e'\big) && \text{for} \quad 1 \le j \le |s_1|,
\end{aligned}$$

where $d'$ is the tail gap open penalty, $e'$ is the tail gap extension penalty, and $c$ is the comparison cost function. For the example, $s_0$ is sequence $X$ along the top and $s_1$ is sequence $Y$ along the left.

The initialization concludes by setting the trace-back matrices as follows:

$$\begin{array}{rll} TB_M(0,0) = & f & \\ TB_M(1,1) = & M & \\ TB_{I_x}(0,0) = & f & \\ TB_{I_x}(i,0) = & I_x & \text{for} \quad 1 \leq i \leq |s_0| \\ TB_{I_y}(0,0) = & f & \\ TB_{I_y}(0,j) = & I_y & \text{for} \quad 1 \leq j \leq |s_1|, \end{array}$$

where $f$ is the indication for the final cell. After initialization has completed, the current set of score matrices for the example are as depicted in Figure 3.6. Note

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $A$ | 0.00 | 1.00 | | | | |
| $G$ | 0.00 | | | | | |
| $G$ | 0.00 | | | | | |
| $T$ | 0.00 | | | | | |

$M$

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | $-8.70$ | $-9.10$ | $-9.50$ | $-9.90$ | $-10.30$ |
| $A$ | 0.00 | 0.00 | | | | |
| $G$ | 0.00 | 0.00 | | | | |
| $G$ | 0.00 | 0.00 | | | | |
| $T$ | 0.00 | 0.00 | | | | |

$I_x$

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $A$ | $-8.70$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $G$ | $-9.10$ | | | | | |
| $G$ | $-9.50$ | | | | | |
| $T$ | $-9.90$ | | | | | |

$I_y$

Figure 3.6: The Needleman-Wunsch scoring matrices for the toy example after initialization.

the cost function is defined as $c(x_i, y_j) = 1$ when $x_i = y_j$ and $c(x_i, y_j) = -0.9$ when $x_i \neq y_j$, and the gap open and extension penalties are $d = 8.7$, $d' = 8.7$, $e = 0.8$, and $e' = 0.4$. The difference between $d'$ and $d$, and $e'$ and $e$ is whether the gap is initiated at either end of the alignment–the tails–or in the middle of the alignment, respectively. The current set of trace-back matrices for the example are as depicted in Figure 3.7.

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | $f$ | · | · | · | · | · |
| $A$ | · | $M$ |  |  |  |  |
| $G$ | · |  |  |  |  |  |
| $G$ | · |  |  |  |  |  |
| $T$ | · |  |  |  |  |  |

$TB_M$

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | $f$ | $I_x$ | $I_x$ | $I_x$ | $I_x$ | $I_x$ |
| $A$ | · | · |  |  |  |  |
| $G$ | · | · |  |  |  |  |
| $G$ | · | · |  |  |  |  |
| $T$ | · | · |  |  |  |  |

$TB_{I_x}$

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | $f$ | · | · | · | · | · |
| $A$ | $I_y$ | · | · | · | · | · |
| $G$ | $I_y$ |  |  |  |  |  |
| $G$ | $I_y$ |  |  |  |  |  |
| $T$ | $I_y$ |  |  |  |  |  |

$TB_{I_y}$

Figure 3.7: The Needleman-Wunsch trace-back matrices for the toy example after initialization.

The forward pass fills out the matrices based on the following rules:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + c(s_0(i), s_1(j)) \implies TB_M(i,j) = M \\ I_x(i-1,j-1) + c(s_0(i), s_1(j)) \implies TB_M(i,j) = I_x \\ I_y(i-1,j-1) + c(s_0(i), s_1(j)) \implies TB_M(i,j) = I_y, \end{cases}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - d \implies TB_{I_x}(i,j) = M \\ I_x(i-1,j) - e \implies TB_{I_x}(i,j) = I_x, \end{cases}$$

and

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - d \implies TB_{I_y}(i,j) = M \\ I_y(i,j-1) - e \implies TB_{I_y}(i,j) = I_y. \end{cases}$$

Here $d$ and $e$ are the gap open and extension penalties, respectively. If it should happen that multiple scores are the max, a random selection is made in determining the entry into the appropriate trace-back matrix. To account for the end tail scoring, when $i = |s_0|$ during the calculation for $I_y$, $d$ and $e$ are replaced with $d'$ and $e'$ respectively. Similarly for $I_x$ when $j = |s_1|$. After the dynamic programming forward pass completes, the score and trace-back matrices are as depicted in Figures 3.8 and 3.9.

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $A$ | 0.00 | 1.00 | −9.60 | −10.00 | −10.40 | −10.80 |
| $G$ | 0.00 | −9.60 | 0.10 | −6.70 | −7.50 | −10.20 |
| $G$ | 0.00 | −10.00 | −8.60 | 1.10 | −5.70 | −8.40 |
| $T$ | 0.00 | −10.40 | −9.40 | −9.50 | 0.20 | **−4.70** |

$M$

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | −8.70 | −9.10 | −9.50 | −9.90 | −10.30 |
| $A$ | 0.00 | 0.00 | −7.70 | −8.50 | −9.30 | −10.10 |
| $G$ | 0.00 | 0.00 | −18.30 | −8.60 | −9.40 | −10.20 |
| $G$ | 0.00 | 0.00 | −18.70 | −17.30 | −7.60 | −8.40 |
| $T$ | 0.00 | 0.00 | −19.10 | −18.10 | −18.20 | −8.50 |

$I_x$

|  | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $A$ | −8.70 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| $G$ | −9.10 | −7.70 | −18.30 | −18.70 | −19.10 | −19.50 |
| $G$ | −9.50 | −8.50 | −8.60 | −15.40 | −16.20 | −18.90 |
| $T$ | −9.90 | −9.30 | −9.40 | −7.60 | −14.40 | −17.10 |

$I_y$

Figure 3.8: The Needleman-Wunsch scoring matrices for the toy example after the forward pass. The score in boldface indicates the starting point of the trace-back, matrix $M$.

The backward pass performs a trace-back, which begins by selecting the maximum score of the final cell in the three distance matrices,

$$\max\left\{M(|s_0|,|s_1|), I_x(|s_0|,|s_1|), I_y(|s_0|,|s_1|)\right\}.$$

Should more than one cell be the maximum, a random selection is made to determine the initial trace-back matrix. Given the choice of trace-back matrix, an appropriate alignment is initialized.

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | **f** | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ |
| $A$ | $\cdot$ | **M** | $I_x$ | $I_x$ | $I_x$ | $I_x$ |
| $G$ | $\cdot$ | $I_y$ | $M$ | $\mathbf{I_x}$ | $I_x$ | $I_x$ |
| $G$ | $\cdot$ | $I_y$ | $I_y$ | $M$ | **M** | $M$ |
| $T$ | $\cdot$ | $I_y$ | $I_y$ | $I_y$ | $M$ | **M** |

$TB_M$

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | $f$ | $I_x$ | $I_x$ | $I_x$ | $I_x$ | $I_x$ |
| $A$ | $\cdot$ | $\cdot$ | **M** | $I_x$ | $I_x$ | $I_x$ |
| $G$ | $\cdot$ | $\cdot$ | $M$ | $M$ | $I_x$ | $I_x$ |
| $G$ | $\cdot$ | $\cdot$ | $M$ | $M$ | $M$ | $I_x$ |
| $T$ | $\cdot$ | $\cdot$ | $M$ | $M$ | $M$ | $M$ |

$TB_{I_x}$

| | $-$ | $A$ | $C$ | $G$ | $G$ | $T$ |
|---|---|---|---|---|---|---|
| $-$ | $f$ | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ |
| $A$ | $I_y$ | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ | $\cdot$ |
| $G$ | $I_y$ | $M$ | $M$ | $M$ | $M$ | $M$ |
| $G$ | $I_y$ | $I_y$ | $M$ | $M$ | $M$ | $M$ |
| $T$ | $I_y$ | $I_y$ | $I_y$ | $M$ | $M$ | $M$ |

$TB_{I_y}$

Figure 3.9: The Needleman-Wunsch trace-back matrices for the toy example after the forward pass. The entries in boldface indicate the path from start to finish, beginning with matrix $M$.

Referring to the example, $M(5,4)$ contains the maximum score of -4.7, which implies that $s_0(5)$ is aligned with $s_1(4)$. Then, looking at $TB_M(5,4)$, the algorithm traces backward to $TB_M(4,3)$ which implies that $s_0(4)$ and $s_1(3)$ are to be aligned with each other. Again, the entry in $TB_M(4,3)$ sends the trace to $TB_M(3,2)$ which means $s_0(3)$ and $s_1(2)$ are aligned. At this point, the entry in $TB_M(3,2)$ causes the trace to jump into a new matrix, $TB_{I_x}(2,1)$; resulting in the alignment of $s_0(2)$ with a gap. The algorithm follows the entry in $TB_{I_x}(2,1)$ back to $TB_M(1,1)$ where it aligns $s_0(1)$ to $s_1(1)$. The backwards trace is complete when the final cell, $f$, has been sampled in $TB_M(0,0)$.

The final pairwise alignment has inserted a gap into $s_1$ thereby adjusting its length and increasing the homologous score by aligning the last three bases in each sequence. The final rectangular array is

$$\begin{array}{ccccc} A & C & G & G & T \\ A & - & G & G & T. \end{array}$$

## Scoring System

A significant ambiguity regarding the dynamic programming procedure is the scoring function used when comparing two elements, or when comparing an element with a gap.

Typically, the pairwise scoring function $c()$ is simply a matrix of values, where each column and row represent one element in the alphabet. In this way, each cell of the matrix corresponds to some measure representing the likelihood that two sequence elements should be aligned with each other. The most well-known amino acid scoring matrices are the Percent Accepted Mutation (PAM) [21], BLOck SUbstitution Matrix (BLOSUM) [42] and GONNET [36]. GramAlign defaults to the GONNET250 substitution matrix for the scoring function $c()$, as other progressive alignment algorithms generally use it as the default choice (e.g., [54] and [97]).

Determining the best gap-open and gap-extension penalties is a challenging problem, made more difficult by introducing two different penalties to account for the beginning and ending tail gaps of alignments. The default gap penalties used by GramAlign have been adjusted to perform well based on the alignment sets presented in the results section.

**Progressive Alignment**

The ensemble is implemented as a doubly-linked list, where each node of the list represents a single column of the alignment. Each node of the ensemble contains an array of letters corresponding to the respective column alignment, a tally of gaps in the column, a weighted combination of substitution scores, and two gap penalties. Once the initial ensemble $A_{(0,1)}$ is constructed between the first two entries in $T$, the remaining sequences need to be added to the ensemble in the order defined by $T$. This is accomplished by checking $T$ for the next sequence not already in the ensemble, call it sequence $s_j$ where $j$ corresponds to the order in which the sequence was added to $T$; that is, $j$ is the priority of the sequence. To progressively add $s_j$ to the alignment, a pairwise alignment between the ensemble $A_{(0,\ldots,j-1)}$ and $s_j$ is created via the afore mentioned dynamic programming algorithm. While the algorithm used is a pairwise alignment algorithm the distance calculated at each step of the pairwise alignment is an average of the distances between the particular position being aligned in the new sequence and the corresponding amino acids or bases in the ensemble at that node. The new pairwise alignment is merged into the ongoing ensemble based on the trace-back. The process continues until all sequences have been added to the ensemble of sequences. When sequence $s_j$ is added to the current ensemble $A_{(0,\ldots,j-1)}$, each node is updated to reflect the new column element.

### 3.2.4 Gap Adjustment

Once all $N$ sequences have been progressively aligned, the final post-processing block in Figure 3.1, "Adjust MSA Gaps", is used to cluster gaps together. The adjustment is further detailed in Figure 3.10, where the ensemble $A$ is scanned so a histogram $H$ of gaps-per-column is generated.

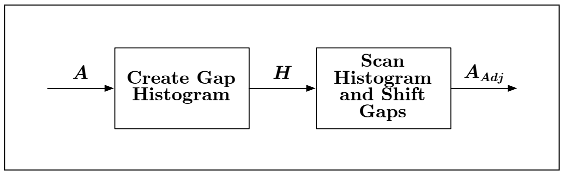


Figure 3.10: Gaps in the complete MSA ensemble $A$ are grouped together via a sliding window. After the histogram $H$ of gaps-per-column is generated, an equidistant column-window is shifted across the alignment, moving one column per interval. If the center column contains more gaps than some parameter threshold, the columns within the window are scanned for possible gaps that may be shifted into the center column. The resulting adjusted ensemble $A_{Adj}$ is presented as the final alignment.

The histogram $H$ is scanned using an equidistant, user-adjustable sliding window about each column. For each column, when the number of gaps is greater than a user-adjustable threshold percentage of gaps-per-column, the following steps are taken for each row in the column under consideration:

1. If the current row has a gap, move to the next row;
2. Otherwise, scan the current row of the neighboring columns within the window, beginning with the nearest columns and work outward;
3. If a neighboring column has a gap in the current row and the neighboring column has fewer total gaps than the center column, shift the gap from the neighboring column into the column under consideration.

As an illustration, consider a portion of the ensemble

$$A : \left\{ \begin{array}{lllll} \ldots x_{1,i-2} & x_{1,i-1} & -_{1,i} & x_{1,i+1} & x_{1,i+2} \cdots \\ \ldots x_{2,i-2} & -_{2,i-1} & -_{2,i} & -_{2,i+1} & x_{2,i+2} \cdots \\ \ldots x_{3,i-2} & -_{3,i-1} & -_{3,i} & -_{3,i+1} & x_{3,i+2} \cdots \\ \ldots -_{4,i-2} & x_{4,i-1} & x_{4,i} & -_{4,i+1} & x_{4,i+2} \cdots \\ \ldots x_{5,i-2} & x_{5,i-1} & x_{5,i} & -_{5,i+1} & x_{5,i+2} \cdots \end{array} \right.$$

where $x_{m,n}$ represents any element other than a gap in column $n$ of sequence $m$, and $-_{m,n}$ represents a gap in column $n$ of sequence $m$. And so, the gap histogram for this section of ensemble is $H = \{..., 1, 2, 3, 4, 0, ...\}$. Assuming the gap threshold is 0.4, then only columns with more than two gaps will be considered for adjustment. In the example, $H$ is scanned until column $i$ is identified as having three gaps. Following the procedure, each row in column $i$ is checked until a non-gap entry is found. In the example, the first non-gap entry $x_{4,i}$ is in row four. Assuming the gap window is 2, elements in the fourth row of the neighboring columns are checked for gap entries. In particular, column $(i+1)$ is checked first, with a gap entry $-_{4,i+1}$. However, no shift occurs because a quick check of $H$ shows that column $(i+1)$ has more gaps than column $i$. Continuing the scan, columns $(i-1)$ and $(i+2)$ are checked before another gap is found in column $(i-2)$. In this case, $H$ indicates column $(i-2)$ has fewer gaps compared to column $i$, and so a blind shift of entries between $(i-2)$ and $i$ occurs, resulting in the ensemble

$$A : \left\{ \begin{array}{ccccc} \cdots x_{1,i-2} & x_{1,i-1} & -_{1,i} & x_{1,i+1} & x_{1,i+2} \cdots \\ \cdots x_{2,i-2} & -_{2,i-1} & -_{2,i} & -_{2,i+1} & x_{2,i+2} \cdots \\ \cdots x_{3,i-2} & -_{3,i-1} & -_{3,i} & -_{3,i+1} & x_{3,i+2} \cdots \\ \cdots x_{4,i-1} & x_{4,i} & -_{4,i-2} & -_{4,i+1} & x_{4,i+2} \cdots \\ \cdots x_{5,i-2} & x_{5,i-1} & x_{5,i} & -_{5,i+1} & x_{5,i+2} \cdots \end{array} \right.$$

where original indices are kept to depict which entries are shifted into which locations.

The result is a blind movement of sparse gaps into dense regions of gaps. Numeric simulations have shown this post-processing stage does not affect alignment scoring based upon the method used in Section 3.2.6 (results not shown). Consequently, the user-defined parameters are set to a threshold of 1.0 and a window of 0 columns by default thereby disabling the gap adjustment block. Should it be known there are conserved regions of gaps, the user may decide to enable this process to encourage gap grouping.

### 3.2.5 Algorithm Complexity

The algorithm complexity of GramAlign may be broken into five pieces, beginning with the generation of each sequence grammar dictionary, $G_i$ for $i \in \{1, ..., N\}$, where $N$ is the number of sequences. Suppose the average sequence length is $L$, then each $G_i$ results in complexity $\mathcal{O}(L)$, so all dictionaries are generated with complexity $\mathcal{O}(LN)$. Next, the distance matrix $D$ is formed by recursively extending a grammar by all other sequences within it's neighborhood, each of which results in complexity $\mathcal{O}(L)$, then splitting the neighborhood into two halves, resulting in a complexity $\mathcal{O}(LN \log(N))$. The spanning tree $T$ is constructed by searching over $D$ with a complexity of $\mathcal{O}(N^2)$. The tree is used as a map in determining the order in which to perform $N - 1$ pairwise alignments, each requiring a complexity of $\mathcal{O}(L^2 + L)$. Thus, the progressive alignment process takes $\mathcal{O}(L^2 N)$. The alignment ensemble is scanned and has gaps shifted in $\mathcal{O}(LN)$ time. Thus, the entire time complexity for GramAlign is $\mathcal{O}(LN + LN \log(N) + N^2 + L^2 N + LN)$, which simplifies to approximately $\mathcal{O}(N^2 + L^2 N)$.

### 3.2.6 Numeric Simulations

In this section, example alignments are used to study the possible advantages of GramAlign. All results were generated by compiling and executing the respective MSA programs on the same computer; specifically, an Apple iBook with a PowerPC G4 operating at 1.2 GHz with 1.25 Gb system memory and a 512 Kb L2 cache. Two sets of experiments were conducted. The first set of experiments were conducted using the unaligned FASTA files from the BAliBASE 3.0 [99] data-set, a well-accepted benchmark database containing example amino acid sequences. The resulting aligned FASTA files from each algorithm were scored using `bali_score`, a program provided with the BAliBASE distribution that generates a Sum-of-Pairs (SP) score and a Total-Column (TC) score based on predetermined reference alignments [98].

Given an $M$-column alignment of $N$ sequences in which column $i$ contains the aligned elements $A_{i1}, ..., A_{iN}$, pairwise residue comparisons are made between $A_{ij}$ and $A_{ik}$. Define the value $p_{ijk} = 1$ if residues $A_{ij}$ and $A_{ik}$ are aligned with each other in the reference alignment, or $p_{ijk} = 0$ if not. For the $i$th column score is defined as:

$$S_i = \sum_{\substack{j=1 \\ j \neq k}}^{N} \sum_{k=1}^{N} p_{ijk},$$

the SP score for the alignment is:

$$SP = \frac{\sum_{i=1}^{M} S_i}{\sum_{i=1}^{M_r} S_{ri}},$$

where $M_r$ is the number of columns in the reference alignment and $S_{ri}$ is the $S_i$ score for the $i$th column in the reference alignment.

Using the same alignment, the score $C_i = 1$ if all the residues in the column are aligned in the reference alignment, or $C_i = 0$ if not. The TC score for the alignment is then:

$$TC = \frac{1}{M} \sum_{i=1}^{M} C_i.$$

The size of the sequences in the BAliBASE distribution are relatively small and, therefore, not very useful in demonstrating the advantages to be obtained using a fast algorithm. The second set of experiments were conducted using sequences generated by Rose version 1.3 to demonstrate algorithms' capabilities on large data sets containing either long or numerous sequences. Rose is a software tool that implements a probabilistic model of sequence evolution, so that a user is able to generate families of related sequences from a common ancestor sequence via insertion, deletion and substitution [94]. Rose allows for many parameter adjustments including rate of mutation, desired average final sequence length and number of desired sequences. The tool outputs the unaligned sequences, as well as the real alignment based on how mutations occur, and an evolutionary tree. The set of sequences generated by Rose were based on the default seed file provided with the Rose software distribution, where the seed file is the method used to input parameters to Rose.

Note the use of simulated data here is to demonstrate the speed advantage of GramAlign, while maintaining a similar qualitative score. The default values were used to generate the data and the algorithms were not tuned to the data. The use of simulated data may actually provide a biased advantage in quality score to any given alignment program, depending on how the simulated data is generated. A wider breadth of simulated data, such as was done in [77], would provide a better assessment of overall alignment quality.

**BAliBASE Experiments**

Alignment files in the BAliBASE database are separated into five categories (RV1x through RV50), each exhibiting different classes of alignment issues (e.g., one sequence might be significantly longer than the other sequences in a file). The first class is further divided into two subcategories labeled RV11 and RV12. The results presented in Table 3.1 and Table 3.2, respectively, detail the average SP and TC scores over each category as aligned by GramAlign version 1.14, ClustalW version 1.83, T-Coffee version 4.45, PSAlign using ProbCons as the tree generation (no version given, archive created on 3/2/2006), Kalign version 1.04, MAFFT version 5.861, and MUSCLE version 3.6. Additionally, a fast version was tested for ClustalW, MAFFT, MUSCLE and MAFFT version 6.240. In particular, the command line options used were `clustalw -quicktree`, `mafft --retree 1`, `muscle -maxiters 1 -diags -sv -distance1 kbit20_3` and `mafft --retree 1 --parttree --partsize 50` to incorporate high-speed progressive options. In all cases the default parameters were used for each program. In general, there are no significant differences in the performance of GramAlign and other algorithms as far as the SP and TC scores are concerned. As may be seen, GramAlign provides similar alignments in terms of the quality determined via the scoring method used.

Table 3.1: Average SP score for each algorithm for each category offered by the BAliBASE test suite. The bold entries indicate the lowest and highest scores.

| **Algorithm** | **RV11** | **RV12** | **RV20** | **RV30** | **RV40** | **RV50** |
|---|---|---|---|---|---|---|
| MUSCLE (fast) | 0.4904 | 0.8303 | 0.8359 | 0.7076 | 0.6904 | 0.6823 |
| MAFFT (fast) | 0.4801 | 0.8161 | 0.8404 | 0.7345 | 0.7187 | 0.7089 |
| MAFFT v6.240 | 0.4790 | **0.8066** | **0.8096** | **0.6801** | **0.6610** | 0.6985 |
| MAFFT | 0.4914 | 0.8258 | 0.8459 | 0.7437 | 0.7347 | 0.7253 |
| GramAlign | 0.5089 | 0.8328 | 0.8270 | 0.6855 | 0.7239 | 0.6903 |
| Kalign | 0.5029 | 0.8504 | 0.8410 | 0.7389 | 0.7259 | 0.7299 |
| ClustalW (fast) | **0.4748** | 0.8367 | 0.8258 | 0.6843 | 0.6705 | 0.6715 |
| MUSCLE | 0.5578 | 0.8583 | 0.8548 | 0.7492 | 0.7623 | 0.7384 |
| ClustalW | 0.4908 | 0.8197 | 0.8219 | 0.6841 | 0.6950 | **0.6698** |
| PSAlign | **0.5924** | **0.8804** | **0.8720** | 0.7554 | **0.7937** | **0.7739** |
| T-Coffee | 0.5181 | 0.8650 | 0.8660 | **0.7588** | 0.7452 | 0.7715 |

Presented in Table 3.3 are the execution times necessary to generate the entire data presented in Table 3.1 and Table 3.2. GramAlign finishes in approximately 0.4% of the time needed by PSAlign, which generated the highest scoring alignments in five out of the six BAliBASE categories as far as SP scores are concerned. PSAlign's average SP and TC score on the other hand were 9.4 and 17.5% better than GramAlign's scores, which was approximately 223 times faster. Out of the four approaches MAFFT, MAFFT v6, MAFFT (fast), MUSCLE (fast), which were 17.1, 49.9, 54.0, and 55.7% faster than GramAlign, respectively, only MAFFT had a 2% better average SP score than GramAlign. All other average SP and TC scores were equivalent or worse than that of GramAlign. Further, the GramAlign alignments scored equal-to or greater-than 56.9, 59.6, 60.8, and 71.1% of the trials based on TC score, compared to MAFFT, MAFFT v6, MAFFT (fast), and MUSCLE (fast) (results not shown). GramAlign

Table 3.2: Average TC score for each algorithm for each category offered by the BAliBASE test suite. The bold entries indicate the lowest and highest scores.

| **Algorithm** | **RV11** | **RV12** | **RV20** | **RV30** | **RV40** | **RV50** |
|---|---|---|---|---|---|---|
| MUSCLE (fast) | 0.2421 | 0.6349 | **0.2599** | **0.2457** | **0.2614** | 0.2719 |
| MAFFT (fast) | 0.2354 | **0.6209** | 0.3094 | 0.2910 | 0.3108 | 0.3087 |
| MAFFT v6.240 | 0.2461 | 0.6320 | 0.2978 | 0.2987 | 0.3104 | 0.3435 |
| MAFFT | 0.2532 | 0.6256 | 0.3168 | 0.3158 | 0.3073 | 0.3303 |
| GramAlign | 0.2993 | 0.6701 | 0.2917 | 0.2503 | 0.3292 | 0.3006 |
| Kalign | 0.2538 | 0.6749 | 0.2765 | 0.2955 | 0.3253 | 0.3223 |
| ClustalW (fast) | **0.2317** | 0.6651 | 0.2680 | 0.2513 | 0.2808 | 0.2752 |
| MUSCLE | 0.3217 | 0.6961 | 0.3077 | 0.3087 | 0.3484 | 0.3397 |
| ClustalW | 0.2395 | 0.6417 | 0.2602 | 0.2478 | 0.3024 | **0.2658** |
| PSAlign | **0.3503** | **0.7384** | **0.3517** | 0.2992 | **0.3951** | 0.3816 |
| T-Coffee | 0.2716 | 0.6986 | 0.3257 | **0.3637** | 0.3659 | **0.3974** |

finishes in 33% of the time required by ClustalW using `-quicktree`, and only 8% needed by ClustalW, possibly the most widely used MSA program.

**Long Sequence Experiments**

In order to compare the performance of MSA algorithms on long data sets, two sets of seven FASTA files each containing ten sequences were generated using Rose version 1.3. The first set of seven FASTA files contains protein sequences and the second set contains DNA sequences. In both sets, the first file contains sequences with an average length of 5,000 residues, with each file increasing the average sequence length by 5,000 residues. Thus, the seventh file contains ten sequences with an average sequence length of 35,000 residues.

Figures 3.11 and 3.12 depict the execution time required for the fastest algorithms to align the seven large protein and DNA sequence sets, respectively. As the average length of sequences increases, the difference in time required by GramAlign compared

Table 3.3: Execution time necessary to align all trials in the BAliBASE test suite.

| **Algorithm** | **Execution Time (sec)** |
|---|---|
| MUSCLE (fast) | 301 |
| MAFFT (fast) | 313 |
| MAFFT v6.240 | 341 |
| MAFFT | 564 |
| GramAlign | 680 |
| Kalign | 1,329 |
| ClustalW (fast) | 2,071 |
| MUSCLE | 6,129 |
| ClustalW | 8,720 |
| PSAlign | 152,168 |
| T-Coffee | 403,815 |

to the other algorithms also increases. In particular, at an average sequence length of 35,000 residues GramAlign completes the alignments in 3,363 and 3,092 seconds, while the nearest algorithm (MAFFT in fast-mode) requires 10,362 and 6,981 seconds. That is, GramAlign finishes in 32% and 44% of the time required by the next fastest algorithm.

MUSCLE in fast mode encountered a segmentation fault during the Root Alignment step while running on the longest test sequences, and so the execution time is not included in Figures 3.11 and 3.12.

### Numerous Sequence Experiments

In order to compare the performance of MSA algorithms on data sets with many sequences, two sets of seven FASTA files each containing sequences with an average length of 100 residues were generated using Rose version 1.3. The first set of seven FASTA files contains protein sequences and the second set contains DNA sequences. In both sets, the first file contains 100 sequences, with each file increasing the number of sequences up to the seventh file, which contains 10,000 sequences.

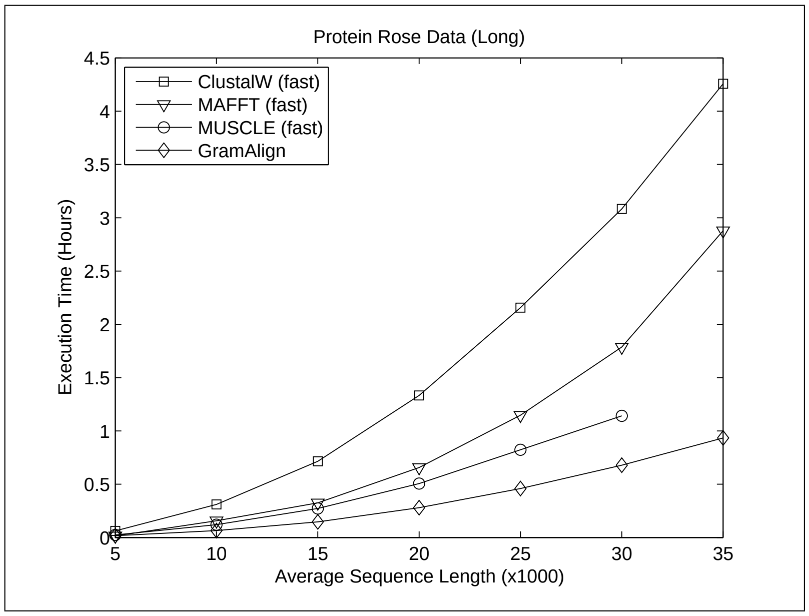


Figure 3.11: Result of executing the fastest algorithms on the Rose-generated long protein sequence sets.

As shown in [52], the authors of MAFFT added a new heuristic method for generating a spanning tree referred to as "PartTree". The increase in performance is dramatic and intended for data sets involving many sequences. Thus, for this set of experiments, MAFFT version 6.240 was added with the command line `mafft --retree 1 --parttree --partsize 50`, which matches the fastest algorithm presented in [52].

Figures 3.13 and 3.14 depict the execution time required for the fastest algorithms to align the seven large protein and DNA sequence sets, respectively. As the number of sequences increases, the difference in time required by GramAlign and MAFFT v6 compared to the other algorithms also increases. In particular, on the sets containing 10,000 protein and DNA sequences GramAlign completes the alignments in 162 and 68 seconds and MAFFT v6 completes the alignments in 119 and 71 seconds, while the next closest algorithm, MUSCLE in fast-mode, requires 621 and 456 seconds. That is, GramAlign finishes in 26% and 15% of the time required by the next fastest

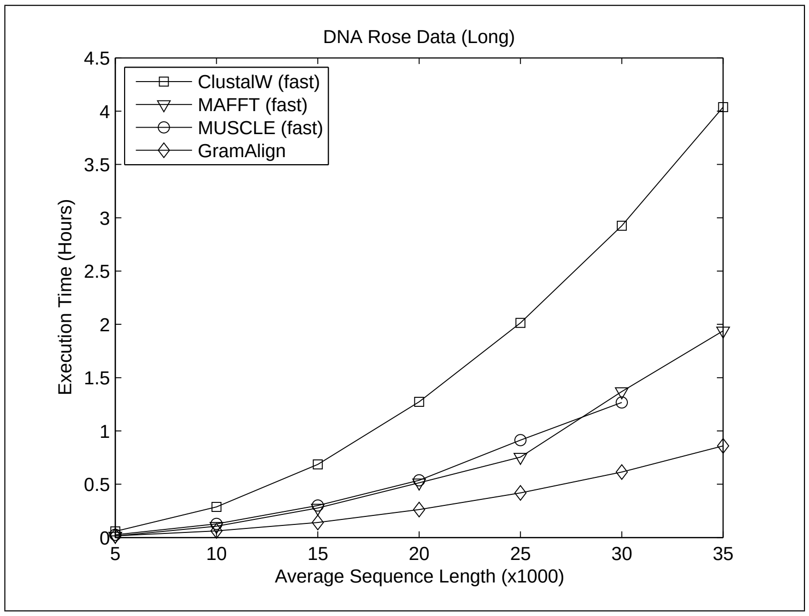


Figure 3.12: Result of executing the fastest algorithms on the Rose-generated long DNA sequence sets.

algorithm other than MAFFT v6.

The results imply the promising viability of GramAlign, especially when aligning either long or numerous sequences such as in whole-genome applications. Further, better alignment scores may be achieved with little change in execution time via the user-alterable parameters.

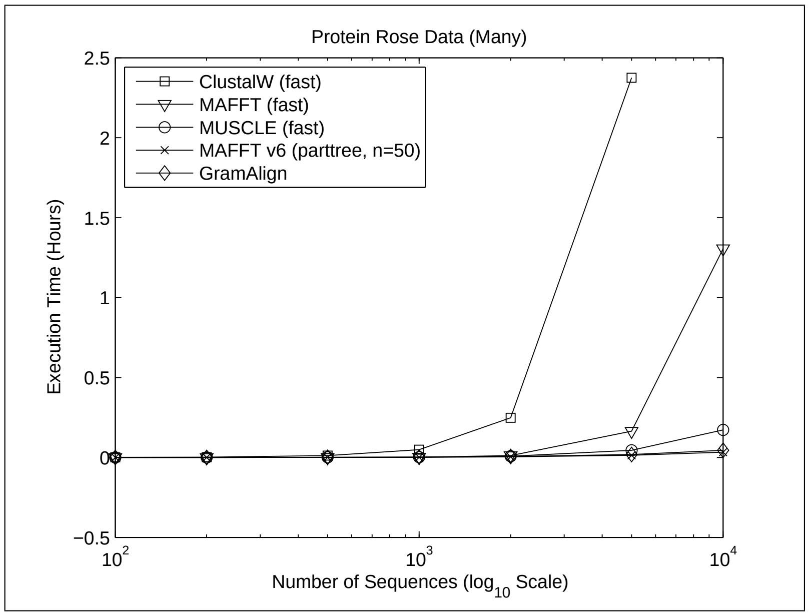


Figure 3.13: Result of executing the fastest algorithms on the Rose-generated numerous protein sequence sets.

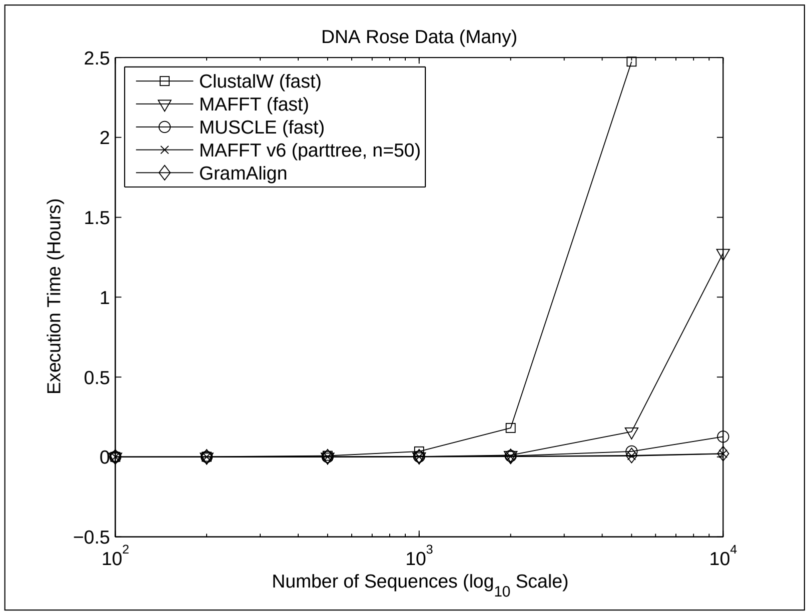


Figure 3.14: Result of executing the fastest algorithms on the Rose-generated numerous DNA sequence sets.

## 3.3 Multiple Contig Arrangement Algorithm

The second sequence alignment application developed is GramContig, a fragment-to-reference sequence alignment method. Sequence assembly is a fundamental process in computational biology because the current DNA sequencing tools are able to generate only short sequence fragments from the source DNA material, on the order of 20 to 1000 bases in length. There are two different types of sequence assembly methods: 1) *de-novo* assembly refers to constructing a new sequence from a set of fragments without any additional information, while 2) *mapping* assembly uses an existing reference sequence to guide the arrangement of a set of fragments. Evidently, the reference sequence used in the latter method needs to represent a good template for aligning the fragments.

For the mapping algorithm presented here, the grammar-based distance between a small portion of a biological sequence–a fragment or *contig*–and windowed regions of a complete reference sequence are used to identify the contig's location relative to the reference sequence. Then, a set of contigs can be processed resulting in a newly assembled sequence.

One problem that commonly occurs with DNA sequencing technology is coverage. Sometimes regions of the source DNA material will not be represented in the output set of contigs. This kind of fragment assembly is useful in locating regions of gaps implying portions that may not have been properly sequenced. By locating the gap-filled areas, the newly mapped sequence can be used to create the *primers*, or small fragments used to start sequence replication, necessary to sequence the missing subsequences. A biologist would be able to use these primers to sequence the source DNA again with specific focus on the missing regions.

A general overview of the GramContig algorithm is depicted in Figure 3.15. A

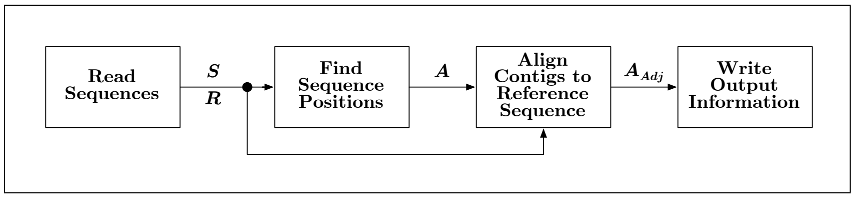


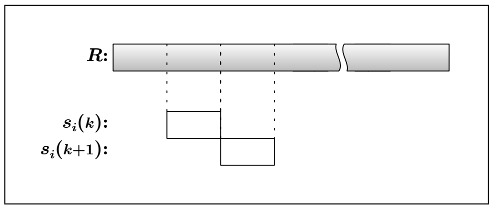
Figure 3.15: The algorithm operates on a reference sequence, $R$, and a set of contig sequences, $S$, originally read in FASTA format. The reference sequence is searched for the occurrence of each contig sequence, resulting in an initial contig alignment, $A$. Based on the positions in $A$, each contig is pairwise aligned to the respective subsequence of $R$, resulting in a final assembled sequence, $A_{adj}$.

single reference sequence, $R$, and the set of contig sequences to be aligned, $S$, are regarded as input to the algorithm with $S = \{s_1, ..., s_N\}$, where $s_i$ is the $i$th contig sequence and $i \in \{1, ..., N\}$.

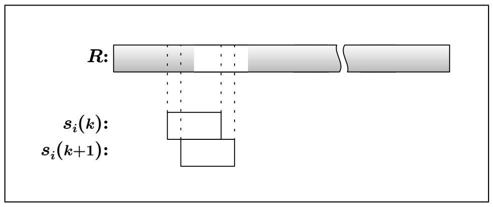
### 3.3.1 Find Sequence Positions

The first step in the procedure involves identifying the approximate alignment position of each sequence $s_i$ relative to the reference sequence $R$. The method implemented in GramContig uses a two-pass approach. The first scan identifies the neighborhood within $R$ to which $s_i$ is most likely to be aligned. As shown in Figure 3.16(a), a coarse scan is performed with a jumping window along $R$. In particular, a window length of $|s_i|$ is used to divide $R$ into $\lceil |R|/|s_i| \rceil$ non-overlapping subsequences. The same grammar-based distance calculation used in GramAlign is repeatedly applied to $s_i$ and each subsequence of $R$. At each position, the grammar-based distance is also determined between the reverse complement of $s_i$ and the respective subsequence of $R$. The overall lowest distance is recorded and the associated subsequence and its direction is marked as the neighborhood within $R$ most likely to be aligned with $s_i$.

Once the initial region in $R$ has been identified, a second scan is used to refine the position prior to the final alignment step. Referring to Figure 3.16(b), the second scan



(a) Jump Window



(b) Fine Window

Figure 3.16: Finding each contig position relative to the reference sequence, $R$, involves two search rounds. A grammar-based distance is calculated between the contig and a portion of $R$ defined by a window length relative to the contig length. (a) The first search procedure scans $R$ via a coarse jumping window. (b) After identified, the initial position in $R$ is refined with a sliding window.

begins upstream of the position determined in the first scan. The scan includes the subsequence of $R$ taken from a sliding window that shifts from its upstream position across the initial neighborhood until a portion of the window is downstream of the coarse scan results. Again, the lowest distance and respective subsequence starting position is recorded for the final alignment step.

### 3.3.2 Align Contigs to Reference Sequence

Once the approximate positions of each $s_i$ have been recorded, the final step is to align them to $R$. In particular, the same pairwise alignment procedure used in GramAlign is applied to each $s_i$ and an appropriate subsequence of $R$. Given the starting position $p_i$, the subsequence starting position is given by $p_i - (|s_i| \times P)$, where $P$ is a user defined percentage. Similarly, the ending position is given by $p_i + (|s_i| \times (1 + P))$. The extra bases on either side of the predetermined subsequence allow for error in the scanning process as well as the potential for evolutionary differences between $s_i$ and the region in $R$.

Once the pairwise alignment is performed for each contig, the final starting positions are recorded in an output file. Additionally, a new composite sequence is created in which each aligned contig is inserted at its determined location. The resulting two-sequence aligned FASTA file contains a copy of the reference sequence, $R$, and the composite contig alignment sequence, $A_{Adj}$.

### 3.3.3 Example Simulations

In this section, an example contig alignment is used to illustrate the use of GramContig. The result was generated by compiling and executing GramContig and its associated graphical user interface (GUI) on an Apple MacBook Pro with an Intel Core 2 Duo operating at 2.53 GHz with 4 Gb of system memory and a 3 Mb L2 cache. The reference sequence used was the complete genome of *Francisella tularensis* obtained from the NCBI website (http://www.ncbi.nlm.nih.gov) using the accession number NC_009257.1. The set of contig sequences were obtained as the result of a 454 pyrosequencing operation.

Once GramContig has been executed on the data set, the alignment information is made available to the user. As a supplemental program, the GUI depicted in Figure 3.17 provides the ability to view the resulting assembly alignment. This screen

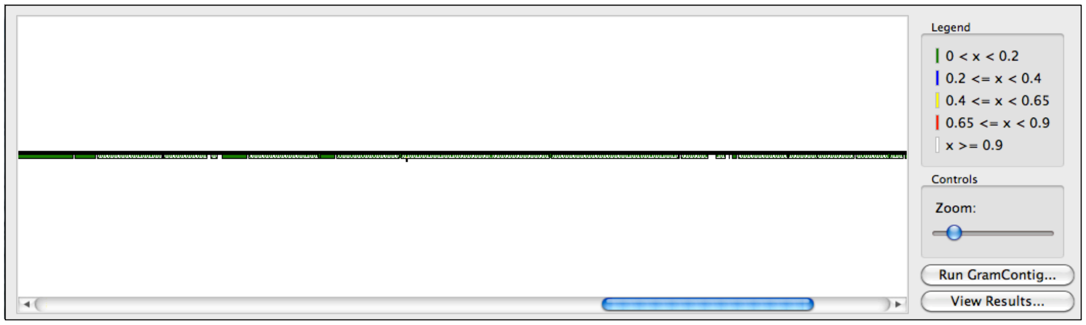


Figure 3.17: A screen capture of the GUI displaying the final GramContig alignment results. The zoomed-out window allows the user to quickly scan the overall result for interesting areas.

capture shows a “zoomed-out” version of the final alignment. It allows the user to quickly scroll along the total alignment in search of any interesting areas that might require additional attention. The user can increase visibility by “zooming-in” on the current position within the viewing window, an example of which is shown in Figure 3.18. This screen capture provides additional detail. In particular, the black

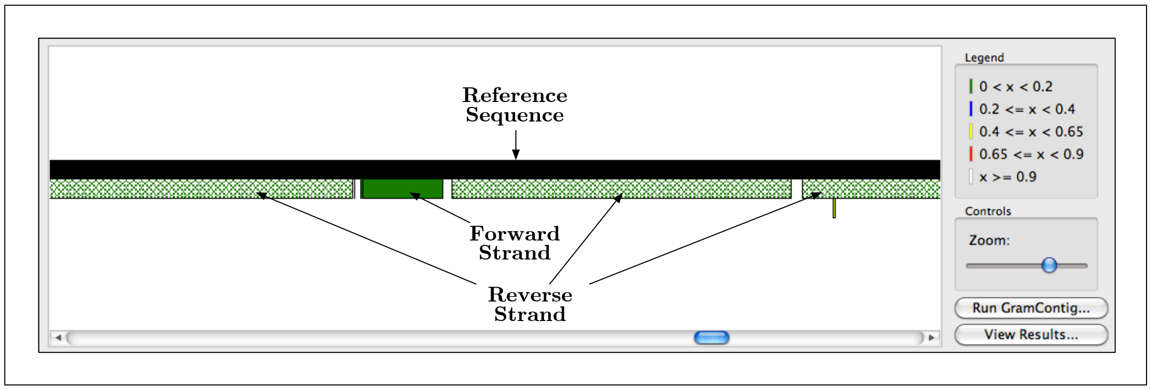


Figure 3.18: A screen capture of the GUI displaying the final GramContig alignment results. This zoomed-in window provides the user with a more detailed picture of the alignment. The black bar on top represents $R$, and each bar on the bottom represent a specific $s_i$. The solid bars on the bottom represent contigs aligned in the forward direction while the cross-hatched bars on the bottom represent contigs aligned to the reverse of $R$. The color of each bar indicates the final grammar-based distance, and so implies the confidence of its location.

bar along the top of the alignment represents the reference sequence, while the colored bars along the bottom of the window detail the contigs as they are aligned to $R$. The base color of each contig is meant to imply the confidence of the identified location and its resulting alignment. The alignment region in the window of Figure 3.18 shows four green contigs meaning their grammar-based distance is each within the high-confidence interval, $d_i \in [0.0, 0.2)$. Additionally, there is a very small contig that was identified in a region already occupied by a larger contig. It is also presented in the window on a second row below the primary row. While it is difficult to see in the figure, its color is yellow implying its grammar-based distance falls in a lower-confidence interval, $d_i \in [0.4, 0.65)$. In addition to the color, each bar is also shaded with a pattern that implies the direction relative to $R$ with which the contig is aligned. In the example, there is one solid bar that indicates the associated contig is aligned along the forward strand of $R$. The other three contigs have a cross-hatched pattern implying they are all aligned along the reverse strand of $R$.

One of the primary applications of GramContig is identifying potential gaps in a pyrosequencing outcome. For example, the zoomed-in area depicted in Figure 3.19 shows a significant gap between adjacent contigs. This indicates the likelihood of

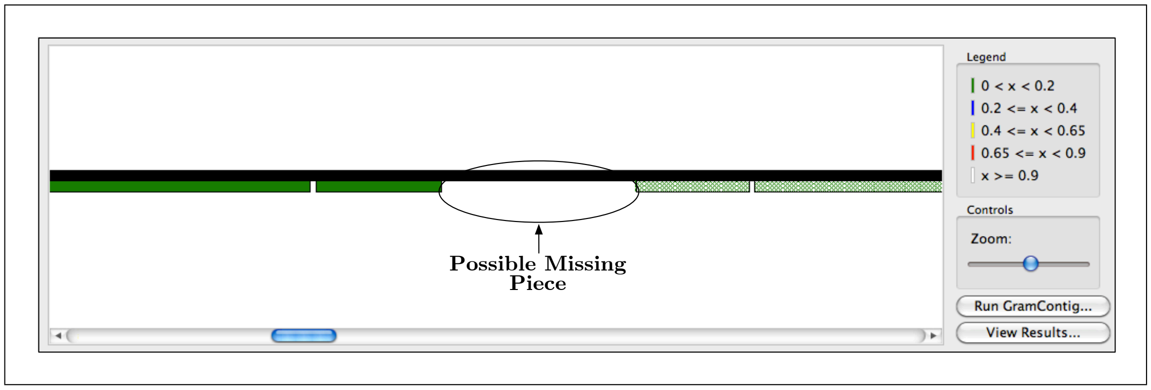
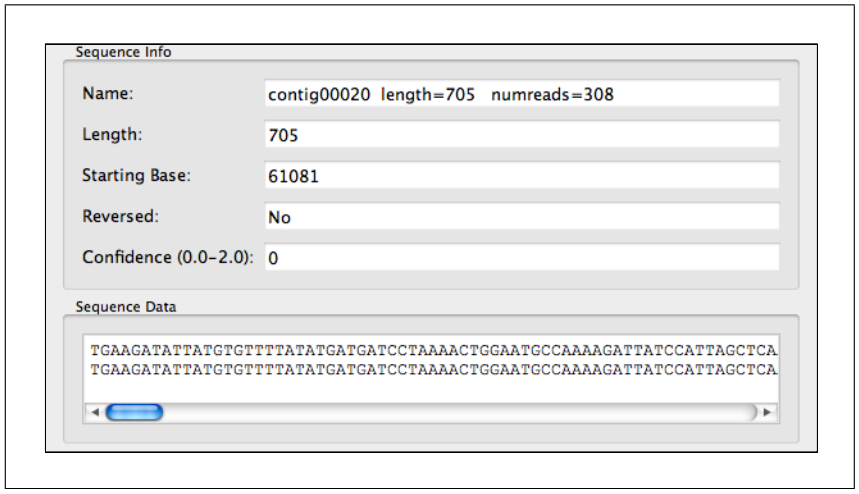


Figure 3.19: A screen capture of the GUI displaying the final GramContig alignment results. This zoomed-in area shows a relatively large gap in the alignment, which implies the possibility of a missing contig in the data set.

a region that was not properly sequenced in the original procedure. At this point, the user might want to re-sequence the specific area in question. That is, the user might only be interested in the small region of the organism, without the inclusion of the rest of the genome. What they require is the identification of a primer, which is a portion of the genome from which replication can begin. The user can look at the sequence occupied by the contig immediately upstream of the gap. The GUI allows for the user the ability to click on any contig to view the individual alignment in addition to various statistics. As an example, Figure 3.20 shows the information associated with "contig00020" that was aligned along the forward strand of $R$ starting


Sequence Info

| | |
|---|---|
| Name: | contig00020 length=705 numreads=308 |
| Length: | 705 |
| Starting Base: | 61081 |
| Reversed: | No |
| Confidence (0.0–2.0): | 0 |

Sequence Data

```
TGAAGATATTATGTGTTTTATATGATGATCCTAAAACTGGAATGCCAAAAGATTATCCATTAGCTCA
TGAAGATATTATGTGTTTTATATGATGATCCTAAAACTGGAATGCCAAAAGATTATCCATTAGCTCA
```

Figure 3.20: A screen capture of the GUI displaying the statistics associated with a specific contig alignment. This window appears in response to the user clicking on any one of the contigs in the main window. The details include the header name provided in the source FASTA file, the contig length, the starting position relative to $R$, the direction of alignment, the final grammar-based distance, and the final alignment which the user can scroll across.

at base 61,081. Of particular interest to finding a primer, the user has the ability to view the entire alignment between the contig and $R$ by scrolling along the lower pane within the window. This particular contig had a grammar-based distance of 0.0 which implies an exact match between the contig and the subsequence of $R$, which is further demonstrated by the portion of alignment shown in the lower pane.

## 3.4 Conclusions

This chapter introduced two alignment applications and their respective algorithms. The grammar-based distance work presented in [79] was adapted to generate a numeric metric useful in each application. Additionally, a merged amino acid alphabet was determined to allow an improved grammar-based distance when operating on protein sequences. Results from extensive alignments were presented in an attempt to study the overall quality of the resultant alignments as well as the computation time necessary to achieve the alignments. Correctly aligning multiple biological sequences in an efficient amount of time is an important and challenging problem with a wide spectrum of applications. In this chapter, we adapt existing ideas in a novel way introducing innovative improvements.

The next chapter introduces another problem found in computational biology, that of clustering related sequences together to reduce database search time. The grammar-based distance used in the alignment applications is modified for the developed grammar-based clustering program, which is able to generate high-quality clusters.

# Chapter 4

# Biological Sequence Clustering

The next application of grammar-based models is on the problem of sequence clustering. Clustering involves placing similar sequences from a set into a common group; thereby creating a partitioning of the sequences. A single sequence from each partition can be used as a representative of all sequences in the cluster. This issue applies to database searching problems. Specifically, the amount of biological sequence information present in public databases is already enormous and growing rapidly. Unfortunately, many entries are actually quite redundant in that very similar or identical entries are already in place. Therefore, applications that are able to remove the redundancy by clustering can greatly reduce database search times. A specific example application, GramCluster, was developed to determine high-quality clusters based on a modified version of the grammar-based distance used in Chapter 3.

## 4.1 Sequence Clustering Background

The amount of biological information being gathered is growing faster than the rate at which it can be analyzed. Data clustering, which compresses the problem space by reducing redundancy, is one viable tool for managing the explosive growth of data. In general, clustering algorithms are designed to operate on a large set of related values, eventually generating a smaller set of elements that represent groups of similar data points. A central data element may then be used as the sole representative of a group.

Significant clustering work relating to bioinformatics may be traced to the late 1990s when methods for quick generation of nonredundant (NR) protein databases were developed. These combined identical or nearly identical protein sequences into single entries [45, 60, 61]. The primary benefits of these methods include faster searches of the NR protein databases and reduced statistical bias in the query results [45]. Similarly, computer programs such as those in ICAtools [80] were developed for compressing DNA databases by removing redundant sequences found via clustering resulting in faster database queries. Note that the use of the term “clustering” in these applications differs from another use often found in the literature where clustering refers to generating a phylogenetic distance matrix, such as in [9]. The operation of clustering used in this work identifies groups of sequences related by phylogeny; and it additionally applies to redundancy removal by identifying a sequence that suitably represents similar sequences.

Recently, DNA/RNA clustering has attracted attention for a variety of reasons. The drive to lower the expense of genome sequencing has led to the development of high-throughput sequencing technologies capable of generating millions of sequence fragments simultaneously. A clustering preprocessing step can be used to remove a great amount of fragment redundancy which, in turn, allows for quicker fragment reassembly.

One of the more popular DNA/RNA clustering algorithms is CD-HIT-EST [59] which was based on the protein clustering methods of [60, 61] and was developed for clustering DNA/RNA database data such as non-intron-containing expressed sequence tags (ESTs).

A major application of CD-HIT has been for clustering large data sets from microbiota analysis (e.g. [19]), often as a preprocessing step to create sets of highly related sequences representing operational taxonomic units (OTUs). These OTUs are subsequently used as a basis for estimating species diversity between treatment groups or quantitative relationships of taxa between treatment groups. Alternatively, representative sequences from the OTUs are used for phylogeny-based analyses.

A recent effort in [29] to develop software tools which reduce the time required by BLAST [4] to search large biological databases has resulted in a set of programs, including UBLAST and USEARCH, that reduce the search time by orders of magnitude. As part of the work, an additional clustering program called UCLUST was created which utilizes the heuristic algorithm provided by USEARCH. UCLUST generates results that dramatically improve upon the time required by CD-HIT.

The next sections present GramCluster, a fast and accurate algorithm for clustering large data sets of 16S rDNA sequences based on the inherent grammar of DNA and RNA sequences. Lempel-Ziv parsing [55] is used to estimate the grammar of each sequence to provide a distance metric among sequences. The implementation of this algorithm allows for fast and accurate clustering of biological information. The following sections describe the algorithm and present results, including comparisons with the CD-HIT-EST algorithm and the recently developed UCLUST algorithm.

## 4.2 Sequence Clustering Algorithm

A general overview of the GramCluster algorithm is shown in Figure 4.1. The set of sequences, $S$, is regarded as input to the algorithm with $S = \{s_1, ..., s_N\}$, where $s_i$ is the $i$th sequence and $i \in \{1, ..., N\}$. The goal of the algorithm is to partition $S$ where each sequence is grouped with similar sequences from $S$ such that all sequences within each resulting cluster are more similar to each other than sequences from other clusters. The final partition is represented by the set of clusters, $C = \{c_1, ..., c_M\}$, where $c_j$ is the $j$th cluster and $j \in \{1, ..., M\}$. The algorithm initially generates a suffix tree, $t_i$, and grammar dictionary, $d_i$, associated with each sequence, $s_i$. For each sequence, $s_i$, these data structures are used to determine if an existing cluster contains sufficiently similar sequences to $s_i$ or if a new cluster needs to be created. If a cluster, $c_j \in C$, already exists with similar sequences, the sequence $s_i$ is added to $c_j$. However, if no cluster contains similar sequences, a new cluster containing only $s_i$ is added to $C$. This clustering continues for all sequences in $S$. The algorithm is described in more detail below with reference to the various blocks in Figure 4.1.

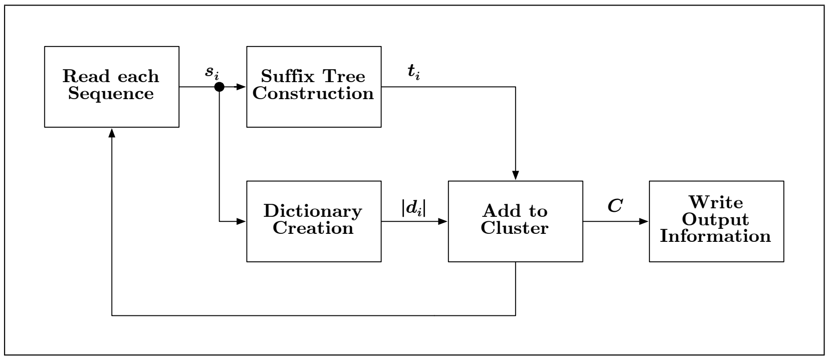


Figure 4.1: The algorithm operates on each sequence, $s_i$, which is parsed into a suffix tree, $t_i$, and dictionary, $d_i$, for rapid distance comparison with other sequences. Each sequence is either added to an existing cluster, $c_j \in C$, or becomes the initial representative sequence in a new cluster, $c_k$.

### 4.2.1 Dictionary Creation

One of the core processes of the clustering algorithm is the formation of a distance estimate between an unprocessed sequence, $s_i$, and each cluster, $c_j$, already in the partition, $C$. To this end, one sequence, called the representative sequence, is used to represent all other sequences within each cluster. The distance between $s_i$ and $s_{r_j} \in c_j$, where $s_{r_j}$ represents $c_j$, is used to determine if $s_i$ should be added to $c_j$.

Each sequence, $s_i$, is compared with, at most, the set of representative sequences, $\{s_{r_j} \mid s_{r_j} \text{ represents } c_j \in C\}$, to discover the correct cluster for $s_i$.

The distance metric relies on the structural rules necessarily present in all information containing sequences. GramCluster uses the grammar estimation method based on Lempel-Ziv (LZ) parsing [55, 105, 106] as used in [7] for language-phylogeny inference, in [79] for phylogeny reconstruction, and in [85] and Chapter 3 to construct a guide tree for multiple sequence alignment. A similar grammar-based distance is also the focus of [83] which analyzes the quality of the distance metric as a function of the length of the sequences.

The primary aspects of LZ dictionary creation are shown in Figure 4.2 where a set of grammar rules for each sequence is calculated. Initially, the dictionary, $d_i^1 = \emptyset$, is empty, a fragment, $f^1 = s_i(1)$, is set to the first residue of the corresponding sequence, and only the first element, $s_i(1)$, is visible to the algorithm. At the $k$th iteration of the procedure, the $k$th residue is appended to the fragment resulting from the $(k-1)$th step; and the visible sequence is checked. If $f^k \notin s_i(1, ..., k-1)$, then $f^k$ is considered a new rule and so added to the dictionary, $d_i^k = d_i^{k-1} \cup \left\{f^k\right\}$; and the fragment is reset for the start of the next iteration, $f^k = \emptyset$. However, if $f^k \in s_i(1, ..., k-1)$, then the current dictionary contains enough rules to reproduce the current fragment, i.e., $d_i^k = d_i^{k-1}$. In either case, the iteration completes by appending the $k$th residue to the visible sequence. This procedure continues until the visible sequence is equal to the entire sequence, at which time the size of the dictionary, $\left|d_i\right|$, is determined for use in the metric calculation. The distance between the sequences is estimated using the dictionary sizes. Intuitively, sequences with a similar grammar should be clustered together. The correlation of the LZ-based distance with phylogenetic distance was exploited in [79] to obtain phylogenies for a set of mammalian species using complete mitochondrial DNA and for the superfamily *Cavioidea* using exon#10 of the growth hormone receptor (GHR) gene, the transthyretin (TTH) gene, and the 12S rRNA gene. In [6], the same distance metric was used to obtain phylogenies for fungal species using the cytochrome b gene and internal transcribed spacer regions of the rDNA gene complex.

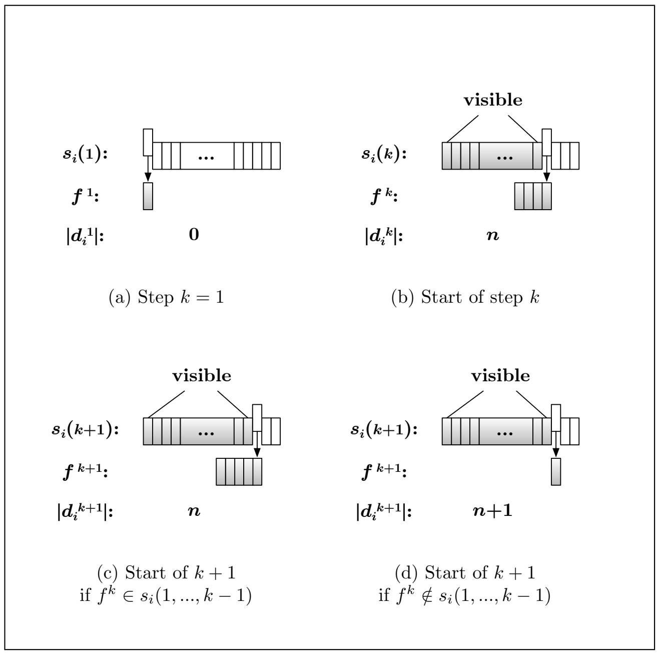


Figure 4.2: Determining the order of the LZ dictionary, $|d_i|$, for sequence $s_i$. (a) The initial step in which the initial fragment, $f^1$, is set to the first letter, $s_i(1)$, of the sequence. (b) The start of the $k$th step in which the $k$th letter, $s_i(k)$, is appended to the current fragment, $f^k$. After the first $k-1$ letters of $s_i$ are scanned for the occurrence of the fragment, $f^k$, the two possible outcomes are (c) the fragment is reproducible with combinations of existing rules, or (d) the fragment is unique up to this point in the sequence, and so a new grammar rule is added to the dictionary and the fragment is reset.

### 4.2.2 Suffix Tree Construction

As shown in Figure 4.1, the algorithm also constructs a suffix tree for the sequence. Suffix trees are data structures designed to contain all $L$ suffix substrings of a length-$L$ sequence [102, 64, 100]. For example, a suffix tree for the sequence “gagacat” is schematically shown in Figure 4.3. All seven suffixes {gagacat, agacat, gacat, acat,

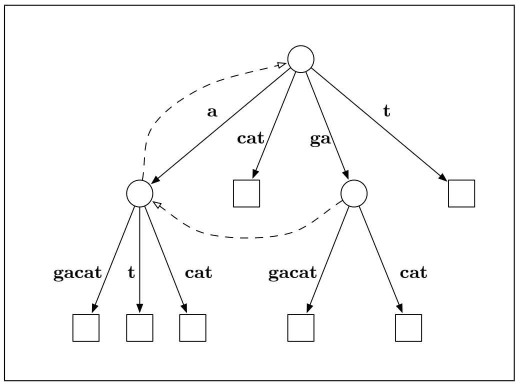


Figure 4.3: Completed suffix tree diagram of the string "gagacat." Tracing a path from root to leaf along a solid line results in a suffix of the string. The dashed lines indicate suffix links that are useful during the creation of the suffix tree.

cat, at, t} are found by tracing a unique path from the root node to one of the seven leaf nodes along solid lines. One valuable use of suffix trees is searching for substrings which can be thought of as the prefix of a suffix. By using a suffix tree, a length-$L$ sequence can be completely scanned for a length-$F$ fragment in $\mathcal{O}(F)$ time as opposed to $\mathcal{O}(L)$ for a brute force search. Also depicted in Figure 4.3 are the dashed-line suffix links which are a fundamental feature for linear-time construction of the suffix tree [100]. A sequence, $s_i$, can be converted into a suffix tree, $t_i$, in linear time and then searched for substrings in linear time based on the fragment length. As will be shown, suffix tree sequence representation is important for reducing the time required for GramCluster to complete all necessary grammar-based comparisons.

### 4.2.3 Clustering

The final component of the algorithm depicted in Figure 4.1 is represented by the block labeled, "Add to Cluster." The procedure for adding a sequence to a cluster is shown in greater detail in Figure 4.4. The algorithm checks each cluster, $c_j \in C$, until a cluster is found where the distance between the representative sequence, $s_{r_j}$, and $s_i$, $D_j = \text{dist}(s_i, s_{r_j})$, is less than a user-defined threshold, $T$. Once this condition is met, the cluster is updated, $c_j = c_j \cup \{s_i\}$; and processing in this block terminates. If no clusters meet the condition of $D < T$, a new cluster is created with $s_i$ as its first member.

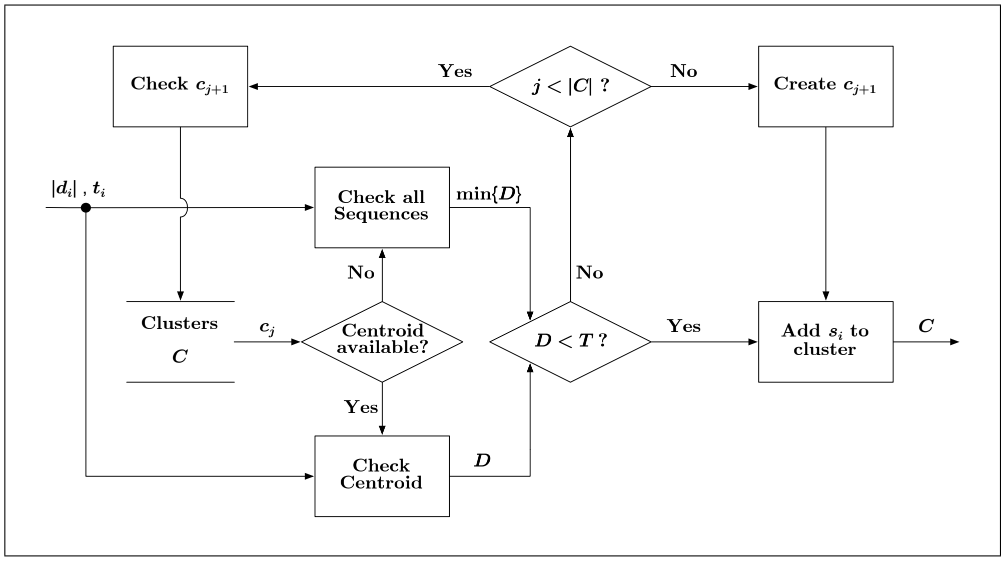


Figure 4.4: A block diagram detailing the process by which sequence $s_i$ is added to a cluster, $c_j \in C$. A distance, $D$, is generated between $s_i$ and the representative of $c_j$. If $D$ is below a user-specified threshold, $T$, then $s_i$ is added to $c_j$, otherwise the next cluster, $c_{j+1}$, is checked. If no cluster is identified as suitable for $s_i$, a new cluster containing $s_i$ is created and added to $C$.

The following sections describe the cluster data structure, the representative sequence selection method, and the grammar-based distance calculation.

## Cluster Data Structure

In order to follow the cluster classification process, it is helpful to understand the data structure used to represent each cluster. In particular, every cluster uses a list of suffix trees, $t_i$, and dictionary sizes, $|d_i|$, to identify its set of sequences. The remaining components contained in the data structure are used to determine and specify the representative sequence, $s_{r_j}$, of the cluster, $c_j$. A good selection for $s_{r_j}$ is a sequence that appears grammatically similar to all other sequences within the cluster. This implies the need to estimate the grammar-based distance between all sequences of the cluster, a computationally expensive task. To avoid this cost, GramCluster selects only a few specific sequences in the cluster, that we will call "basis sequences," to which all others are compared. The representative sequence, $s_{r_j}$, can be determined by considering the sets of relative distances between all sequences and each basis sequence. The centroid of the cluster is then defined as the vector containing the mean values of each set of relative distances. The sequence with relative distances nearest to the centroid is selected as $s_{r_j}$.

To see why this method is effective, consider that clustering is often performed in vector spaces where each element being classified is specified by a vector. The points spatially near each other are placed into the same cluster, and the representative is typically selected as the point that is closest to the center of the cluster. This idea is adapted in GramCluster, with an example depicted in Figure 4.5. The example in the figure contains forty sequences plotted in a two-dimensional space. Each dimension represents the grammar-based distance between the plotted sequence point and a basis sequence. The data set used in this example contained forty 16S rDNA sequences each from four genera (*Acetobacter*, *Achromobacter*, *Borrelia*, *Flavobacterium*). Of the two initially selected basis sequences, one came from *Acetobacter* and the second from *Flavobacterium*. Then, the pair of distances between each sequence and the basis sequences was computed and plotted. As can be seen from the plot, the sequences group into clusters which correspond to their genus. Note that the basis sequences are not orthogonal; however, use is made of the fact that the grammar-based distances

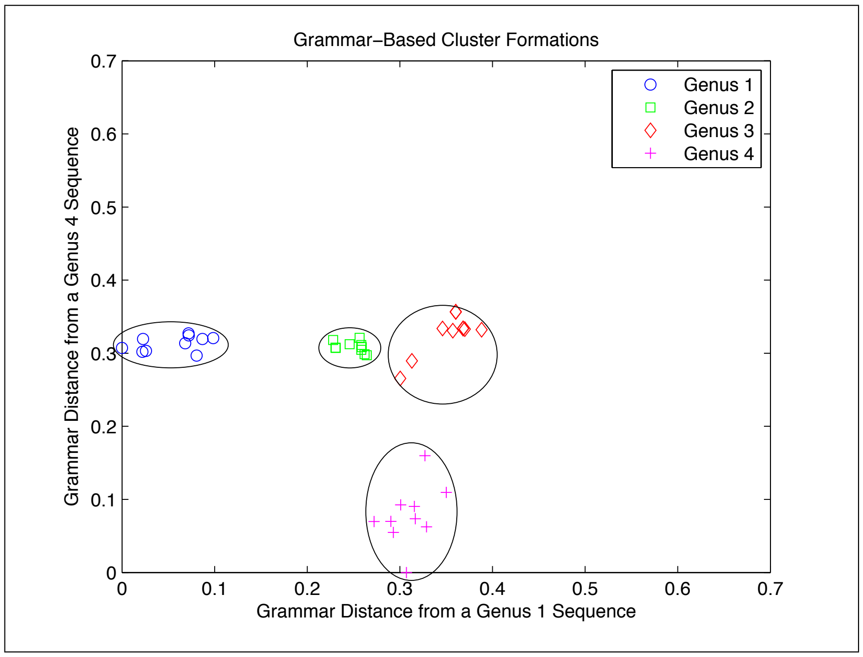


Figure 4.5: Forty sequences being processed via a vector quantizer. Each of the four genera is represented by ten sequences. Every sequence is grammatically compared to the same two sequences from within the set. The resulting pair of distances form two-dimensional vectors in a space. When considering the clusters in this space, the representative sequence of the cluster should be the sequence that is nearest the cluster center.

tend to obey the transitive property such that if

$$D_b = \text{dist}(s_a, s_b)$$

$$D_c = \text{dist}(s_a, s_c)$$

and if $D_b$ is close to $D_c$, then $s_b$ and $s_c$ tend to be grammatically similar to each other. The example in Figure 4.5 demonstrates this by the use of basis sequences from *Acetobacter* (genus one) and *Flavobacterium* (genus four). One would expect that comparing all sequences to one sequence would provide separation between the sequences from the same genus as the basis sequence and the rest. However, sequences from the other genera also form into clusters as a result of sequences being compared to a single basis sequence. In our example, all forty sequences are compared to just two sequences; and four clear clusters appear.

For comparison, 100,000 randomly generated sequences are processed using the same concept. Each sequence is composed of 1,000 randomly selected RNA bases. The first two sequences are the basis sequences, to which all other sequences are grammatically compared. As can be seen from the distance vectors plotted in Figure 4.6, unrelated sequences tend to have a grammar-based distance that is within the range

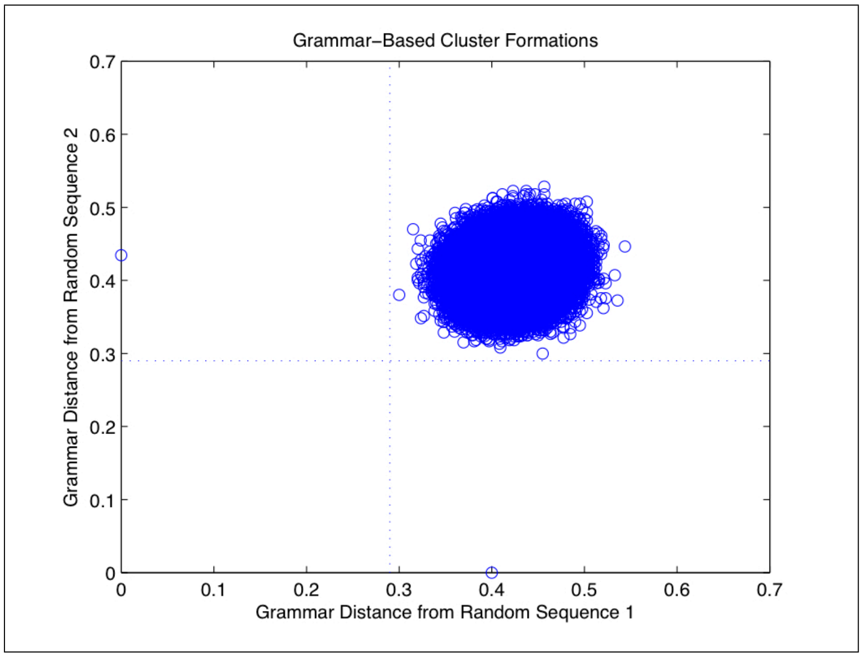


Figure 4.6: 100,000 randomly generated sequences being processed via a vector quantizer. Each sequence is composed of 1,000 randomly selected bases. Every sequence is grammatically compared to the same two sequences from within the set. The resulting pair of distances form two-dimensional vectors in a space. All sequence distances are between 0.29 and 0.55.

$[0.29, 0.55]$. The information depicted in Figure 4.6 implies a certain confidence level for a grammar-based distance calculation between two unknown sequences. In particular, if a distance is below the 0.29 lower interval, then a structural relationship between the sequences is likely to exist. For example, referring back to Figure 4.5, the sequences from *Achromobacter* (genus two) have enough grammar-based structure relative to *Acetobacter* (genus one) to separate them from *Borrelia* (genus three).

The method presented here for building vectors of distances relative to basis sequences is similar to the concept of embedding presented in [9]. The work of [9] details an algorithm called mBed that operates on a set of sequences to generate a distance matrix representing a phylogenetic guide tree, a process that is closely related to the data clustering problem presented here. The mBed algorithm selects a subset of $t$ seed reference sequences that are not close together relative to a distance metric. Then each sequence has a $t$-dimensional vector associated with it where each coordinate value is the distance between the sequence and the respective reference sequence. The distance used in [9] was selected to be the $k$-tuple distance measure of [103] and implemented in ClustalW [97]. The basis sequence concept used in this chapter is similar, with the grammar-based distance metric replacing the $k$-tuple distance measure being the primary difference. Additionally, a single reference subset is used in [9] to build all vectors. The algorithm presented here creates vectors for each sequence contained in a cluster relative to basis sequences also sampled from the same cluster.

**Representative Sequence Selection**

As shown in Figure 4.4, the clustering process begins by comparing sequence $s_i$ to the representative sequence of cluster $c_j \in C$. For clusters containing many sequences, a representative sequence is determined using the basis sequence method described above. In this case, only the representative sequence, $s_{r_j}$, is compared to $s_i$

$$D = \mathrm{dist}(s_i, s_{r_j}).$$

However, the progressive addition of sequences to clusters means there are clusters containing only a few sequences. These clusters do not contain a large enough sample set to yield a reliable representative. Thus, until a cluster is large enough, all sequences are considered representative and compared to $s_i$

$$D_k = \text{dist}(s_i, s_k) \quad \forall\ s_k \in c_j.$$

The minimum distance, $\min\limits_k\{D_k\}$, is used as the classification metric.

**Grammar-Based Distance Calculation**

The distance metric used in GramCluster is a modified form of the grammar-based distance metric introduced in [79, 83] and used in [85] and Chapter 3.

The original distance metric is computed by concatenating the two sequences being compared into a single sequence and then performing the operations detailed in Figure 4.2. Formally, consider the process of comparing sequences $s_m$ and $s_n$. Initially, the dictionary, $d^1_{m,n} = d_m$, is set to that of sequence $s_m$, a fragment, $f^1 = s_n(1)$, is set to the first residue of the $n$th sequence, and the visible sequence is all of $s_m$. The algorithm operates as described previously, resulting in a new dictionary size, $\left|d_{m,n}\right|$. When complete, more grammatically similar sequences will have a new dictionary size with fewer entries as compared to sequences that are less grammatically similar. Therefore, the size of the new dictionary, $\left|d_{m,n}\right|$, will be close to the size of the original dictionary, $\left|d_m\right|$. The distance between the sequences is estimated using the dictionary sizes, in particular

$$D = \begin{cases} \frac{\left|d_{m,n}\right| - \left|d_m\right|}{\left|d_m\right|} \times \frac{\left|s_m\right|}{\left|s_n\right|} & \text{if } \left|s_m\right| > \left|s_n\right|, \\ \frac{\left|d_{n,m}\right| - \left|d_n\right|}{\left|d_n\right|} \times \frac{\left|s_n\right|}{\left|s_m\right|} & \text{if } \left|s_m\right| \leq \left|s_n\right|. \end{cases} \tag{4.1}$$

This particular metric accounts for differences in sequence lengths and normalizes accordingly. Smaller values of $D$ indicate a stronger similarity. Intuitively, sequences with a similar grammar should be clustered with each other.

While this grammar-based distance metric works well, it requires that the extended sequence be rescanned for every residue in the second sequence. This means that $s_m$ will be rescanned completely for every character in $s_n$. This process is repeated as many times as the number of sequences compared to $s_m$. As a result, approximately 75% of the computation is devoted to string searching and concatenation. To improve the execution time, we introduce two significant modifications described below.

**Fragment Markers**

The original distance calculation would simply repeat the process depicted in Figure 4.2 on the concatenation of two sequences being compared. Thus, for the $k$th character in the second sequence, the first sequence is completely scanned along with the initial $k-1$ portion of the second sequence. However, this is quite unnecessary since many fragments formed from the second sequence were already found in the second sequence during the initial scan. Formally, consider sequences $s_m$ and $s_n$ which have already had their own dictionaries created in a previous step. Now suppose the concatenated sequence $s_{m \cdot n}$ is being processed for the $k$th character in $s_n$, at which point there is a nonempty fragment, $f^k$. The process begins with the fragment completely composed of consecutive letters from $s_n$, which means that this fragment has already been created once before when $s_n$ was processed by itself. As long as $f^k$ was previously found within $s_n(1, ..., k-1)$, there will be no new information gained by scanning $s_{m \cdot n}(1, ..., |s_m| + k - 1)$, because it is certain to be there since $s_n(1, ..., k-1) \subset s_{m \cdot n}(1, ..., |s_m| + k - 1)$. So, there is no need to scan for fragments that have been previously found during any distance calculation. The inverse statement is also true: fragments not previously found do need to be scanned for during a distance calculation. This is implemented as shown in Figure 4.7, in which fragment $f^k \notin s_i(1, ..., k-1)$, so $k$ is added to a list of marked fragment indices.

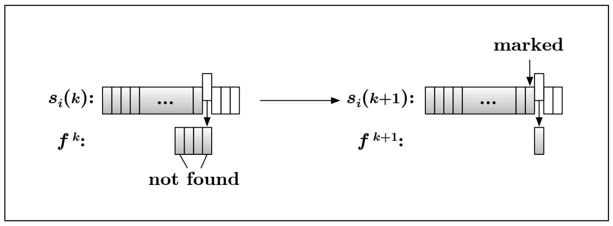


Figure 4.7: One of the implementation optimizations is marking locations in the sequences where fragments are not found in the visible sequence. Doing so eliminates the need to rescan sequences during the distance calculation for fragments that are already known to be found within the original sequence.

The same distance metric given by (4.1) is used, but there is no longer a need to perform string concatenation; and only the first string is scanned for the marked fragments from the second string. Formally, consider the process of comparing sequences $s_m$ and $s_n$. Initially, the dictionary, $d^1_{m,n} = d_m$, is set to that of sequence $s_m$, a fragment, $f^{\text{marked}(1)}$, is set to the first marked substring of the $n$th sequence, and the visible sequence is always just $s_m$. The algorithm simply scans $s_m$ for an occurrence of the fragment and adds one to the dictionary if the fragment is not found. Either way, the fragment is updated to the next marked substring of $s_n$; and $s_m$ is scanned again. This continues for all marked fragments from $s_n$ resulting in a new dictionary size, $\left|d_{m,n}\right|$. This fragment marking process significantly reduces the total number of substring searches performed, as well as the character concatenations that would be otherwise required.

The second optimization involves a time-efficient method of searching a string for a substring of characters, a very relevant problem for suffix trees.

### Suffix Tree Searches

As stated previously, a length-$L$ sequence stored in a suffix tree data structure can be completely scanned for a length-$F$ fragment in $\mathcal{O}(F)$ time. To see why this is true, consider the simple example depicted in Figure 4.3. Every suffix is represented in the data structure as a unique path beginning at the root node and traversing along a solid line to a leaf node. Any substring occurring in this string has to be the start of a suffix, so searching for a substring amounts to finding a suffix that begins with the substring. Consider searching "gagacat" for the substring fragment "gac" which is present in the string. The first step is to find a branch beginning with "g" leaving the root, which is found as the third entry in the data structure. Following the branch to the internal node indicates that all suffixes in this tree that begin with "g" are always followed by an "a," which is also true of the fragment. At the internal node, the next step is to search for any branch that begins with "c," which is found as the second entry in the data structure, concluding the search. Next, consider searching for the substring fragment "gact," which follows the previous search to the internal node and includes identifying the branch beginning with "c." The final step is looking at the subsequent character along the branch, which is "a," and does not match. This search finishes having determined that "gact" is not a substring of "gagacat." The use of the suffix tree in this context means that the time necessary for identifying whether previously marked fragments from sequence $s_n$ are present in sequence $s_m$ is $\mathcal{O}(F)$.

### 4.2.4 Algorithm Complexity

The algorithm complexity of GramCluster may be broken into three pieces, beginning with the generation of each sequence grammar dictionary, $d_i$ for $i \in \{1, ..., N\}$, where $N$ is the number of sequences. Suppose the average sequence length is $L$, then each $d_i$ results in complexity $\mathcal{O}(L)$, so all dictionaries are generated with complexity $\mathcal{O}(LN)$. Next, each suffix tree, $t_i$, has a complexity $\mathcal{O}(L^2)$, so all sequences are converted into trees with complexity $\mathcal{O}(L^2N)$. Finally, suppose the average number of clusters is $M$. As an upper bound, all clusters are scanned until each sequence is classified and each scanning process has complexity $\mathcal{O}(L)$. The result is a total scanning complexity of $\mathcal{O}(LMN)$. Thus, the entire time complexity for GramCluster is $\mathcal{O}(LN + L^2N + LMN)$, which simplifies to $\mathcal{O}(L^2N + LMN)$.

Regarding the memory complexity of GramCluster and continuing with $N$ as the number of sequences, suppose the average sequence header length in the FASTA file is $H$. Because every header line is stored for subsequent file output, this memory complexity is $\mathcal{O}(HN)$. As before, if the average sequence length is $L$, then each sequence is stored in $\mathcal{O}(L)$. The worst-case memory usage for the clusters themselves occurs if every cluster created has an incomplete set of basis sequences. In this case, each cluster has a memory complexity of $\mathcal{O}(C + B + BC + LC)$ where $C$ is the number of sequences held within the cluster and $B$ is the number of basis sequences per cluster. Because there are $N$ sequences stored in memory during this worst-case scenario, a final upper bound on the memory complexity is $\mathcal{O}((H + B + L)N)$ in which the most significant component has a memory complexity of $\mathcal{O}(LN)$.

### 4.2.5 Command Line Options

The following list details the user-definable command line options available in the current GramCluster implementation.

1. `-B <value>` Specify the full basis amount. The value specified in this option represents the number of nonidentical sequences added to a cluster before a centroid sequence is determined. If this option is not specified, the default value is 4 sequences.

2. `-b <value>` Specify the grammar distance identical threshold. The value specified in this option represents the grammar-based distance threshold for two sequences to be consider grammatically identical. When a new sequence is

added to a cluster, it has a distance less than one of the thresholds (specified by -C or -G). In the event that two sequences are very similar (or identical), this threshold prevents the new sequence from becoming a basis sequence. If this option is not specified, the default value is 0.01.

3. `-C <value>` Specify the grammar distance-to-centroid maximum threshold. The value specified in this option represents the grammar-based distance threshold to the centroid sequence. If a distance calculated between a new sequence and the centroid sequence is less than this value, then the new sequence is added to the cluster. If this option is not specified, the default value is 0.13.

4. `-G <value>` Specify the grammar distance maximum threshold. The value specified in this option represents the grammar-based distance threshold to all basis sequences for clusters that do not have a centroid already determined. If a distance calculated between a new sequence and any basis sequence is less than this value, then the new sequence is added to the cluster. If this option is not specified, the default value is 0.13.

5. `-c` Turn on complete cluster searching. This causes the algorithm to scan every cluster for the lowest distance before adding it. The default is greedy cluster searching, which causes sequences to be added to the first cluster presenting a distance lower than the specified thresholds.

6. `-R` Turn on reverse complement checking. This causes GramCluster to check both the input sequence as well as its reverse complement against each cluster representative. The lowest resulting distance is used for classification.

Note that the -C and -G options specify thresholds that function similar to the identity percentage thresholds used by other clustering programs, such as CD-HIT-EST and UCLUST. However, the thresholds function in just the opposite way, whereby sequences are only added if their grammar-based distance is calculated as a value below the threshold value. In contrast, the identity percent thresholds of CD-HIT-EST and UCLUST require sequences to have a metric score higher than the threshold before they are added to the respective cluster.

### 4.2.6 Numeric Simulations

We performed several clustering experiments to validate GramCluster version 1.3. In particular, we used GramCluster to cluster sets of 16S rDNA sequences. The resulting clusters were analyzed for correctness whereby the genus of each sequence was compared to that of all other sequences in the data set. Correct classification is considered when sequences belonging to the same genus fall into the same cluster. Likewise, incorrect classification occurs when sequences belonging to different genera are placed into the same cluster.

Each output set was analyzed using several statistical quality metrics. In each file, the header line of each sequence was replaced by an integer number associated with that sequence's genus. In this way, the resulting clusters could be validated for quality by comparing the header integers with all other entries. In particular, we used three statistical measures, identified in [40], to assess the quality of resulting clusters, including the Rand Statistic, the Jaccard Coefficient, and the Folkes and Mallows Index. In all cases, a count was created based on the pair-wise comparison of each element with all other elements being clustered. When two elements were compared, they fell into one of four possible categories: 1) the pair should be in the same cluster and they are in the same cluster ($SS$), 2) the pair should be in different clusters but they are in the same cluster ($DS$), 3) the pair should be in the same cluster but they are in different clusters ($SD$), and 4) the pair should be in different clusters and they are in different clusters ($DD$). The goal of a clustering algorithm is to obtain maximal values for $SS$ and $DD$ and minimal values for $DS$ and $SD$. The three metrics all operate on combinations of these counts in order to provide an indication as to the quality of actual clustering versus ideal clustering, as follows:

$$s_{\mathrm{RS}} = (SS + DD)/(SS + DS + SD + DD)$$

$$s_{\mathrm{JC}} = SS/(SS + DS + SD)$$

$$s_{\mathrm{FMI}} = SS/\sqrt{(SS + DS)(SS + SD)}.$$

Notice all metrics are bounded between 0 and 1, with 0 being a poor clustering score and 1 a perfect clustering score. Additionally, the in-cluster classification and sequence differentiation percentages

$$s_{\mathrm{in}} = SS/(SS + SD)$$

$$s_{\mathrm{diff}} = DD/(DS + DD)$$

are provided. Given all sequence pair comparisons, the total number that implies a pair of sequences belong to the same genus is ($SS + SD$). Of that total, only $SS$ pairs were actually classified into the same cluster. Thus, the in-cluster classification is the percent of sequence-to-sequence pairs that have correctly clustered sequences together out of all that should be clustered together. Similarly, the total number of sequence pair comparisons that imply two sequences do not belong to the same genus is ($DS + DD$). Out of the total, only $DD$ pairs were correctly separated into different clusters. The sequence differentiation used here was the percent of sequence pair comparisons that have correctly classified sequences apart out of the total that should not be clustered together.

We repeated the first two experiments using two different random permutations of the FASTA file (results not shown). All programs produced very similar results, thereby demonstrating that the order in which sequences are input to the algorithms does not affect the resulting clusters.

In order to identify the best set of default parameters for the GramCluster implementation, we used two different training methods. In the first method, we randomly selected 10% of the sequences for training while the remaining 90% were used for testing. In the second method, we randomly divided the genera into two sets, one containing about 10% of the sequences and the other containing 90% of the sequences. The smaller set was again used as a training set to obtain the parameters for the algorithm. The default parameters ended up being the same as those found in the first training experiment. In particular, a grammar-based threshold of 0.13 was found to produce the best overall clustering metrics based on genera. We applied the same training methods to identify the best thresholds for GramCluster when clustering based on species. In this case, the best overall clustering metrics based on species occurred when the grammar-based threshold of 0.03 was applied.

For comparison, CD-HIT-EST (no version given, archive created on 4/27/2009) [59] and UCLUST version 3.0.617 [29] were also used to cluster the same 16S rDNA sequences and analyzed using the same quality metrics. All results were generated by compiling and executing the respective clustering programs on the same computer, specifically an Apple MacBook Pro with an Intel Core 2 Duo operating at 2.53 GHz with 4 Gb of system memory and a 3 Mb L2 cache. In the case of UCLUST, the binary was downloaded from the author's website. The experiments were conducted using various versions of FASTA files containing 74,709 16S rDNA sequences from 7,043 different species of 2,255 genera obtained from the Ribosomal Database Project (http://rdp.cme.msu.edu). For example, the second set of experiments involved a processed version of the FASTA file to simulate the application of clustering a large set of unknown fragments that typically result from high-throughput sequencing technologies, such as 454 pyrosequencing. In particular, every sequence was reduced to only the first 200 bases; and then the entire file was repeated 14 times for a total of 1,045,926 sequences from 2,255 genera.

## Experiments with Moderate-sized Data Set

The clustering algorithm was evaluated using the Folkes and Mallows Index, the Jaccard Coefficient, and Rand Statistic measures [40], along with in-cluster classification and sequence differentiation percentages, all defined above. The results for GramCluster, CD-HIT-EST, and UCLUST are presented in Figure 4.8.

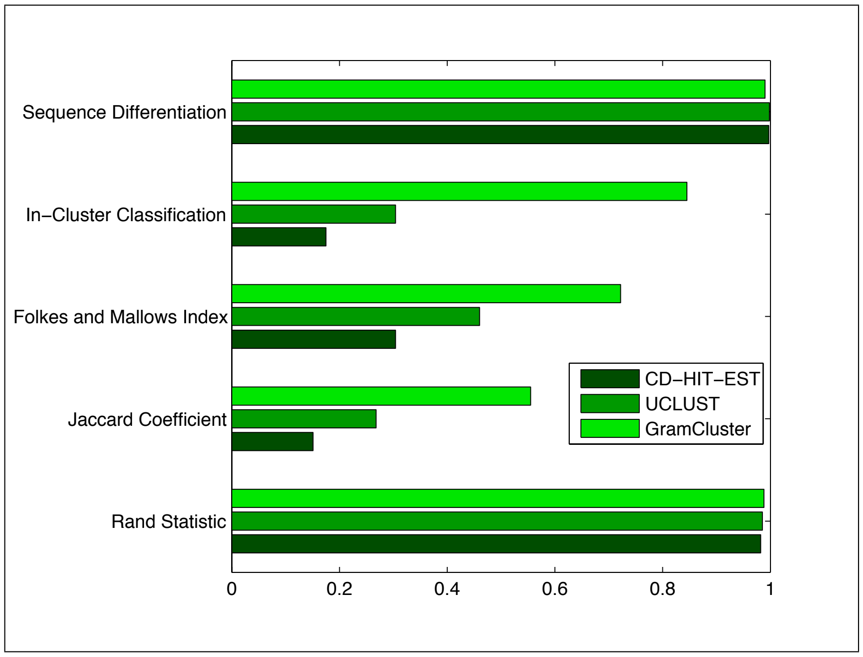


Figure 4.8: Cluster metrics for each algorithm operating on 74,709 16S rDNA sequences from 2,255 different genera.

Results indicate that CD-HIT-EST achieved 17.5% in-cluster classification and 99.7% sequence differentiation out of the 2,050 total clusters determined. That is, for sequences that were supposed to be in the same cluster, CD-HIT-EST placed them together 17.5% of the time; and for sequences that were not supposed to be in the same cluster, it correctly kept them in different clusters 99.7% of the time. Improved results for UCLUST show 30.4% and 99.8% in-cluster classification and sequence differentiation out of the 1,680 total clusters determined. By comparison, GramCluster achieved 84.5% in-cluster classification and 99.0% sequence differentiation out of the 2,447 total clusters identified. Clearly, GramCluster provides a significant improvement in clustering sequences correctly. This improvement can be further observed using common statistical measures for evaluating the performance of clustering algorithms [40] described previously. These measures are shown for GramCluster, CD-HIT-EST, and UCLUST operating on a set of 74,709 16S rDNA genes obtained from 2,255 different genera. The Jaccard Coefficient and Folkes and Mallows Index exceed those of CD-HIT-EST four-fold and over two-fold, respectively. The CPU execution time of GramCluster (1342 seconds) is on the same order as that of CD-HIT-EST (8277 seconds), which is considered ultra-fast [58]. The UCLUST CPU execution time (89 seconds) is much faster than GramCluster, however its quality metrics fall significantly short of those provided by GramCluster.

**Experiments with Large Data Set**

In order to simulate the application of clustering a large set of unknown fragments that typically result from 454 pyrosequencing, the previous FASTA file was modified such that every sequence was reduced to only the first 200 bases and then repeated 14 times for a total of 1,045,926 sequences from 2,255 genera.

Figure 4.9 contains data covering the same categories as in the previous experiment. CD-HIT-EST achieved only 3.3% in-cluster classification and 99.9% sequence differentiation of the 11,758 clusters found. So, for sequences that were supposed to be in the same cluster, CD-HIT-EST placed them together 3.3% of the time; and for sequences that were not supposed to be in the same cluster, it correctly kept them in different clusters 99.9% of the time. As in the previous experiment, results for UCLUST show 5.1% and 99.9% in-cluster classification and sequence differentiation out of the 10,686 total clusters determined. By comparison, GramCluster achieved

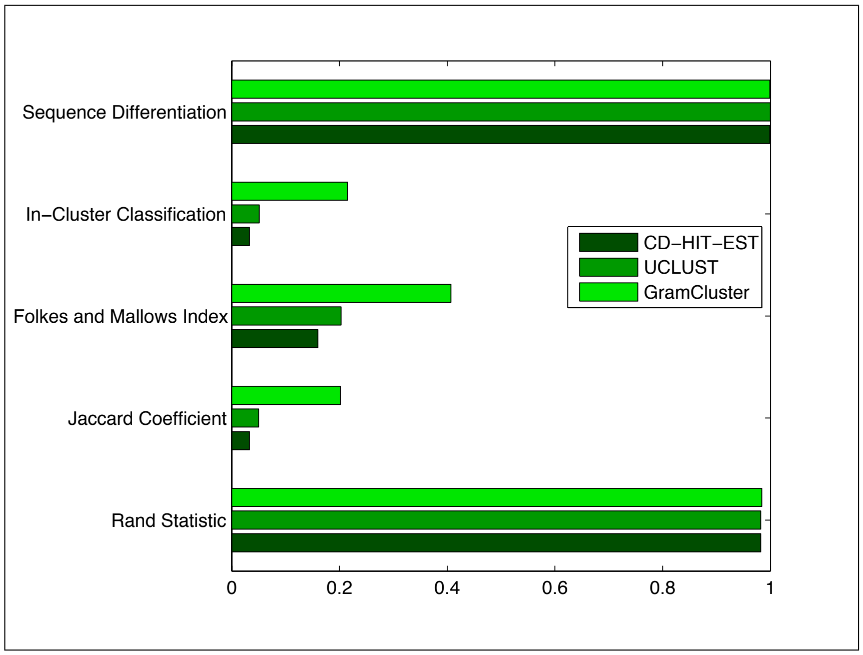


Figure 4.9: Cluster metrics for each algorithm operating on 1,045,926 16S rDNA sequences from 2,255 different genera.

21.5% and 99.9% out of the 5,917 clusters identified. GramCluster continues to show a significant improvement in terms of clustering sequences correctly with each other. This improvement can be seen further with the higher statistical measures, especially in the Jaccard Coefficient and Folkes and Mallows Index which are over six and two times those of CD-HIT-EST. Perhaps most interestingly, GramCluster identified a more accurate number of clusters at 5,917, even though the length of the sequences was significantly reduced, while both CD-HIT-EST and UCLUST reported identifying over 10,000 clusters.

We also tested BLASTClust [4] on 16S sequences. The program was too slow for classifying the original set of 74,709 sequences so we tested it using only 10% of the sequences. The results are shown in Figure 4.10. As can be seen, the results of CD-HIT-EST, UCLUST, and GramCluster all tend to match those of Figure 4.8. As

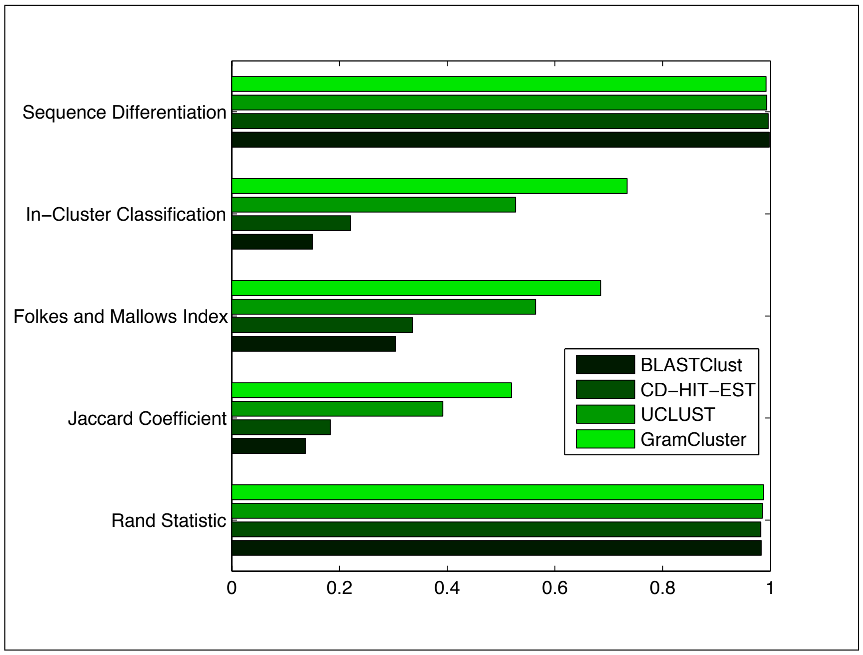


Figure 4.10: Cluster metrics for each algorithm operating on 7,470 16S rDNA sequences from 898 different genera.

can be seen in Figure 4.10, BLASTClust resulted in lower statistical metric scores in all categories, a high number of clusters compared to the number of genera. It is clear that the exclusion of BLASTClust from the other experiments due to its inability to operate on the size of the input data set has not diminished the results.

## Varying Command Line Options

Next, we consider the effect of varying the command line options primarily responsible for affecting the resulting data set partition. We ran two additional clustering experiments on the original set of sequences with GramCluster and UCLUST. The GramCluster experiments had both grammar-based distance thresholds altered from the default setting of 0.13 to 0.15 and 0.11. Similarly, the UCLUST experiments had the identity threshold altered from the default setting of 90% to 85% and 95%.

Figure 4.11 contains data covering the same categories as in the previous experiments. As the grammar-based distance threshold increased, sequences that were

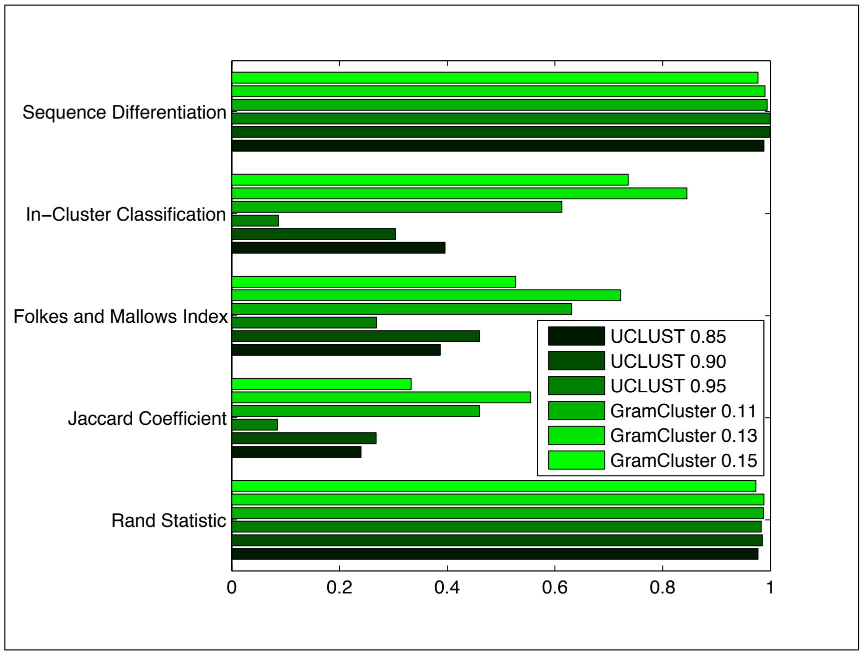


Figure 4.11: Cluster metrics for GramCluster and UCLUST operating on 74,709 16S rDNA sequences from 2,255 different genera. The grammar-based distance thresholds were both set to 0.11, 0.13, and 0.15 for GramCluster. The identity threshold was set to 85%, 90%, and 95% for UCLUST.

increasingly dissimilar were clustered together resulting in fewer clusters and poorer metrics. This same trend occurred with UCLUST as the identity threshold was relaxed by reducing it. Likewise, when the grammar-based distance threshold was reduced, sequences with an appropriately smaller distance clustered together. Similar behavior occurred when the UCLUST identity threshold was increased. In general, the default parameters for both programs seem to provide the best clustering of genus based on overall comparison of the metrics in Figure 4.11.

## Experiments Clustering on Species

The final experiment operated on the original set of sequences, but the partitioning was based on the sequence species instead of their genus.

Figure 4.12 contains data covering the same categories as in the previous experiments. In order to achieve the metrics in Figure 4.12 based on sequence species, it

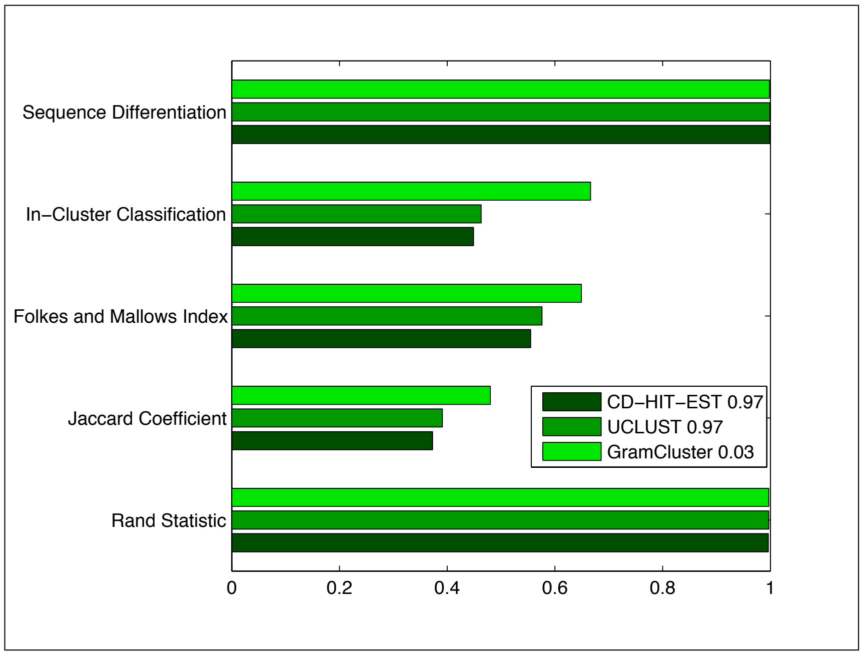


Figure 4.12: Cluster metrics for each algorithm operating on 29,566 16S rDNA sequences from 5,472 different species.

was necessary to modify the threshold of each clustering program. The UCLUST and CD-HIT-EST percent identity parameter was adjusted upward to require a higher sequence similarity before clustering sequences together. The best overall metric scores based on sequence species occurred at 97% identity for each algorithm. In contrast, the grammar-based distance thresholds in GramCluster had to be lowered to restrict the distance between sequences before classifying them together. The threshold of

0.03 caused the best overall metrics due to sequence species. The results presented in Figure 4.12 show a similar trend to those of the first experiment in Figure 4.8.

The results from all experiments show viable promise of GramCluster, especially when clustering numerous sequences such as in datasets produced by high-throughput sequencing applications.

## 4.3 Conclusions

This chapter introduced a computationally efficient clustering algorithm which can be used for clustering large datasets with high accuracy. The algorithm introduced was validated against a specific class of datasets containing 16S rDNA sequences but was designed to cluster any set of RNA, DNA, or protein sequences. The grammar-based distance work introduced in [79, 83] and previously used in [85] and Chapter 3 was modified to generate an estimation of the proper classification in which sequences are to be grouped. Results from clusters generated were presented in an attempt to study the overall quality of the resultant classifications as well as the computation time necessary to achieve the outputs. Accurate clustering of large numbers of biological sequences in an efficient amount of time is an important and challenging problem with a wide spectrum of applications. In this chapter, we adapted existing ideas in a novel way and introduced significant improvements.

We have introduced three applications of grammar-based models on two categories of computational biology problems. In all cases, we utilized an LZ-inferred regular grammar to generate numeric distance measures using information regarding the primary structure of sequences. In the next two chapters, we turn our attention to gaining additional information about biological sequences. In particular, we introduce two new methods for inferring CFGs capable of estimating secondary structure present in DNA and RNA sequences.

# Chapter 5

# Polynomial-Time CFG Inference of DNA/RNA Sequences

There is substantial interest in the primary and underlying secondary structure of biological sequences at some level of abstraction. The literature suggests the presence of a correlation between linguistic structure and that of biological function. However, often the grammar is assumed to be known *a priori*, which may not always be true. Chapters 3 and 4 used an LZ-grammar to compare the primary structure between biological sequences. The comparisons resulted in a grammar-based distance metric that was utilized in three different programs designed to solve various sequence analysis problems. This chapter introduces a novel framework for inferring secondary structure via context-free grammars (CFGs) and their associated parse trees in a polynomial-time algorithm. The grammar can be used to identify significant biological structural information present in the sequences without recourse to thermodynamic considerations, which is typically necessary.

## 5.1 Background

Identification of function and/or meaning of segments of biological sequences remains an ongoing and active area of research. This involves studying both primary and secondary structure; that is, the sequential ordering of symbols and the three-dimensional shapes that form due to attractions that occur among separated segments within the sequences. Two common methods for predicting RNA secondary structure include phylogenetic analysis of homologous RNA molecules [32] and thermodynamic modeling [44, 108], the latter of which has been made into web-based computer programs [43, 107] where dynamic programming is used to minimize the free energy in a given RNA sequence. A third method is found not by considering the physical molecules, but focusing on the information contained within the sequences. Examples such as [14, 88] review ways in which abstract grammars may be used to model and analyze secondary structures found in biological data, especially RNA sequences.

The literature suggests several uses of abstract grammars for studying biological data. For example, [91] and [35] describe correlations between linguistic structures and biological function. Grammars have been used to model nucleic acid structure [90, 89, 47], protein linguistics [1, 82], and gene regulation [18, 84, 56]. However, often the literature assumes the source grammar is already known, and it is usually for a specific set of biological data [84, 56]. This is an unrealistic assumption when analysis involves unknown data. Hence, the need for efficient grammar inference methods on biological data.

Previous inference work includes general algorithms presented in [71, 73], [87] and [68] for inferring CFGs for generic sequential inputs, which includes biological data. One drawback with all of these algorithms, is the inability to make use of domain knowledge, although [71] discusses the improvement available when domain knowledge is applied. This idea was exploited in [13] to operate specifically on DNA, and is the motivating idea of the linear-time algorithm developed in Chapter 6.

The next sections introduce an algorithm for inferring grammars of unknown biological sequences via a novel method based partially upon the classical CYK linguistic categorization approach. The CYK categorization works by testing for membership of a sequence to a language given a known grammar. Here we make use of the framework of the categorization approach to develop an inference engine for generating a, perhaps novel, grammar from a given sequence. This is as opposed to making use of a known grammar for classifying a given sequence. Results are presented showing the potential of the algorithm for detecting structure in a biological sequence. Possible further applications are detailed for future research.

### 5.1.1 Admissible Grammars

An *admissible grammar* [53] is defined as a CFG $G$ for which the following conditions are true:

- $G$ is deterministic. That is, for $A \in V(G)$ then $A$ is the head of exactly one element in $P(G)$;
- $\epsilon$ is not the body of any element in $P(G)$;
- $L(G) \neq \emptyset$;
- $G$ has no useless symbols. That is, for $Y \in (V(G) \cup T(G))$ where $Y \neq S$, there exists a derivation
$$S \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n \in L(G)$$
  such that $Y$ appears in $\alpha_i$ for some $1 \leq i \leq n$.

We will show that the grammar inferred by the algorithm presented here is always an admissible grammar. Thus, while the algorithm was developed with the intent of modeling secondary structure present in biological sequences, it may find application elsewhere based on the ability to infer an admissible grammar.

## 5.2 Inference Algorithm

The inference algorithm is meant to infer CFGs as part of the method for discovering secondary structure, especially for RNA. We first describe the CYK algorithm and then show how the basic framework of the CYK algorithm can be used to develop an inference algorithm for inferring a grammar from a given sequence. It should be noted that "CYK algorithm" implies the original, deterministic classification algorithm, not the stochastic version that is well known and often used in managing SCFGs such as detailed in [23]. Further, the goal of this work is to infer a minimal CFG in order to model the secondary structure of biological data, as opposed to starting with an *a priori* SCFG and modeling the stochastic parameters, which is the general goal of the nondeterministic version of the CYK algorithm.

It can be shown [46] that any nonempty context-free language (CFL) without $\epsilon$ has a grammar $G$ in which all productions are in either of the two forms $A \rightarrow a$ or $A \rightarrow BC$, where $A, B, C \in V(G)$ and $a \in T(G)$. Such a grammar is said to be in Chomsky Normal Form (CNF) and has binary parse trees, the fact of which was the initial motivation of this research.

### 5.2.1 CYK Algorithm

Membership of string $w$ in CFL $L(G)$ may be tested efficiently given a known CNF grammar $G$ and using a dynamic programming technique. Referring to Figure 5.1, the CYK algorithm (named after J. Cocke, D. Younger, and T. Kasami) begins at $row_1$ of the empty, lower-triangular matrix and works upward by loading the set at each matrix cell, $X_{i,j}$, with all production heads from $G$ that derive the corresponding terminal subsequence $a_i, ..., a_j$. Notice that the subscript indices $i$ and $j$ on each set refer to the starting and ending position of the terminal subsequence within $w$ derived by variables belonging to that matrix cell set. Further, the correlation between matrix cell and subsequence may be visualized in Figure 5.1 by starting at set $X_{i,j}$ and moving straight down the column to set $X_{i,i}$ whose members all derive the sequence element $a_i$, which is the first terminal of the string derived by all variables belonging to $X_{i,j}$. Next, begin with $X_{i,j}$ and move along the off-diagonal down and to-the-right to set $X_{j,j}$ whose members all derive the single terminal $a_j$, which is the end of the substring derived by variable members of $X_{i,j}$. Hence, if $A \in X_{i,j}$ then $A \overset{*}{\Rightarrow} a_i, ..., a_j$.

$$\begin{array}{r|ccccc}
\boldsymbol{row_5} & \boldsymbol{X_{1,5}} & & & & \\
\boldsymbol{row_4} & \boldsymbol{X_{1,4}} & \boldsymbol{X_{2,5}} & & & \\
\boldsymbol{row_3} & \boldsymbol{X_{1,3}} & \boldsymbol{X_{2,4}} & \boldsymbol{X_{3,5}} & & \\
\boldsymbol{row_2} & \boldsymbol{X_{1,2}} & \boldsymbol{X_{2,3}} & \boldsymbol{X_{3,4}} & \boldsymbol{X_{4,5}} & \\
\boldsymbol{row_1} & \boldsymbol{X_{1,1}} & \boldsymbol{X_{2,2}} & \boldsymbol{X_{3,3}} & \boldsymbol{X_{4,4}} & \boldsymbol{X_{5,5}} \\
\hline
\boldsymbol{sequence} & \boldsymbol{a_1} & \boldsymbol{a_2} & \boldsymbol{a_3} & \boldsymbol{a_4} & \boldsymbol{a_5}
\end{array}$$

Figure 5.1: Depicting the CYK algorithm for determining if a sequence $a_1, ..., a_5$ is a member of the language $L(G)$ generated via the known CNF grammar $G$.

To construct the matrix in Figure 5.1, the algorithm begins by initializing each

$$X_{i,i} = \big\{A \mid A \in V(G),\, (A \rightarrow a_i) \in P(G)\big\} \quad \text{for} \quad 1 \leq i \leq N$$

where $N$ is the length of $w$. The algorithm continues to completion by moving up one row each step and filling in

$$X_{i,j} = \big\{A \mid A \in V(G),\, (A \rightarrow BC) \in P(G),\, BC \overset{*}{\Rightarrow} a_i, ..., a_j\big\} \tag{5.1}$$

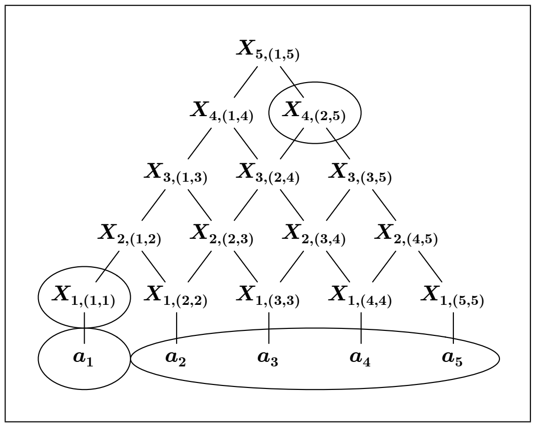
where indices $i$ and $j$ are governed as a pair by

$$(1, k) \leq (i, j) \leq (N - k + 1, N) \quad \text{for} \quad 2 \leq k \leq N$$

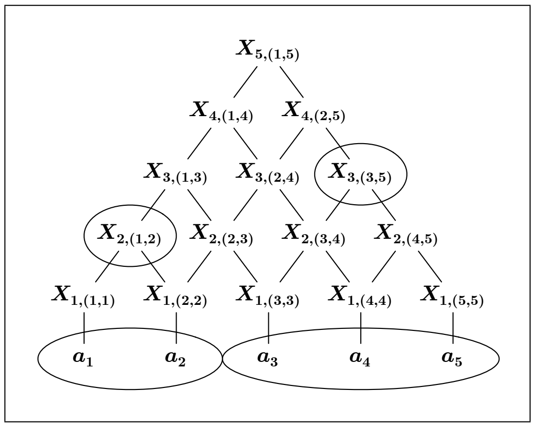
in which $k$ is the subscript of $row_k$ in Figure 5.1.

Searching over $V(G)$ for each $A \in X_{i,j}$ of (5.1) may be completed inductively by forming appropriate concatenations of set elements in the column below and along the off-diagonal from the $X_{i,j}$ under consideration. As an example, consider the set $X_{2,5}$. In particular, if $(A \rightarrow BC) \in P(G)$ where $B \in X_{2,2}$ and $C \in X_{3,5}$, then $A$ is added to $X_{2,5}$. Working up the column and down the off-diagonal, members of $X_{2,3}$ and $X_{2,4}$ are respectively concatenated with heads found in $X_{4,5}$ and $X_{5,5}$. Each time a concatenated pair forms the body of a production in $P(G)$, the associated head is added to the set $X_{2,5}$. When the algorithm completes, if $S \in X_{1,N}$ then $S \overset{*}{\Rightarrow} a_1, ..., a_N$, and so $w \in L(G)$.

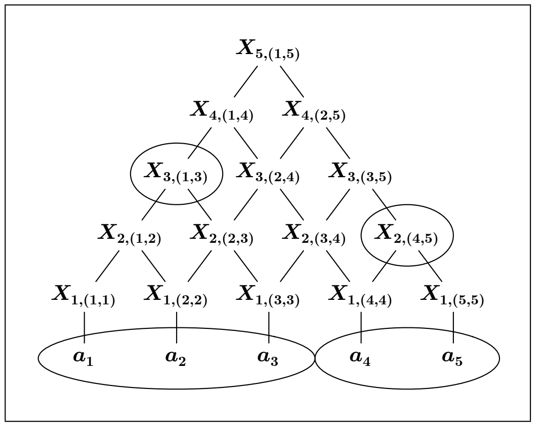
The matrix form depicted in Figure 5.1 is the classic representation for the CYK algorithm as presented in [46]. An alternative depiction is given in Figure 5.2, which shifts the matrix into a pseudo-binary tree form, with the exception that neighboring parent nodes share an inherited connection via a common child node. Additionally, the subscripts on each $X_{k,(i,j)}$ have changed, where $k$ indicates the "inverted depth" (ID) in the tree of the node, and $(i, j)$ are as before. As a matter of notation, ID 0 corresponds to the frontier (i.e., bottom, or leaf-nodes) of the tree. Figure 5.2 shows the four set concatenations used to fill in the root of the tree, $X_{5,(1,5)}$. In particular, from the node under construction $X_{5,(1,5)}$, the algorithm traverses up the left-most child nodes $X_{1,(1,1)}$, $X_{2,(1,2)}$, $X_{3,(1,3)}$ and $X_{4,(1,4)}$, and respectively concatenates with elements down the right-most child nodes $X_{4,(2,5)}$, $X_{3,(3,5)}$, $X_{2,(4,5)}$ and $X_{1,(5,5)}$. This depiction shows how all combinations of sub-string concatenations are checked within the rules of the given grammar in order to ultimately determine if $A \overset{*}{\Rightarrow} a_1, ..., a_N$ for some $A \in V(G)$. More importantly, this pseudo-binary tree form is used for the inference algorithm because a binary parse tree immediately follows from this graph by selectively removing nodes within.


$X_{5,(1,5)}$
$X_{4,(1,4)}$ $X_{4,(2,5)}$
$X_{3,(1,3)}$ $X_{3,(2,4)}$ $X_{3,(3,5)}$
$X_{2,(1,2)}$ $X_{2,(2,3)}$ $X_{2,(3,4)}$ $X_{2,(4,5)}$
$X_{1,(1,1)}$ $X_{1,(2,2)}$ $X_{1,(3,3)}$ $X_{1,(4,4)}$ $X_{1,(5,5)}$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$

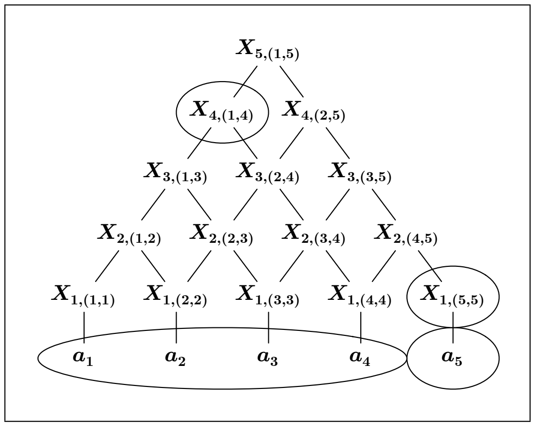
$X_{1,(1,1)} \cdot X_{4,(2,5)} \in X_{5,(1,5)}$


$X_{5,(1,5)}$
$X_{4,(1,4)}$ $X_{4,(2,5)}$
$X_{3,(1,3)}$ $X_{3,(2,4)}$ $X_{3,(3,5)}$
$X_{2,(1,2)}$ $X_{2,(2,3)}$ $X_{2,(3,4)}$ $X_{2,(4,5)}$
$X_{1,(1,1)}$ $X_{1,(2,2)}$ $X_{1,(3,3)}$ $X_{1,(4,4)}$ $X_{1,(5,5)}$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$

$X_{2,(1,2)} \cdot X_{3,(3,5)} \in X_{5,(1,5)}$


$X_{5,(1,5)}$
$X_{4,(1,4)}$ $X_{4,(2,5)}$
$X_{3,(1,3)}$ $X_{3,(2,4)}$ $X_{3,(3,5)}$
$X_{2,(1,2)}$ $X_{2,(2,3)}$ $X_{2,(3,4)}$ $X_{2,(4,5)}$
$X_{1,(1,1)}$ $X_{1,(2,2)}$ $X_{1,(3,3)}$ $X_{1,(4,4)}$ $X_{1,(5,5)}$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$

$X_{3,(1,3)} \cdot X_{2,(4,5)} \in X_{5,(1,5)}$


$X_{5,(1,5)}$
$X_{4,(1,4)}$ $X_{4,(2,5)}$
$X_{3,(1,3)}$ $X_{3,(2,4)}$ $X_{3,(3,5)}$
$X_{2,(1,2)}$ $X_{2,(2,3)}$ $X_{2,(3,4)}$ $X_{2,(4,5)}$
$X_{1,(1,1)}$ $X_{1,(2,2)}$ $X_{1,(3,3)}$ $X_{1,(4,4)}$ $X_{1,(5,5)}$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$

$X_{4,(1,4)} \cdot X_{1,(5,5)} \in X_{5,(1,5)}$

Figure 5.2: Depicting the alternative CYK graph for filling in the root node, $X_{5,(1,5)}$.

### 5.2.2 Framework

The CYK algorithm has previously been applied to grammar inference, such as in [68] and [87] in which a tabular and a heuristic algorithm are presented, respectively. In particular, the "Synapse" algorithm [68] uses the CYK algorithm to incrementally verify and subsequently modify a given grammar in order to allow in-set sequences to be generated by the grammar and disallow the derivation of out-of-set sequences. The "TBL" algorithm [87] constructs a lower-triangular matrix (see Figure 5.1), where each cell contains every combination of possible productions for each sequence of an in-set. The algorithm searches for a good grammar by using a genetic algorithm to repeatedly partition the set of all productions until a certain fitness metric is satisfied.

By contrast, the goal of the algorithm presented here is to infer a grammar from an unknown sequence with biological secondary structure in mind. The method does not use the CYK algorithm directly, but builds a CFG from the bottom up based upon scoring node entries at each depth of the pseudo-binary tree such that only a head with the maximum score is added to the node. Then, the graph is pruned from top down, removing unreachable rules from the grammar and resulting in a binary parse tree representing linguistic structure in the form of repeated $k$-grams.

Referring to Figure 5.2, every cell at ID 1 has only one way to derive their respective 1-gram, in the form of a "type-1" rule $A \rightarrow a$. Similarly, all cells at ID 2 have one production possible for their associated 2-gram derivation in the form of a "type-2" rule $A \rightarrow BC$. Because there are no choices, there is no apparent need for scoring metrics. However, when the algorithm moves to ID 3 each cell has exactly two choices of productions that will derive the proper 3-gram, $A_{3a} \rightarrow B_1 C_2$ or $A_{3b} \rightarrow B_2 C_1$ where the subscripts in these two productions are used to indicate the ID of the head. If the algorithm did not pick a single rule for each cell, the number of choices would continue to increase exponentially, as in

$$\left|X_{k,(-,-)}\right| = \sum_{i=1}^{k-1} \left|X_{i,(-,-)}\right| \left|X_{k-i,(-,-)}\right|.$$

For example, $\left|X_{k,(-,-)}\right| = \{1, 1, 2, 5, 14, 42, 132, ...\}$ for $k = \{1, 2, 3, 4, 5, 6, 7, ...\}$. Thus, while considering every possible combination would allow for the discovery of a best linguistic model, it would be costly in performing the forward pass of the dynamic programming, and difficult in searching over the final set of possible productions at $X_{N,(1,N)}$. The inference algorithm handles the computational complexity of this problem by scoring every rule at each ID $k$, and selecting only the highest scoring one to represent each $k$-gram identified by the respective cell at depth $k$. Thus, $\left|X_{N,(1,N)}\right| = N-1$ is the largest set to be searched for the highest scoring production. The remaining key to the algorithm is the metric used in scoring the production rules.

We will see how the inference algorithm works and introduce the scoring metrics by using the toy example, $w = acagt$. The first step corresponds to creating "type-1" productions for the nodes at ID 1. New variables are created as the heads of production rules that derive a single terminal symbol. For the simplest inference, duplicate productions are assumed never to exist. Thus, for a sequence of length $N$, and a terminal alphabet $\Sigma$, there shall be no more than $\min(N, |\Sigma|)$ grammar productions created at ID 1. For the given example, the initial grammar is $V(G) = \{A_0, A_1\}$, $T(G) = \{a, c, g, t\}$, $P(G) = \{A_0 \rightarrow a, A_1 \rightarrow c\}$, and the sets in the graph depicted in Figure 5.3 are $X_{1,(1,1)} = \{A_0\}$, $X_{1,(2,2)} = \{A_1\}$, $X_{1,(3,3)} = \{A_0\}$, $X_{1,(4,4)} = \{A'_1\}$, and $X_{1,(5,5)} = \{A'_0\}$.

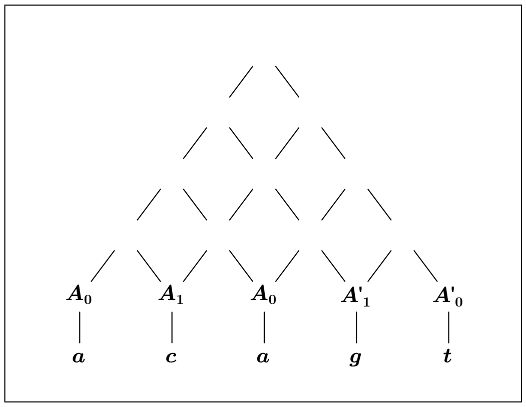


Figure 5.3: Depiction of the grammar inference example after the first ID.

Because this algorithm is intended for analyzing DNA and RNA sequences, domain knowledge is used to improve the grammar inference and the subsequent structure identification. In particular, whenever the input sequence is known to be either

DNA or RNA, the Chargaff base pairing rules are used within the production bodies and the unary operator $'$ is used to indicate a reverse complementary repeat. This corresponds to $a' = t/u$ and $c' = g$, so that at ID 1 $P(G) = \{A_0 \to a, A_1 \to c, A_1' \to g, A_0' \to t/u\}$. The algorithm is designed such that if base pairing is unwanted or unwarranted, for example in protein-sequence analysis, it may be disabled producing tandem repeat structure identification only.

After the nodes at ID 1 have been identified, the input sequence is scanned for every pair of non-overlapping complementary $m$-grams and non-overlapping tandem repeat $m$-grams, where $m$ is any integer greater than some user-defined input. The scanning is performed using two $N \times N$ matrices, shown for the toy example in Figure 5.4. The algorithm uses each matrix to identify common structural elements by locating diagonal runs. The sequence is compared to itself and its reverse complement using the tandem repeat matrix and the reverse complement matrix, respectively. The score of cell $(i, j)$ in both matrices is given by

$$M(i,j) = \begin{cases} M(i-1, j-1) + 1 & x_i = x_j \\ 0 & x_i \neq x_j, \end{cases}$$

where $x_k$ is the $k$th element of the sequence along the left or the top. When complete, each matrix contains nonzero entries that imply common subsequences. Runs in the Figure 5.4(a) matrix indicate common tandem repeats, while runs in the Figure 5.4(b) matrix imply common reverse complements.

Referring to the example, the elements of ID 1 shown in Figure 5.3, $A_0A_1A_0A_1'A_0'$, are listed along the left side of each matrix in Figure 5.4. The same sequence is repeated across the top of the left matrix while the reverse complement, $A_0A_1A_0'A_1'A_0'$, is placed on the top of the right matrix. After the matrices have been filled with their scores, both are scanned for nonzero diagonal runs. The coordinate pair of each structural run is added to a set containing all identified runs from the matrices. Note

$$\begin{array}{c|c|c|c|c|c|}
 & A_0 & A_1 & A_0 & A_1' & A_0' \\ \hline
A_0 & 1 & 0 & 1 & 0 & 0 \\ \hline
A_1 & 0 & 2 & 0 & 0 & 0 \\ \hline
A_0 & 1 & 0 & 3 & 0 & 0 \\ \hline
A_1' & 0 & 0 & 0 & 4 & 0 \\ \hline
A_0' & 0 & 0 & 0 & 0 & 5 \\ \hline
\end{array}$$

(a) Tandem Repeats

$$\begin{array}{c|c|c|c|c|c|}
 & \mathbf{A_0} & \mathbf{A_1} & A_0' & \mathbf{A_1'} & \mathbf{A_0'} \\ \hline
\mathbf{A_0} & \mathbf{1} & 0 & 0 & 0 & 0 \\ \hline
\mathbf{A_1} & 0 & \mathbf{2} & 0 & 0 & 0 \\ \hline
A_0 & 1 & 0 & 0 & 0 & 0 \\ \hline
\mathbf{A_1'} & 0 & 0 & 0 & \mathbf{1} & 0 \\ \hline
\mathbf{A_0'} & 0 & 0 & 1 & 0 & \mathbf{2} \\ \hline
\end{array}$$

(b) Reverse Complements

Figure 5.4: The structural component matrices for the toy example. Each element of the sequence along the left side is compared to each element of the sequence across the top. In (a) the sequence along the top is the same as the sequence along the left side, in (b) the sequence along the top is the reverse complement of the sequence on the left side. Note that (a) is symmetric and the diagonal has a trivial run of length $N$. Boldface is used in (b) to highlight the structural runs.

that the tandem repeat matrix in Figure 5.4(a) is symmetric and the diagonal has a trivial nonzero run of length $N$. The only real structural runs are highlighted by boldface in Figure 5.4(b), and they are actually equivalent as the start of one sequence pairs with the end of the other and *vice versa*. When complete the set contains $\Big\{\big((1,2),(4,5)\big)\Big\}$ indicating the subsequence $x_1, x_2$ pairs with subsequence $x_4, x_5$.

Longer sequences will generate sets of pairs that overlap and intersect each other. Thus, not all identified structural pieces will be included in the grammar inference. While this is not the case of the toy example, the resulting set of possible pairs is tested for compatibility (i.e., pseudoknot structures, and overlapping runs of bases are not allowed). The algorithm greedily identifies the largest structural coverage of the sequence in terms of number of bases. An efficient algorithm for solving this classic problem is detailed in [8].

Once the various structural pieces have been identified, each base position within a subsequence $m$-gram is marked with identification that it should be in a linguistic structural element. In the case of the example the first two and last two positions are marked, $\underline{A_0A_1}A_0\underline{A'_1A'_0}$, for subsequent use by the scoring criteria.

Similar to the CYK algorithm, the inference algorithm starts at the bottom of the graph in Figure 5.3 and operates from left-to-right, bottom-to-top, loading each node of the current ID prior to moving up a depth, eventually reaching the root of the pseudo-tree. At lower IDs, the algorithm's objective is to discover as many of the largest, non-overlapping, repeated $k$-grams as possible. All nodes in Figure 5.2 at ID $k$ necessarily derive a $k$-gram. Thus, any nodes with the same value at ID $k$ spatially appearing at least $k$ nodes apart represent the appearance of a repeated $k$-gram in the terminal sequence. Such nodes will be referred to as (linguistically) structural elements.

Eventually, the algorithm will cease to discover structural elements. This is evident from Figure 5.5, in which all nodes overlap each other at ID $k$ for $k > \lfloor N/2 \rfloor$. When the algorithm is unable to identify additional structural elements, it changes objectives to creating a minimal pathway between structural elements. The result of this objective is a reduced number of intermediate production rules, thereby creating a smaller grammar model.

After ID 1, all subsequent steps correspond to creating the best "type-2" productions for the nodes within ID 2 up through ID $N$. In particular, for any cell at ID $k$, $k-1$ rules are generated, compared and one is selected based on the scoring criteria. Each of the possible rules occur by concatenating vertices running up the left-most descendants and down the right-most descendants of the node under consideration, as was done in Figure 5.2 for node $X_{5,(1,5)}$. The inference algorithm searches over the $k-1$ appropriate concatenations by comparing two rules at a time, until the single best rule is discovered.

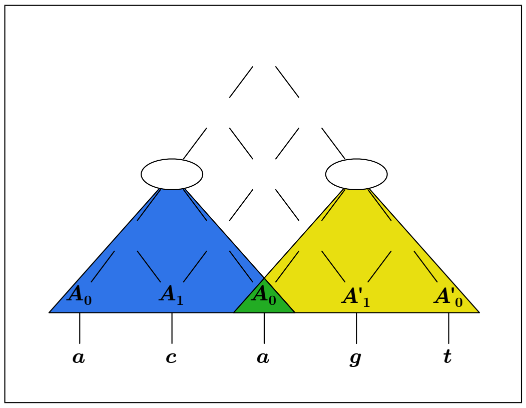


Figure 5.5: The inference algorithm is unable to identify linguistic structural pieces above ID $k$ for $k > \lfloor N/2 \rfloor$ since the $k$-grams overlap each other.

Let the two rules under consideration be $A \rightarrow CD$ and $B \rightarrow EF$, and define the path distance to the nearest structural element as $d_i$ for rule $i$. This value is saved per grammar rule, and is set to 0 when a rule is determined to be a structural element. Since each production at ID 1 is considered to be the smallest structural component available, the path distances for the current example grammar are $d_{A_0} = 0$ and $d_{A_1} = 0$.

Next, define a structural component vector as an $\lfloor N/2 \rfloor$-dimensional vector in which the value at coordinate $k$ indicates the number of $k$-gram structural components included in the derivation of the associated production rule. Let $\mathbf{v}_{t,i}, \mathbf{v}_{r,i}$, and $\mathbf{v}_{s,i}$ be structural component vectors for grammar rule $i$, where the subscripts $t, r$, and $s$ correspond to linguistic tandem repeat components, reverse complement components, and self-complement components, respectively. The linguistic tandem repeat structural component vectors for the toy example grammar are $\mathbf{v}_{t,A_0} = [1, 0]$ and $\mathbf{v}_{t,A_1} = [1, 0]$ since all type-1 productions are stored without complements. On the other hand, $\mathbf{v}_{r,A_0} = [0, 0]$, $\mathbf{v}_{r,A_1} = [0, 0]$, $\mathbf{v}_{s,A_0} = [0, 0]$, and $\mathbf{v}_{s,A_1} = [0, 0]$ for the same reason.

Also, let $\mathbf{v}_{T,n}$ and $\mathbf{v}_{R,n}$ be structural component vectors for the grammar rule at position $n$, where the subscripts $T$ and $R$ correspond to compatible pair tandem repeat components and complementary components, respectively. While the previous linguistic structural component vectors were defined per grammar rule, these vectors are defined per graph node. For the example, all tandem repeat component vectors are $[0,0]$, while the reverse complement component vectors are $\mathbf{v}_{R,1} = [1,0]$, $\mathbf{v}_{R,2} = [1,0]$, $\mathbf{v}_{R,3} = [0,0]$, $\mathbf{v}_{R,4} = [1,0]$, and $\mathbf{v}_{R,5} = [1,0]$.

Continuing the example, the inference algorithm can fill out ID 2 without any decision making since only one rule is possible at each graph node at that inverted depth. The current graph shown in Figure 5.6 depicts the additional type-2 rules, $A_2 \rightarrow A_0A_1$, $A_3 \rightarrow A_1A_0$, and $A_4 \rightarrow A_0A_1'$. After each new rule is added to the

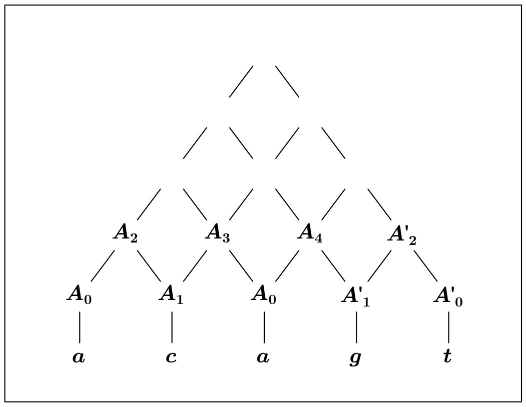


Figure 5.6: Depiction of the grammar inference example after the first two IDs.

grammar, the associated metrics are calculated based on the metrics of its constituent rules. Consider the rule $A \rightarrow BC$ and define the function

$$\theta(i, \mathbf{v}_m, \mathbf{v}_n) = \begin{cases} (\mathbf{v}_n, \mathbf{v}_m) & \text{if rule } i = j' \text{ for existing rule } j, \\ (\mathbf{v}_m, \mathbf{v}_n) & \text{otherwise} \end{cases}$$

which is used to swap the tandem repeat structural component vector with the reverse complement structural component vector when rule $i$ is the reverse complement of some previously defined rule. Then the tandem repeat and reverse complement structural component vectors are determined by

$$(\mathbf{v}_{t,A}, \mathbf{v}_{r,A}) = \theta(B, \mathbf{v}_{t,B}, \mathbf{v}_{r,B}) + \theta(C, \mathbf{v}_{t,C}, \mathbf{v}_{r,C}),$$

and the self-complement structural component vector is

$$\mathbf{v}_{s,A} = \mathbf{v}_{s,B} + \mathbf{v}_{s,C}.$$

Finally, the new path-distance is given by

$$d_A = d_B + d_C + 1.$$

Define $\gamma(i)$ as the non-overlapping head count for rule $i$. Then if $\gamma(A) > 1$ indicating that $A$ is a structural component, then $d_A = 0$ and either $\mathbf{v}_{s,A}(k) = 1$ or $\mathbf{v}_{t,A}(k) = 1$, depending on if rule $A$ is a reverse complement of itself or not.

Returning to the example, consider the left-most node of ID 2 which contains the rule $A_2 \to A_0A_1$ with initial metrics $d_{A_2} = 0 + 0 + 1 = 1$, $\mathbf{v}_{t,A_2} = [2, 0]$, $\mathbf{v}_{r,A_2} = [0, 0]$ and $\mathbf{v}_{s,A_2} = [0, 0]$. However, the fourth vertex of ID 2 also contains $A_2'$ and its 2-gram does not overlap the 2-gram of the first node. Hence, $\gamma(A_2) = 2$, leading to a revision of the final metrics of $d_{A_2} = 0$, $\mathbf{v}_{t,A_2} = [2, 1]$, $\mathbf{v}_{r,A_2} = [0, 0]$ and $\mathbf{v}_{s,A_2} = [0, 0]$. Notice the distance has been set to zero, and because $A_2 = A_0A_1 \neq A_1'A_0' = A_2'$, the tandem repeat structural component vector is modified in the second coordinate, corresponding to ID 2. Similarly, $A_3$ and $A_4$ are created in the second and third vertices, but are not modified since they are not considered structural components. Therefore, $d_{A_3} = 1$, $\mathbf{v}_{t,A_3} = [2, 0]$, $\mathbf{v}_{r,A_3} = [0, 0]$, $\mathbf{v}_{s,A_3} = [0, 0]$, $d_{A_4} = 1$, $\mathbf{v}_{t,A_4} = [2, 0]$, $\mathbf{v}_{r,A_4} = [0, 0]$, and $\mathbf{v}_{s,A_4} = [0, 0]$. The new variables $\{A_2, A_3, A_4\}$ and productions $\{A_2 \to A_0A_1, A_3 \to A_1A_0, A_4 \to A_0A_1'\}$ are appended to the grammar and the sets at ID 2 are $X_{2,(1,2)} = \{A_2\}$, $X_{2,(2,3)} = \{A_3\}$, $X_{2,(3,4)} = \{A_4\}$, and $X_{2,(4,5)} = \{A_2'\}$.

Finally, the tandem repeat and reverse complement vectors are updated for this ID by adding the corresponding vectors of the body rules giving $\mathbf{v}_{R,1} = [2, 0]$, $\mathbf{v}_{R,2} = [1, 0]$, $\mathbf{v}_{R,3} = [1, 0]$, and $\mathbf{v}_{R,4} = [2, 0]$. Additionally, any rules that derive a substring completely contained in a tandem repeat or reverse complement run that is specified in the set of compatible pairs have their depth coordinate supplemented by 1. Thus, $\mathbf{v}_{R,1} = [2, 1]$ and $\mathbf{v}_{R,4} = [2, 1]$ since it was determined in Figure 5.4 that the ends pair together.

The inference algorithm continues up the graph using the following prioritized criteria to always select a single production rule to represent the corresponding $k$-gram. For each choice, let the two rules under consideration be $A \rightarrow CD$ and $B \rightarrow EF$. Combinations of the structural component vectors and the structural distance are used in the following comparisons to determine whether $A$ or $B$ is better.

1. If either (but not both) $d_C + d_D = 0$ or $d_E + d_F = 0$, then select either $A$ or $B$, respectively. If neither are 0, jump to condition 2, if both are 0 move to sub-condition a.

   These conditions occur when both body members of a new production are structural elements. Naturally, to achieve the shortest path-distance, new rules constructed completely from structural elements are selected before any rules that are not directly using structural pieces.

   (a) Recall $\gamma(i)$ is defined as the non-overlapping head count for rule $i$ and let $\phi(i)$ be ID $i$. Then, if

   $$\gamma(C)\phi(C) + \gamma(D)\phi(D) > \gamma(E)\phi(E) + \gamma(F)\phi(F),$$

   pick $A$, or if

   $$\gamma(C)\phi(C) + \gamma(D)\phi(D) < \gamma(E)\phi(E) + \gamma(F)\phi(F),$$

pick $B$. If neither are true, check the next condition.

When both $A$ and $B$ are constructed using two structural elements, the algorithm requires more criteria to determine which is better. This metric gives precedence to recurring bodies at higher IDs, thereby keeping more-significant structural pieces and resulting in a smaller grammar.

For the first cell of ID 3 in the toy example, the two possible productions are $A_A \rightarrow A_0 A_3$ and $A_B \rightarrow A_2 A_0$. First we check the structural distances, $d_{A_0} + d_{A_3} = 1$ and $d_{A_2} + d_{A_0} = 0$. This criteria selects rule $A_B$ for $X_{3,(1,3)}$. Now consider the second cell of ID 3 which has the two possible productions $A_A \rightarrow A_1 A_4$ and $A_B \rightarrow A_3 A_1'$. In this case, the structural distances are not able to score one rule above the other since both are 1; thus the algorithm moves to the second criteria.

2. Let $>_{lex}$ and $<_{lex}$ be the lexicographic comparisons of vectors such that the highest dimension is compared first, and only subsequent dimensions are checked if all previous are equal. If

$$\mathbf{v}_{R,n_A} >_{lex} \mathbf{v}_{R,n_B},$$

pick $A$. If

$$\mathbf{v}_{R,n_A} <_{lex} \mathbf{v}_{R,n_B},$$

pick $B$. Recall the reverse complement vectors are position dependent; here $n_i$ represents the cell position from the left containing rule $i$ within the current inverted depth. If neither are true, check the next condition.

These comparisons lead to the selection of rules containing the most pieces from the greedy compatible pairs search over the set of complementary pairs by selecting the production with the maximal lexicographic vector, in which the element with the highest dimension is the most-significant position in the comparison.

The reverse complement vectors of the example are $\mathbf{v}_{R,2_A} = [2,0]$ and $\mathbf{v}_{R,2_B} = [2,0]$, so the algorithm moves to the third criteria.

3. If

$$\mathbf{v}_{T,n_A} >_{lex} \mathbf{v}_{T,n_B},$$

pick $A$. If

$$\mathbf{v}_{T,n_A} <_{lex} \mathbf{v}_{T,n_B},$$

pick $B$. If neither are true, check the next condition.

These comparisons lead to the selection of rules containing the most pieces from the greedy compatible pairs search over the set of tandem repeat pairs by selecting the production with the maximal lexicographic vector, in which the element with the highest dimension is the most-significant position in the comparison.

All of the tandem repeat vectors for the simple example are zero, so the algorithm checks the fourth criteria.

4. If

$$\mathbf{v}_{t,A} + \mathbf{v}_{r,A} + \mathbf{v}_{s,A} >_{lex} \mathbf{v}_{t,B} + \mathbf{v}_{r,B} + \mathbf{v}_{s,B},$$

pick $A$. If

$$\mathbf{v}_{t,A} + \mathbf{v}_{r,A} + \mathbf{v}_{s,A} <_{lex} \mathbf{v}_{t,B} + \mathbf{v}_{r,B} + \mathbf{v}_{s,B},$$

pick $B$. If neither are true, check the next condition.

These comparisons lead to the selection of rules containing the most recurring largest structural components by selecting the production with the maximal lexicographic vector, in which the element with the highest dimension is the most-significant position in the comparison.

The two rules currently under consideration have the vectors $\mathbf{v}_{t,A_A} = [3,0]$, $\mathbf{v}_{r,A_A} = [0,0]$, $\mathbf{v}_{s,A_A} = [0,0]$, $\mathbf{v}_{t,A_B} = [3,0]$, $\mathbf{v}_{r,A_B} = [0,0]$, and $\mathbf{v}_{s,A_B} = [0,0]$. Since they are equivalent, the algorithm moves to criteria five.

5. If

$$\left|\left|\mathbf{v}_{t,A} - \mathbf{v}_{r,A}\right| - \mathbf{v}_{s,A}\right| >_{lex} \left|\left|\mathbf{v}_{t,B} - \mathbf{v}_{r,B}\right| - \mathbf{v}_{s,B}\right|,$$

pick $B$, otherwise if

$$\left|\left|\mathbf{v}_{t,A} - \mathbf{v}_{r,A}\right| - \mathbf{v}_{s,A}\right| <_{lex} \left|\left|\mathbf{v}_{t,B} - \mathbf{v}_{r,B}\right| - \mathbf{v}_{s,B}\right|,$$

pick $A$. If neither are true, check the next condition.

Here the algorithm incorporates domain knowledge when an alphabet is used such that $\Sigma$ is closed under $'$. In the case of DNA or RNA sequences the structural component vectors $\mathbf{v}_{r,i}$ and $\mathbf{v}_{s,i}$ will contain entries that identify $k$-grams that are either reverse complements of other $k$-grams or of themselves, respectively. Especially when identifying secondary structure in RNA, it is important to locate the largest $k$-grams that have reverse complements, thereby identifying the longest runs of base pairing present in the sequence. Hence, the algorithm chooses rules that contain the most pairings between rules and inverted complements by lexicographically comparing the absolute value of the difference between the tandem repeat component vector and the two complementing vectors. The rule with the smallest resultant vector is selected since it implies that more of the largest structural components have matched up with reverse complements.

As in the previous criteria, the component vectors are the same for the rule being considered, so the algorithm will consider the sixth criteria next.

6. At this point, the algorithm selects the rule that has the shortest structural path distance between both body nodes. As mentioned previously, this will result in fewer intermediate variables added to the grammar, thereby resulting in a smaller model. The decision is if

   $$d_C + d_D < d_E + d_F,$$

   then pick $A$, or if

   $$d_C + d_D > d_E + d_F,$$

   then pick $B$. If neither are true, check the next condition.

   The structural distances for the two rules being compared are both 1, so the algorithm moves to criteria seven.

7. Next, define the variance between two rules as the difference between their ID. The algorithm is encouraged to select the minimum-variant rule leading to an improved parse tree shape, which has been shown to be beneficial in some coding schemes (e.g. V.42 bis). So, if

   $$\left|\phi(C) - \phi(D)\right| < \left|\phi(E) - \phi(F)\right|,$$

   pick $A$, otherwise if

   $$\left|\phi(C) - \phi(D)\right| > \left|\phi(E) - \phi(F)\right|,$$

   pick $B$.

   The variance is the same in both rules being considered in the example. The algorithm has to use the final criteria to pick a production rule.

8. Finally, the algorithm has exhausted all criteria, and so the two rules are equivalent. Thus, random selection is used to determine the final selection.

The remainder of the forward pass for the example follows the same procedure through the criteria list. After each node of the pseudo-tree is set, the final step involves pruning all unused nodes and grammar rules. This is done by selecting the rule at the root of the tree as the grammar start symbol $S$ and generating a derivation. All unused variables and internal tree nodes are deleted, resulting in the inferred grammar and associated binary parse tree.

To complete the example, the resulting grammar derivation is

$$A_6 \Rightarrow A_2A_5 \Rightarrow A_0A_1A_5 \Rightarrow A_0A_1A_0A_2' \Rightarrow A_0A_1A_0A_1'A_0' \Rightarrow acagt$$

and the binary parse tree is depicted in Figure 5.7. Notice the reverse complement digram $ac$, represented by the variable $A_2$, appears in the parse tree. Also, note that

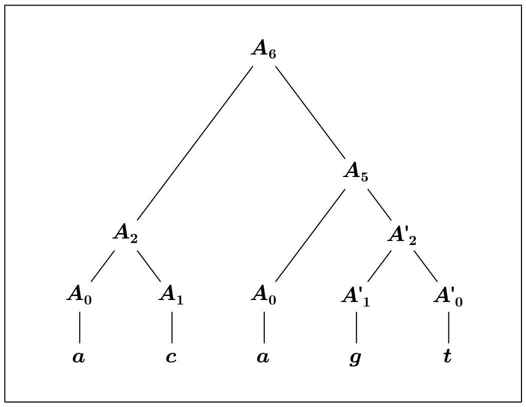


Figure 5.7: Depiction of the inferred structure of example $w$.

$\mathbf{v}_{t,A_6} = [3, 1]$, $\mathbf{v}_{r,A_6} = [2, 1]$ and $\mathbf{v}_{s,A_6} = [0, 0]$, indicating that there is one pair of complementary digrams and five singletons within this derivation. Using structural component vectors in this way, the algorithm is able to estimate a good model of the given sequence paying special attention to the secondary structure of the input as the algorithm moves up the graph.

### 5.2.3 Symbolic Sequence

While the grammar inference is the core of this chapter, the resultant grammar is not immediately useful without further processing. As a result, the program implementing the inference algorithm outputs various formats including the binary parse tree of the grammar in encapsulated postscript (see for example Figure 5.11), and a symbolic sequence detailing the secondary structure in FASTA format as well as HTML (see for example Figure 5.12).

The symbolic sequence is used to indicate where structural pieces have been identified within the input sequence. The symbols used are

- '.' = not contained in a structural element;
- '(' = contained in the first half of a complementary repeat;
- ')' = contained in the second half of a complementary repeat;
- '*' = contained in a tandem repeat;
- '[' = contained in a tandem repeat and the first half of a complementary repeat;
- ']' = contained in a tandem repeat and the second half of a complementary repeat.

The present implementation allows the user to switch between allowing the square brackets and just parenthesis to provide control over the importance of tandem repeats compared to complementary repeats.

### 5.2.4 Example Simulations

In this section, example simulations are used to study different aspects of the inference framework. All results were generated by compiling an executing the initial implementation, ICYK–short for "Inferring via CYK," on an Apple MacBook Pro with an

Intel Core 2 Duo operating at 2.53 GHz with 4 Gb of system memory and a 3 Mb L2 cache. ICYK assumes one of three possible terminal sets depending on if the input is a sequence of amino acids, or DNA/RNA bases. Including necessary characters to account for unknowns during sequencing, the former contains 23 different terminal elements while the latter two contain the five well known characters $\{a, c, g, t/u, x\}$, where $x$ is for unknown.

**Experiments Detailing Grammar**

The first set of experiments demonstrate several aspects of the grammar inference framework by considering the inferred grammar in addition to the associated binary parse trees. The experiments presented in this section will be revisited again in Chapter 6 for comparison, especially those involving reverse complement fragments.

Beginning with two real molecules, consider the RNA input sequence *gagc...gagc* taken from the Jena Library of Biological Macromolecules. This sequence is Chain W of the TRP RNA-binding attenuation protein (TRAP) bound to an RNA molecule containing 11 *gagc* repeats. The resulting grammar is given by $V(G) = \big\{A_0, ..., A_{10}\big\}$, $T(G) = \big\{a, c, g\big\}$, $P(G) = \big\{A_0 \rightarrow a, A_1 \rightarrow c, A_2 \rightarrow A_0A_1', A_3 \rightarrow A_2A_1, A_4 \rightarrow A_1'A_3, A_5 \rightarrow A_4A_4, A_6 \rightarrow A_5A_4, A_7 \rightarrow A_4A_6, A_8 \rightarrow A_7A_4, A_9 \rightarrow A_4A_8, A_{10} \rightarrow A_8A_9\big\}$, where $S = A_{10}$, and the resulting binary parse tree is depicted in Figure 5.8. As shown in the figure, the algorithm outputs colored $\triangle$s to aid in matching repeated rules within the same ID. Similarly, colored $\triangledown$s are used to indicate the reverse complement of some other rule within the same ID. Not shown in this example are the colored pentagons which represent self-complementing variables.

Consider that the example is composed of 11 continuous tandem repeats of the 4-gram *gagc*. It can be seen in Figure 5.8, that ID 4 contains exactly 11 interior nodes, all of which are set to the same grammar rule $A_4$ which has the derivation

$$A_4 \Rightarrow A_1'A_3 \Rightarrow A_1'A_2A_1 \Rightarrow A_1'A_0A_1'A_1 \Rightarrow c'ac'c \Rightarrow gagc.$$

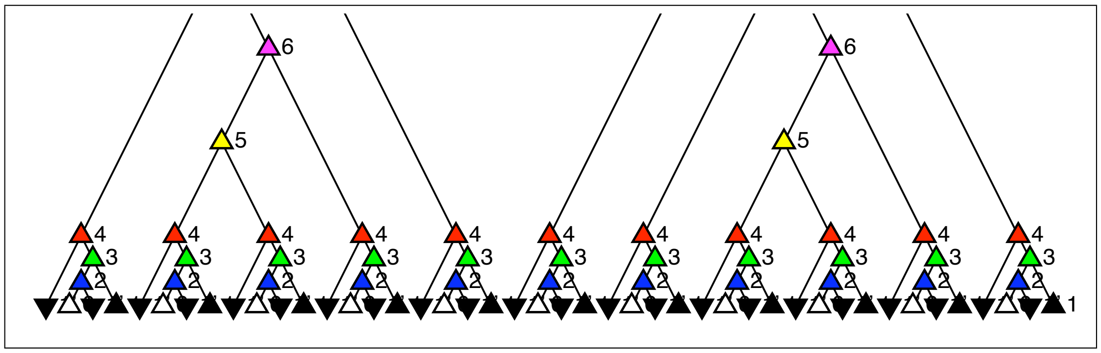


Figure 5.8: Depiction of the inferred structure of an RNA molecule containing 11 *gagc* repeats.

The production $A_4$ has been identified as a significant linguistic structural component which is further validated from the description of this particular molecule. Above ID 4, the algorithm decides that nodes $A_5$, $A_6$, $A_7$, and $A_8$ are also structural elements, as is evident by their repetition. However, visual inspection shows that $A_4$ is likely the most interesting structural element, and that it is repeated many times leads to the other structural pieces above it.

Next, consider the RNA input sequence *gguauuuugguacc*, which is Chain B of the crystal structure of a 14mer RNA containing double *uu* bulges. Applying the algorithm results in the grammar $V(G) = \{A_0, ..., A_9\}$, $T(G) = \{a, c, g, u\}$, $P(G) = \{A_0 \to a, A_1 \to c, A_2 \to A_1' A_1', A_3 \to A_0' A_0', A_4 \to A_2 A_0', A_5 \to A_4 A_0, A_6 \to A_3 A_2, A_7 \to A_5 A_3, A_8 \to A_6 A_5', A_9 \to A_7 A_8\}$, where $S = A_9$ and the resulting binary parse tree is depicted in Figure 5.9. This case is interesting in that the 4-gram *ggua* at the beginning of the sequence has both a tandem repeat, as well as a reverse complement *uacc* located in overlapping fashion at the end of the sequence in *gguacc*. Here is an example of using domain knowledge of the base pairing of RNA. In particular, the algorithm favors the reverse complement rule due to criterion 5 of the scoring metrics. Thus, rule $A_5$ in ID 4 is used in both its forward derivation $A_5 \Rightarrow A_4 A_0 \overset{*}{\Rightarrow} ggua$ as

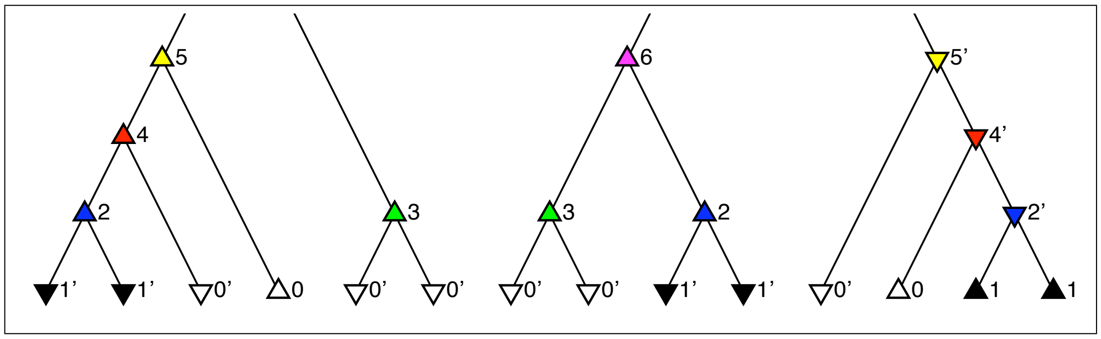


Figure 5.9: Depiction of the inferred structure of a 14mer RNA containing double *uu* bulges.

well as the reverse complement derivation $A'_5 \Rightarrow A'_0 A'_4 \overset{*}{\Rightarrow} uacc$. The result indicates that the 4-gram derived from $A_5$ is structurally significant, in that it base-pairs with $A'_5$, forming a hairpin structure. The remaining terminal elements in between the two $A_5$-derived terminal sub-strings would then be the loop at the end of the hairpin stack. Aside from the digrams generated by $A_2$ and $A_3$, there is not much structurally significant about the loop elements, and so they are added to the parse tree in a minimum-variant fashion.

The next result comes from analyzing the RNA sequence

$$\underline{gcg}uaagg\underline{cgc}g\mathbf{gcac}cuu\mathbf{gugc},$$

which was taken from an example presented in [43], in which there are two non-crossing hairpin structures identified by underline and boldface, for convenience. Analyzing this sequence via RNAFold is presented in Figure 5.10 as a reference for comparison. When analyzed via the algorithm presented in this chapter, the resulting grammar has 16 variables and productions and a partial parse tree as depicted in Figure 5.11. This example also shows that colored pentagons represent the self-complementing variable $A_2 \overset{*}{\Rightarrow} gc$.

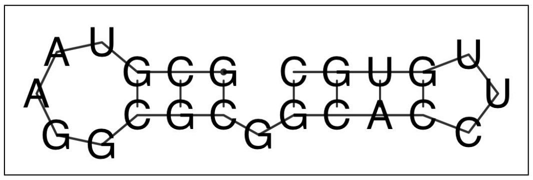


Figure 5.10: Depiction of the secondary structure of an RNA sequence containing two small hairpin structures via the RNAFold software [43].

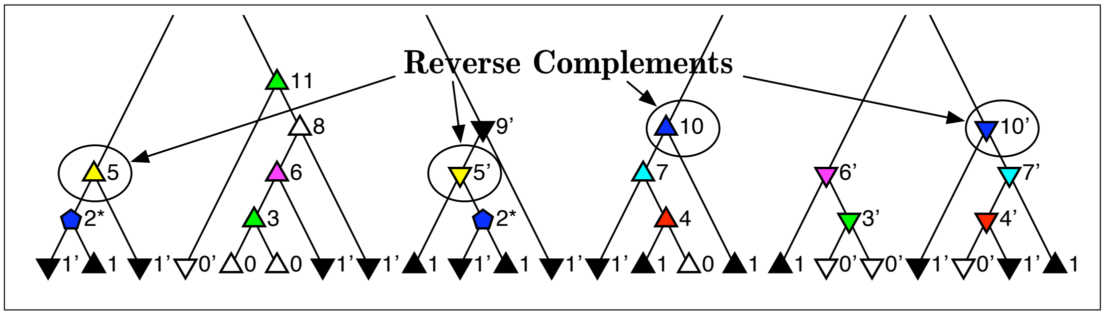
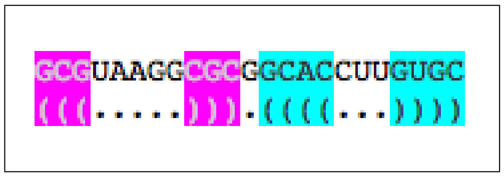


Figure 5.11: Depiction of the inferred structure of an RNA sequence containing two small hairpin structures.

Of particular importance within the parse tree, ID 3 and ID 4 each contain variables and their respective reverse complements, $A_5$ and $A_{10}$. Specifically, the derivations are $A_5 \overset{*}{\Rightarrow} gcg$ and $A'_5 \overset{*}{\Rightarrow} cgc$, which form a hairpin with a stack of three, and $A_{10} \overset{*}{\Rightarrow} gcac$ and $A'_{10} \overset{*}{\Rightarrow} gugc$ which also reverse complement each other resulting in a hairpin of four base pairs. This interpretation matches the result presented in Figure 5.10, which uses a method based upon minimization of free energy within a dynamic programming algorithm. This further ties together the concept of secondary structure physically present in molecules to that linguistically present in information-carrying sequences.

The final example was fabricated during development to represent a more challenging problem of correctly identifying the structural pieces, and then using them at higher IDs. In particular, the 7-gram *gagacat* is repeated seven times with small



```
GCGUAAGGCGCGGCACCUUGUGC
(((.....))).((((...))))
```

Figure 5.12: Symbolic sequence HTML output of an RNA sequence containing two small hairpin structures.

random fragments of length four or less in between each occurrence. The random fragments were inserted to make it difficult for the algorithm to identify repeated structure. The parse tree resulting from analyzing the 67 element sequence is depicted in Figure 5.13. As expected, variable $A_9 \overset{*}{\Rightarrow}$ *gagacat* is repeated seven times at

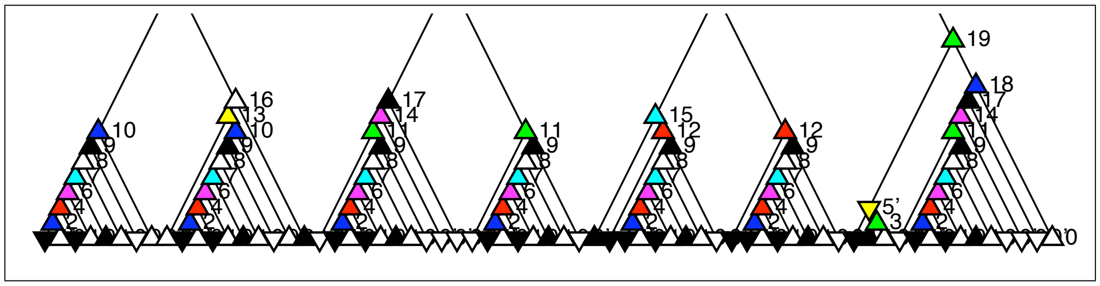


Figure 5.13: Depiction of the inferred structure of a fabricated DNA sequence containing seven tandem repeats of the 7-gram *gagacat* intermixed with small random fragments.

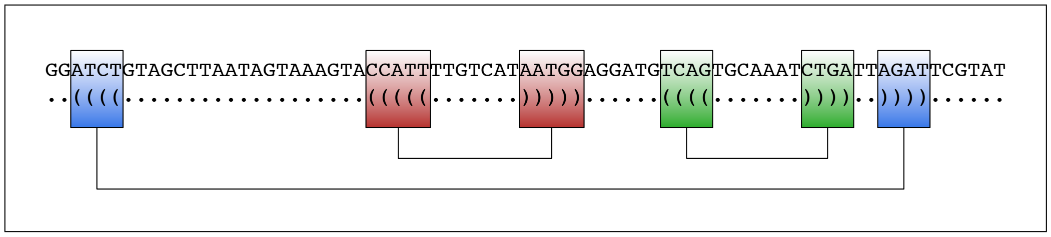
ID 7 indicating that the algorithm was successful in finding the structural component. As was the case in the first example, there are additional structural components discovered above ID 7 that all contain $A_9$ in their derivation of a terminal string. Again, visual inspection would lead the user to see the most common significant element would be the seven repeats of a 7-gram.

**Experiments Applying the Symbolic Sequence**

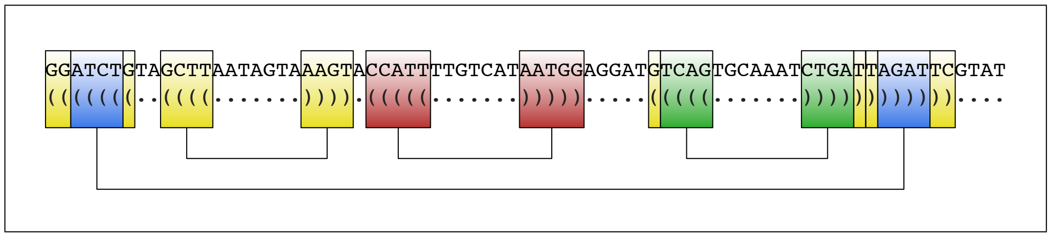
The second set of experiments demonstrate the ability of the inferred grammar to capture the physical secondary structure present in DNA/RNA sequences. In each of the following cases, small segments from the complete genome of *Saccharomyces cerevisiae* mitochondria with known folding behavior were obtained from the NCBI website (http://www.ncbi.nlm.nih.gov) using the accession number NC_001224.1. Each segment was input to the RNAFold web server [43] at http://rna.tbi.univie.as.at/cgi-bin/RNAfold.cgi, producing two different physical structure estimations: 1) Minimum free energy (MFE) structure, and 2) Centroid structure. From the web server help page: "the MFE structure of an RNA sequence is the secondary structure that contributes a minimum of free energy. This structure is predicted using a loop-based energy model and the dynamic programming algorithm introduced by [108]." By comparison, "the centroid structure of an RNA sequence is the secondary structure with minimal base pair distance to all other secondary structures in the Boltzmann ensemble." Both secondary structures are initially presented in "dot bracket notation," similar to the symbolic sequence able to be output from ICYK. Additionally, the web server is able to output a schematic image of the secondary structure in which the sequence is drawn with base paired residues bonded to each other. The information is equivalent to that of the dot bracket notation, but perhaps more appealing visually.

For comparison, ICYK was used to infer each grammar. Subsequently, the additional step was applied to generate an associated symbolic sequence. Appropriate settings were applied to generate only the reverse complement parenthetical sequences, as tandem repeat information is not present in RNAFold outcomes. In all four dot bracket listings presented in Figures 5.14, 5.16, 5.18, and 5.20, the subsequences that pair together have been highlighted with colored boxes. In particular, the regions that are common among all three secondary structures are presented with the same colored boxes. Those regions that present additional or completely different pairing are highlighted using yellow boxes. All four cases include the folded view of the centroid structure to help visualize how the dot bracket notation translates to a three dimensional shape. These additional images are presented in Figures 5.15, 5.17, 5.19, and 5.21.

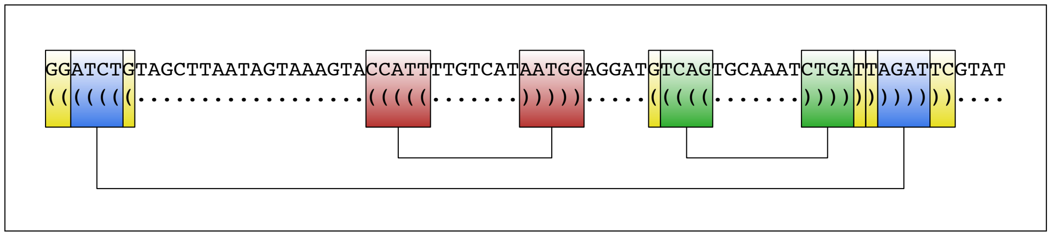
The first segment, $\{68,322-68,396\}$, is shown in Figure 5.14. There is gen-


GGATCTGTAGCTTAATAGTAAAGTACCATTTTGTCATAATGGAGGATGTCAGTGCAAATCTGATTAGATTCGTAT
..((((...................(((((.......)))))......((((.......))))..))))......

(a) ICYK


GGATCTGTAGCTTAATAGTAAAGTACCATTTTGTCATAATGGAGGATGTCAGTGCAAATCTGATTAGATTCGTAT
(((((((..((((.......)))).(((((.......))))).....(((((.......)))))))))))....

(b) MFE


GGATCTGTAGCTTAATAGTAAAGTACCATTTTGTCATAATGGAGGATGTCAGTGCAAATCTGATTAGATTCGTAT
(((((((..................(((((.......))))).....(((((.......)))))))))))....

(c) Centroid

Figure 5.14: Depiction of *Saccharomyces cerevisiae* mitochondria, segment $\{68,322-68,396\}$, in dot bracket notation detailing the predicted secondary structure.

erally good agreement between the grammar-based secondary structure present in

Figure 5.14(a) and the centroid structure determined in Figure 5.14(c). In fact, they are identical except for four extra base pairs present in the centroid result; three of which are $G{\cdot}T$ wobble base pairs. The wobble hypothesis introduced in [20] suggested that there may be some wobble in the pairing, such that a base pair is something other than the standard Watson-Crick base pair. In particular, the literature suggests that the $G \cdot U$ wobble base pair is a fundamental unit of RNA secondary structure [101]. Unfortunately, there are no provisions in the ICYK algorithm to allow for wobble base pairs. This is the most common source of differences between the grammar-based secondary structure and those estimated by the thermodynamic methods. The MFE secondary structure presented in Figure 5.14(b) differs from the other two by including another 4-stack hairpin that happens to include a wobble base pair. This pair was not identified by ICYK due to the minimum structural piece length threshold. This user defined threshold prevents any smaller pieces from consideration, and it defaults to a length of four. Thus, all grammar-based secondary structures contain only linguistic structural pieces of length four or more. The missing piece from Figure 5.14(b) contains a $G \cdot T$ wobble pair at the outer pair; so if the threshold were lowered from four to three, the ICYK algorithm would have captured a 3-stack hairpin at the location of the 4-stack present in the MFE strcture.

The dot bracket notation of Figure 5.14(c) will fold together, presenting a shape similar to the image depicted in Figure 5.15. The MFE shape would differ by including a third inner hairpin in place of the large circular bulge on the left side of the schematic.

The next segment considered, $\{67,309-67,381\}$, is shown in Figure 5.16. These results are similar to the previous segment, in which two of the three structures agree, with the third presenting an additional hairpin stack. However, it is the MFE and centroid structures that are nearly identical, where the MFE structure includes

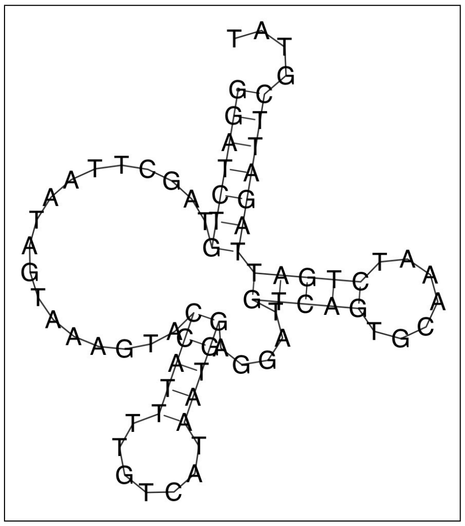
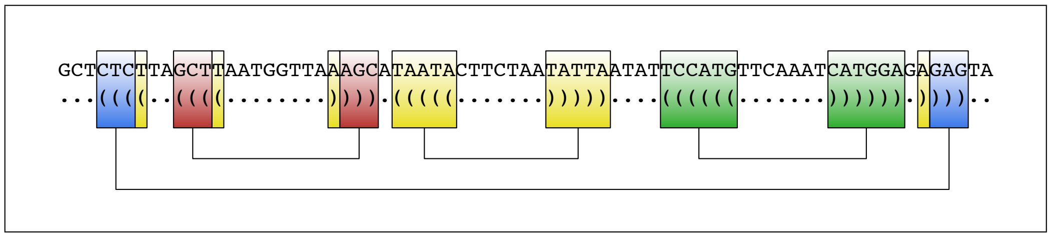


Figure 5.15: Depiction of the centroid-based secondary structure of *Saccharomyces cerevisiae* mitochondria, segment $\{68,322-68,396\}$, via the RNAFold software [43].

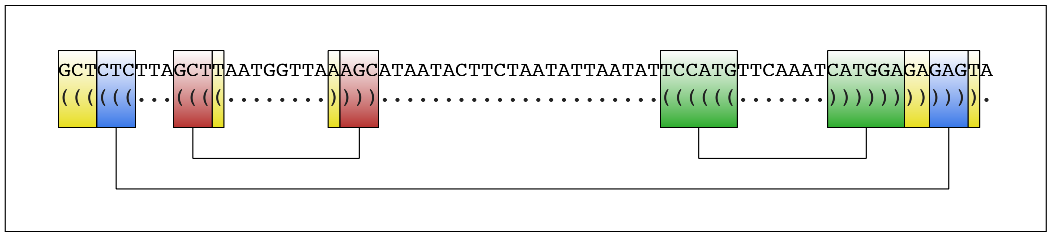
an additional base pair present on the leftmost inner hairpin stack; it is unclear why the centroid structure would not include this base pair. Comparing the MFE structure in Figure 5.16(b) to the ICYK structure in Figure 5.16(a), aside from the additional 5-stack hairpin structure in the center, the outermost base pair differs in an interesting way. The first seven bases of the fragment are $G\mathbf{CT}\underline{\mathbf{C}TCT}$; the underline and boldface are used to detail the overlapping common subsequence that allow the shift in pairing between the two different cases. The shift occurs due to the presence of a $G \cdot T$ wobble pair in the outermost pair depicted in Figure 5.16(b).

The dot bracket notation of Figure 5.16(c) will fold into a shape similar to that depicted in Figure 5.17. This is another "cloverleaf" shape in which there are smaller hairpin elements occurring within the loop of a long distance base pairing. The ICYK secondary structure would present a similar shape, with a third hairpin structure in place of the large loop on the bottom of Figure 5.17.

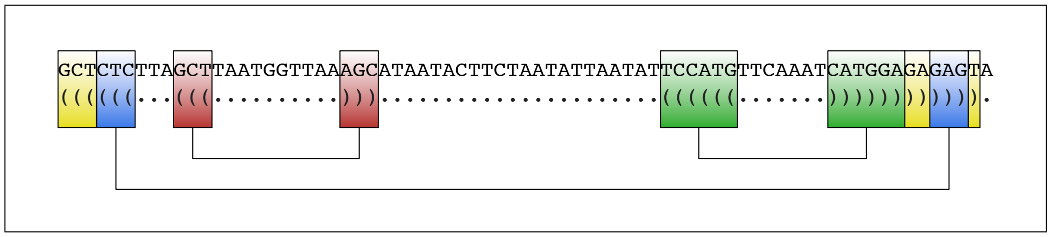
The third segment considered, $\{67,061-67,134\}$, is shown in Figure 5.18. This


GCTCTCTTAGCTTAATGGTTAAAGCATAATACTTCTAATATTAATATTCCATGTTCAAATCATGGAGAGAGTA
...((((..((((........)))).(((((.......)))))....((((((.......)))))).))))..

(a) ICYK


GCTCTCTTAGCTTAATGGTTAAAGCATAATACTTCTAATATTAATATTCCATGTTCAAATCATGGAGAGAGTA
((((((...((((........))))......................((((((.......)))))))))))).

(b) MFE


GCTCTCTTAGCTTAATGGTTAAAGCATAATACTTCTAATATTAATATTCCATGTTCAAATCATGGAGAGAGTA
((((((...(((..........)))......................((((((.......)))))))))))).

(c) Centroid

Figure 5.16: Depiction of *Saccharomyces cerevisiae* mitochondria, segment $\{67,309-67,381\}$, in dot bracket notation detailing the predicted secondary structure.

example details a similar amount of difference when comparing the grammar-based and MFE structures, and then comparing the MFE and centroid structures. Both dot bracket notations in Figure 5.18(a) and Figure 5.18(b) imply two stem cloverleaf shapes in which the rightmost stem contains a bulge in the stack of the hairpin. However, they differ due to two $G \cdot T$ wobble pair on the green stack, and the 4-stack yellow base pair in Figure 5.18(a) is shifted slightly to the 5-stack in Figure 5.18(b). The ICYK algorithm would have scored these pairings the same since there are a total of nine base pair being covered in both cases; that is, to ICYK, the red and yellow pairings in Figure 5.18(a) have an equivalent score to those in Figure 5.18(b). The

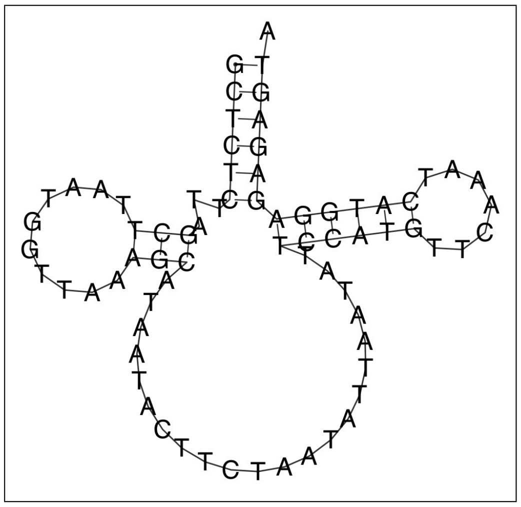
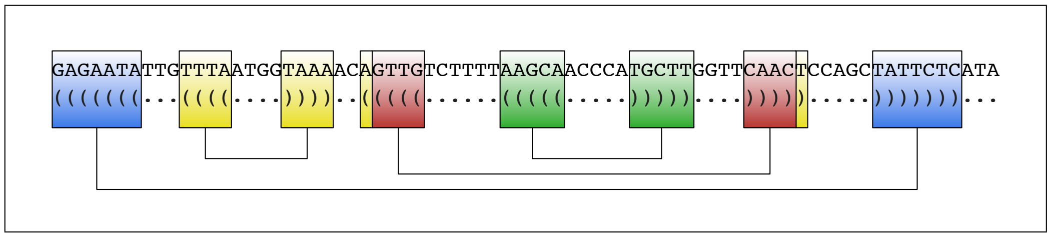


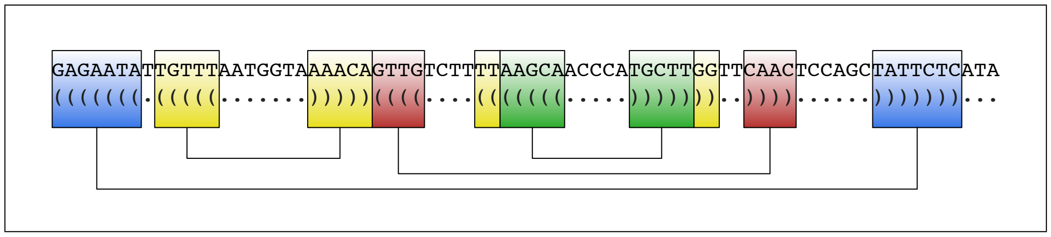
Figure 5.17: Depiction of the centroid-based secondary structure of *Saccharomyces cerevisiae* mitochondria, segment $\{67,309-67,381\}$, via the RNAFold software [43].

centroid structure present in Figure 5.18(c) differs from Figure 5.18(b) by excluding the leftmost 5-stack hairpin; thus its folded shape depicted in Figure 5.19 is more dissimilar to the other two, which would include a hairpin attached to the left side of the large bulge in the middle of the structure.

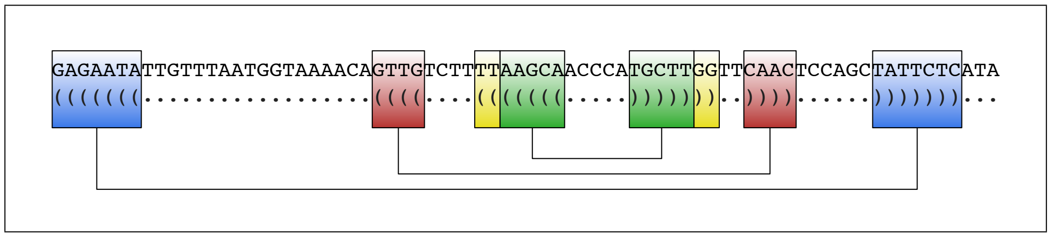
The final segment considered, $\{63,862-63,937\}$, is shown in Figure 5.20. This case resulted in identical MFE and centroid structures, and represents an example of when the grammar-based secondary structure is quite different from the thermodynamic structure. It turns out both Figure 5.20(a) and Figure 5.20(b) have a cloverleaf shape in which there are local hairpins occurring within a long distance pairing. Additionally, the end base pair segments shown in blue are nearly identical. The large difference in identified inner base pair segments is due primarily to the presence of $G \cdot T$ wobble pairs; which appear in all three of the stacks in Figure 5.20(b). As a result of the presence of the wobble pair, the ICYK algorithm is restricted to infer the best available set of length-4 or more base pair segments, which it found in the two



(a) ICYK


GAGAATATTGTTTAATGGTAAAACAGTTGTCTTTTAAGCAACCCATGCTTGGTTCAACTCCAGCTATTCTCATA
(((((((.(((((.......)))))((((....(((((((.....)))))))..))))......)))))))...

(b) MFE


GAGAATATTGTTTAATGGTAAAACAGTTGTCTTTTAAGCAACCCATGCTTGGTTCAACTCCAGCTATTCTCATA
(((((((..................((((....(((((((.....)))))))..))))......)))))))...

(c) Centroid

Figure 5.18: Depiction of *Saccharomyces cerevisiae* mitochondria, segment $\{67,061-67,134\}$, in dot bracket notation detailing the predicted secondary structure.

hairpin structures depicted in Figure 5.20(a). This example demonstrates the effect of the inability of ICYK to take into account the wobble pair concept.

The dot bracket notation of Figure 5.20(b) will fold into the 3-stem cloverleaf shape depicted in Figure 5.21.

The four cases presented here represent common results found in the 24 different segments of the *Saccharomyces cerevisiae* mitochondria genome that were processed. The primary differences between the grammar-based structures and those of the thermodynamic structures are due to the wobble pair concept that is not managed by the ICYK algorithm; a problem left for future research.

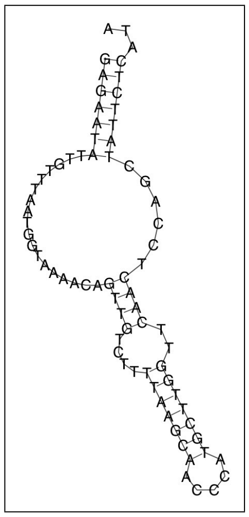


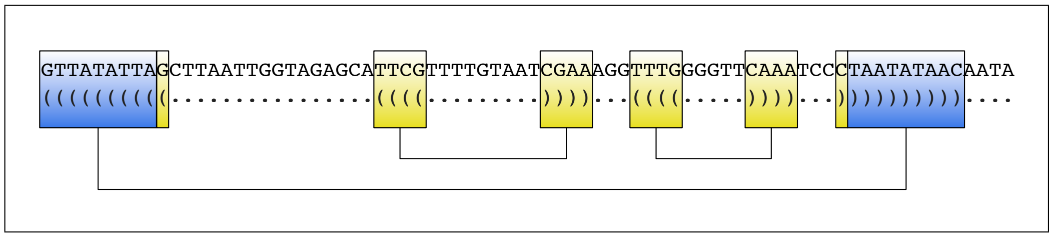
Figure 5.19: Depiction of the centroid-based secondary structure of *Saccharomyces cerevisiae* mitochondria, segment $\{67,061 - 67,134\}$, via the RNAFold software [43].

### 5.2.5 Future Research

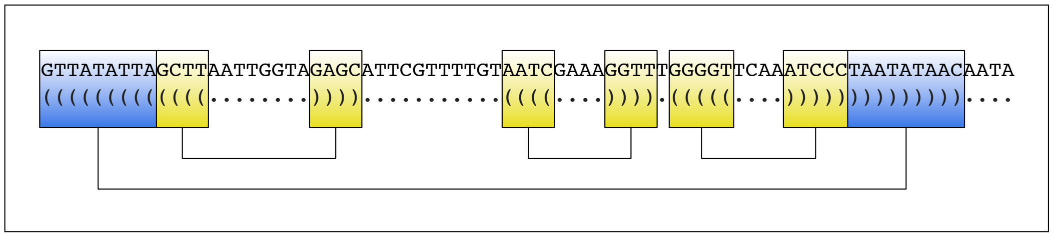
It has been shown in [79, 85] and in Chapters 3 and 4 that inferred grammars may be used as a viable biological sequence distance measure. The framework presented in this chapter may be useful as a structural distance metric. In particular, a grammar may be inferred for a sequence via the method presented followed by applying the classic CYK algorithm using the new grammar and a different sequence. The metric might be the height of the resultant parse tree.

A second application is the identification of significant secondary structures within unknown sequences. However, the application executes within polynomial time. Chapter 6 presents a second framework that provides a linear time algorithm for inferring a CFG of DNA/RNA sequences. Hence, it is feasible for an eventual program to be constructed that picks out significant structures within lengthy fragments within a reasonable amount of time.

As detailed in Section 5.2.4, the present framework does not provide a means for modeling the $G \cdot U$ wobble pair, a fundamental unit of RNA secondary structure


GTTATATTAGCTTAATTGGTAGAGCATTCGTTTTGTAATCGAAAGGTTTGGGGTTCAAATCCCTAATATAACAATA
((((((((((................((((.........))))...((((.....))))...))))))))))....

(a) ICYK


GTTATATTAGCTTAATTGGTAGAGCATTCGTTTTGTAATCGAAAGGTTTGGGGTTCAAATCCCTAATATAACAATA
(((((((((((((........))))...........((((....)))).(((((....))))))))))))))....

(b) MFE and Centroid

Figure 5.20: Depiction of *Saccharomyces cerevisiae* mitochondria, segment $\{63,862-63,937\}$, in dot bracket notation detailing the predicted secondary structure.

[101]. Future research should consider how the wobble pair may be included within a grammar-based model; perhaps an edit grammar would be the best suited to manage the possible "wild-card" behavior of the pairing between a $G$ and both a $C$ and $T/U$.

## 5.3 Conclusions

This chapter presented an algorithm for inferring a novel CFG with the intent of modeling structural regions within biological sequences. Particular focus was applied to RNA data, resulting in a complementary method to thermodynamic modeling for predicting secondary structure. The CYK algorithm for performing sequence categorization given a CFG in CNF was detailed as the basis for the proposed grammar inference algorithm. Preliminary results were provided to demonstrate the viability of the algorithm to generate a useful depiction of the structure within a sequence via a binary parse tree. Future research will focus on ways of applying the inference

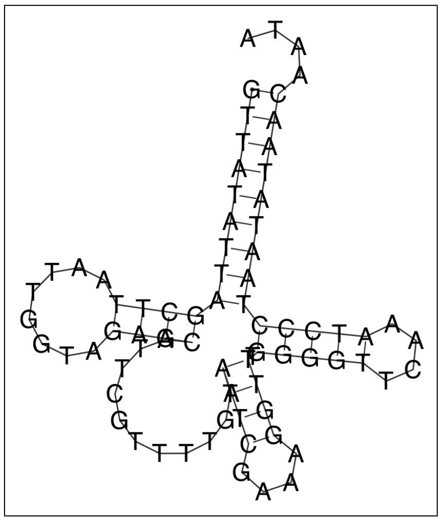


Figure 5.21: Depiction of the centroid-based secondary structure of *Saccharomyces cerevisiae* mitochondria, segment $\{63,862-63,937\}$, via the RNAFold software [43].

algorithm, beginning with constructing a method for converting the binary parse tree into a schematic similar to those found in [44].

The next chapter presents a second approach to inferring CFGs for DNA and RNA sequences. It improves upon the framework presented in this chapter by introducing a linear-time algorithm for CFG inference. The examples presented in this chapter are revisited in the next chapter to validate the approach by showing similar inferred structures.

# Chapter 6

# Linear-Time CFG Inference of DNA/RNA Sequences

In this chapter, we continue the grammar inference work started in Chapter 5 which resulted in a polynomial-time algorithm that could infer a context-free grammar (CFG) for a sequence. The result was processed into a symbolic sequence that could potentially be used in applications that require secondary structure information. Unfortunately, the usefulness of the ICYK algorithm suffers due to its algorithmic order being of polynomial time. Using this fact as motivation, we present a novel CFG inference algorithm that operates in linear time. The method is validated in two ways: 1) the algorithm is used to infer grammars of the examples used in demonstrating ICYK–we compare the two inferred grammars for the relevant sequences, 2) we use a post-processed symbolic sequence as supplemental information to a modified version of GramAlign the multiple sequence alignment (MSA) algorithm introduced in Chapter 3. We compare the alignment-quality of the modified algorithm with that of existing algorithms. The progressive alignment algorithm retains its use of a grammar-based distance metric to determine the order in which biological sequences are to be pairwise aligned. The progressive alignment occurs via pairwise aligning new sequences with an ensemble of the sequences previously aligned. The scoring mechanism used by the progressive alignment is modified to use both the primary structure of the sequences as well as the secondary structure inferred by the proposed algorithm. The performance of the modified MSA algorithm using the inference method is validated via comparison to popular progressive multiple alignment approaches, ClustalW, MAFFT, MUSCLE, and PSAlign using the BRAliBase 2.1 database of RNA alignment files. The modified version of GramAlign has successfully built multiple alignments comparable to other programs with overall improvements due to the inferred secondary structure information.

## 6.1 Background

Motivation was presented in Chapter 5 for developing a method to model the secondary structure necessarily present in DNA and RNA. The result was a polynomial-time algorithm for inferring a CFG, which has the power to model both repeated subsequences and biological palindromes; that is, a special version of palindromes occur in DNA and RNA in which the Chargaff rules are enforced on the second half of the palindrome. This provides the ability for the fragment to fold and bond with itself thus generating spatially functional macromolecules.

The primary drawback of our grammar-based information-theoretic approach in Chapter 5 was the polynomial-time order of execution. In [71] and [73] a linear-time algorithm is presented for inferring CFGs for arbitrary inputs, including biological data. One problem with the algorithm is the inability to make use of available domain knowledge–the Chargaff rules. However, an attempt to modify the algorithm is presented in [13] that operates specifically on DNA sequences and makes use of the Chargaff base pairing rules to generate a more compressed model. Unfortunately, the proposed modification is ultimately too simplistic to capture the necessary complexity present within DNA and RNA sequences.

Thus, we propose a linear-time algorithm that significantly modifies the Sequiter algorithm [71] and the subsequent DNASequiter algorithm [13]. The resulting CFG is then used to identify subsequences that are reverse complemented. A preliminary application is presented in which the location of these structural fragments are provided to a modified version of GramAlign, the MSA algorithm proposed in Chapter 3. Results are presented showing the potential of the grammar inference algorithm for detecting structure in DNA and RNA sequences.

## 6.2 Inference Algorithm

The starting point for the work in [13] is from a grammar inference algorithm called Sequiter first introduced in [71]. The Sequiter algorithm is able to take any finite-length sequence and infer a representative CFG. The primary objective being a good approximation to that of the original for a better understanding of the machine that generated the sequence under scrutiny.

### 6.2.1 Sequiter Algorithm

In general, reverse-engineering a set of rules governing a sequence involves searching for various repetitions in the sequence. When repetitions are found, a rule might be generated where a single symbol is used to represent that repetition. It is these rules of repetition that formulate the grammar. The question is how should an algorithm search for repetitions. The Sequiter algorithm operates by using the two rules:

1. *Digram uniqueness* requires that no two symbols can appear next to each other more than once in the grammar.
2. *Rule utility* requires that any rule in the grammar must be used at least twice somewhere else within the grammar, with the exception of the start rule, $S$.

These rules govern the creation of a compact CFG. To better understand these rules, consider the following example adapted from [13].

**Sequiter Example**

Suppose the observed DNA sequence is *acgtcgacgt*. After the first letter is processed, the initial start rule, $S$, is created,

$$S \rightarrow a.$$

As each subsequent letter is input to the algorithm, the two Sequiter rules are always checked for violation. After the first five elements, the grammar has the single start rule

$$S \rightarrow acgtc.$$

After the $c$ is added, an internal table containing digrams from all production bodies is given by $T = \{ac, cg, gt, tc\}$. As can be seen, all digrams are unique. The next letter input to the algorithm is $g$, which results in

$$S \rightarrow acgtcg$$

and a digram table of $T = \{ac, \underline{cg}, gt, tc, \underline{cg}\}$. When $g$ is added, the digram uniqueness rule is violated and needs to be addressed. The violation is managed by creating a new grammar rule, $A_0 \rightarrow cg$, and replacing all occurrences of the digram $cg$ with the new rule symbol, $A_0$. The resulting grammar is

$$S \rightarrow aA_0tA_0$$

$$A_0 \rightarrow cg$$

with a digram table of $T = \{aA_0, A_0t, tA_0, cg\}$, which contains no repeating digrams. Rule utility is also satisfied since $A_0$ is used twice in rule $S$, and $S$ is the start rule. The next two letters input are $ac$ which are appended to $S$ giving

$$S \rightarrow aA_0tA_0ac$$

$$A_0 \rightarrow cg$$

and a digram table of $T = \{aA_0, A_0t, tA_0, A_0a, ac, cg\}$. Adding the letter $g$ to $S$ and checking the rules

$$S \rightarrow aA_0tA_0a\underline{cg}$$

$$A_0 \rightarrow \underline{cg}$$

results in

$$S \rightarrow aA_0tA_0aA_0$$

$$A_0 \rightarrow cg$$

with a digram table of $T = \{\underline{aA_0}, A_0t, tA_0, A_0a, \underline{aA_0}, cg\}$. Because digram uniqueness is violated, another grammar rule is added, $A_1 \rightarrow aA_0$. Making all the appropriate changes results in the new grammar

$$S \rightarrow A_1tA_0A_1$$

$$A_0 \rightarrow cg$$

$$A_1 \rightarrow aA_0$$

and the digram table $T = \{A_1t, tA_0, A_0A_1, cg, aA_0\}$. Finally, $t$ is added to $S$ resulting in

$$S \rightarrow A_1tA_0A_1t$$

$$A_0 \rightarrow cg$$

$$A_1 \rightarrow aA_0$$

with the digram table $T = \{\underline{A_1t}, tA_0, A_0A_1, \underline{A_1t}, cg, aA_0\}$, which leads to a rule addition $A_2 \rightarrow A_1t$, giving

$$S \rightarrow A_2A_0A_2$$

$$A_0 \rightarrow cg$$

$$A_1 \rightarrow aA_0$$

$$A_2 \rightarrow A_1t.$$

At this point the digram table contains $T = \{A_2A_0, A_0A_2, cg, aA_0, A_1t\}$, and so the digram uniqueness requirement is satisfied. However, notice that rule $A_1$ is only used once on the right-hand side of the grammar rules, in rule $A_2$. This is a violation of rule utility, in that every rule needs to be used at least twice. So, $A_1$ needs to be deleted and the grammar rules must be changed accordingly. The result is

$$S \to A_2A_0A_2$$

$$A_0 \to cg$$

$$A_2 \to aA_0t$$

with a digram table of $T = \{A_2A_0, A_0A_2, cg, aA_0, A_0t\}$, which does not have any repetitions. Also, $A_0$ and $A_2$ are both used twice in grammar production bodies.

The application of interest to [13] is the compressibility of DNA sequences using Sequiter-inferred grammars. The authors introduce an interesting modification to Sequiter in order to operate specifically on DNA sequences. Their proposed modification utilizes reverse complements in their grammar inference algorithm called DNASequiter.

**DNASequiter Example**

Recall the following properties of reverse complements in DNA/RNA sequences:

- The complement of $a$ is $t/u$ and of $c$ is $g$. The $x'$ notation is used to imply complement, so $a' = t/u$, $t'/u' = a$, $c' = g$ and $g' = c$.
- The reverse complement of two DNA/RNA sequences $x$ and $y$ satisfies $(xy)' = y'x'$. For example, say $x = acg$ and $y = tac$. Then $(xy) = acgtac \implies (xy)' = gtacgt = (gta)(cgt) = y'x'$.
- The reverse complement of the reverse complement of a DNA/RNA sequence is the original sequence. For example, say $x = acg$. Then, $((x)')' = ((acg)')' = (cgt)' = acg = x$.

In light of these properties, the algorithm presented in [13] adds a new rule to the two pre-existing rules of Sequiter. The resulting DNASequiter rules are:

1. Digram uniqueness.

2. Rule utility.

3. *Reverse complement digram uniqueness and rule utility* requires that digram uniqueness and rule utility hold for reverse complements as well as the original elements. Note that only one grammar rule is created. Whenever the rule or its complement is used, either case counts toward satisfying the rule utility requirement.

Based on this rule addition, reconsider the example DNA sequence *acgtcgacgt*. The internal table, $T$, used in Sequiter is replaced with the three tables: 1) $T_{\mathrm{TR}}$ to hold tandem repeat digrams, 2) $T_{\mathrm{RC}}$ to hold reverse complement digrams, and 3) $T_{\mathrm{SC}}$ to hold self-complementing digrams. Just as before, we begin with the start rule

$$S \rightarrow a$$

followed by the addition of $c$, which gives

$$S \rightarrow ac$$

with a tandem repeat digram table, $T_{\mathrm{TR}} = \{ac\}$, and a reverse complement digram table, $T_{\mathrm{RC}} = \{gt\}$, since $(ac)' = gt$. Next add $g$ to give

$$S \rightarrow acg$$

and digram tables $T_{\mathrm{TR}} = \{ac\}$, $T_{\mathrm{RC}} = \{gt\}$, and $T_{\mathrm{SC}} = \{cg\}$. For clarity, we have introduced a third table to separate the digrams that are their own reverse complement, or self-complementing, such as the case of $(cg)' = cg$. When $t$ is added, we have the grammar

$$S \rightarrow acgt$$

with the digram tables

$$T_{\text{TR}} = \{\underline{ac}, \mathbf{gt}\}$$

$$T_{\text{RC}} = \{\mathbf{gt}, \underline{ac}\}$$

$$T_{\text{SC}} = \{cg\}.$$

As a result, we have a reverse complement digram uniqueness violation. Hence, we create the rule $A_0 \rightarrow ac$, resulting in the grammar

$$S \rightarrow A_0A_0'$$

$$A_0 \rightarrow ac$$

and digram tables

$$T_{\text{TR}} = \{ac\}$$

$$T_{\text{RC}} = \{gt\}$$

$$T_{\text{SC}} = \{A_0A_0'\}.$$

Adding the next three letters, $cga$, does not cause any rule violations. So, consider the addition of the subsequent $c$,

$$S \rightarrow A_0A_0'cg\underline{ac}$$

$$A_0 \rightarrow \underline{ac},$$

when, followed by a check of the grammar rules, results in

$$S \rightarrow A_0A_0'cgA_0$$

$$A_0 \rightarrow ac$$

with digram tables containing

$$T_{\text{TR}} = \{\underline{A_0'c}, \mathbf{gA_0}, ac\}$$

$$T_{\text{RC}} = \{\mathbf{gA_0}, \underline{A_0'c}, gt\}$$

$$T_{\text{SC}} = \{A_0A_0', cg\}.$$

Here we have another reverse complement digram uniqueness violation because $(A_0'c)' = gA_0$. So a new rule is created, $A_1 \to A_0'c$, giving

$$\begin{aligned} S &\to A_0A_1A_1' \\ A_0 &\to ac \\ A_1 &\to A_0'c \end{aligned}$$

and digram tables

$$\begin{aligned} T_{\text{TR}} &= \{A_0A_1, ac, A_0'c\} \\ T_{\text{RC}} &= \{A_1'A_0', gt, gA_0\} \\ T_{\text{SC}} &= \{A_1A_1'\}. \end{aligned}$$

With no violations, $g$ is input to the algorithm, which does not result in any interesting behavior. So, consider the addition of the final $t$, which begins with the grammar

$$\begin{aligned} S &\to A_0A_1A_1'\underline{gt} \\ A_0 &\to \underline{ac} \\ A_1 &\to A_0'c. \end{aligned}$$

After checking the grammar rules and their complements, we get

$$\begin{aligned} S &\to A_0A_1A_1'A_0' \\ A_0 &\to ac \\ A_1 &\to A_0'c \end{aligned}$$

with digram tables

$$\begin{aligned} T_{\text{TR}} &= \{\underline{A_0A_1}, \mathbf{A_1'A_0'}, ac, A_0'c\} \\ T_{\text{RC}} &= \{\mathbf{A_1'A_0'}, \underline{A_0A_1}, gt, gA_0\} \\ T_{\text{SC}} &= \{A_1A_1'\}. \end{aligned}$$

With the reverse complement digram uniqueness rule in violation, the algorithm creates a new rule $A_2 \rightarrow A_0A_1$, which gives

$$\begin{aligned} S &\rightarrow A_2A_2' \\ A_0 &\rightarrow ac \\ A_1 &\rightarrow A_0'c \\ A_2 &\rightarrow A_0A_1. \end{aligned}$$

But now rule $A_1$ violates rule utility, as it is only used in the rule body of the $A_2$ production. In response, the algorithm deletes rule $A_1$ and makes appropriate changes to get

$$\begin{aligned} S &\rightarrow A_2A_2' \\ A_0 &\rightarrow ac \\ A_2 &\rightarrow A_0A_0'c \end{aligned}$$

with final digram tables containing

$$\begin{aligned} T_{\mathrm{TR}} &= \{ac, A_0'c\} \\ T_{\mathrm{RC}} &= \{gt, gA_0\} \\ T_{\mathrm{SC}} &= \{A_2A_2', A_0A_0'\}. \end{aligned}$$

Comparing this grammar to that of the Sequiter output may not show a dramatic difference in terms of the compressibility as was the goal of [13]. In fact, by measuring grammars with the number of symbols in production bodies, this grammar is only one less symbol than the Sequiter-inferred grammar. However, this grammar captures some additional information at a higher level. Notice the start rule derivation is one variable followed by its reverse complement. This implies the entire sequence is a biological palindrome in which the second half of the sequence is the reverse complement of the first half. This can be an important piece of information considering the mechanical nature of DNA and RNA fragments which form three-dimensional shapes due to folding and subsequent hydrogen bonding. Thus, the secondary structure present in biological sequences is also necessarily present in the sequence schematics. A CFG is able to model the physical secondary structure through palindromes, which are also referred to as secondary structure in a grammar.

Unfortunately, the DNASequiter algorithm is too simplistic to consistently achieve an accurate estimation of many real biological sequences. To see why, consider the following example.

**DNASequiter Example Failure**

A more sophisticated inference algorithm is necessary to capture a better representation of the secondary structure present in DNA/RNA sequences. For example, reconsider the DNASequiter algorithm as applied to the RNA sequence

$$\underline{gcg}uaagg\underline{cgc}g\mathbf{gcac}cuu\mathbf{gugc},$$

which was previously analyzed in Chapter 5 and was adapted from an example presented in [43]. As determined by hand-analysis, this RNA fragment contains two non-crossing hairpin structures identified by underlines and boldface, for convenience.

The algorithm begins by appending eight bases to $S$ before any governing inference rules are violated. The current grammar is

$$S \to gcguaagg$$

with digram tables of

$$T_{\mathrm{TR}} = \{gu, aa, ag, gg\}$$

$$T_{\mathrm{RC}} = \{ac, uu, cu, cc\}$$

$$T_{\mathrm{SC}} = \{gc, cg, ua\}.$$

Append $c$ to give

$$S \to \underline{gc}guaag\underline{gc}$$

with digram tables

$$T_{\text{TR}} = \{gu, aa, ag, gg\}$$

$$T_{\text{RC}} = \{ac, uu, cu, cc\}$$

$$T_{\text{SC}} = \{\underline{gc}, cg, ua, \underline{gc}\}.$$

The first occurrence of a digram uniqueness violation occurs in the self-complementing table. This violation can be thought of as either a tandem repeat, or a reverse complement; it turns out this ambiguity is the downfall of this algorithm. Continuing on, we create the rule $A_0 \rightarrow gc$, resulting in the grammar

$$S \rightarrow A_0^* guaag A_0^*$$

$$A_0 \rightarrow gc$$

where the superscript $*$ is introduced to help identify variables that are self-complementing. The current digram tables are

$$T_{\text{TR}} = \{A_0^* g, gu, aa, ag, gA_0^*\}$$

$$T_{\text{RC}} = \{cA_0^*, ac, uu, cu, A_0^* c\}$$

$$T_{\text{SC}} = \{gc, ua\}.$$

Appending the next $g$ causes a digram uniqueness rule violation. The initial grammar is

$$S \rightarrow \underline{A_0^* g} uaag \underline{A_0^* g}$$

$$A_0 \rightarrow gc$$

where the underlined symbols cause a new rule to be created giving

$$S \rightarrow A_1 uaag A_1$$

$$A_0 \rightarrow gc$$

$$A_1 \rightarrow A_0^* g.$$

Now there is a rule utility violation since $A_0$ is only used one time in rule $A_1$. After correcting the problem, we have the grammar

$$S \rightarrow A_1 uaag A_1$$

$$A_1 \rightarrow gcg$$

and digram tables

$$T_{\mathrm{TR}} = \{A_1u, aa, ag, gA_1\}$$

$$T_{\mathrm{RC}} = \{aA_1', uu, cu, A_1'c\}$$

$$T_{\mathrm{SC}} = \{ua, gc, cg\}.$$

The next $c$ is added without significant effect; a subsequent $g$ is added causing a digram uniqueness rule violation, with the grammar before the correction being

$$S \rightarrow A_1uaagA_1\underline{cg}$$

$$A_1 \rightarrow g\underline{cg}$$

with digram tables

$$T_{\mathrm{TR}} = \{A_1u, aa, ag, gA_1, A_1c\}$$

$$T_{\mathrm{RC}} = \{aA_1', uu, cu, A_1'c, gA_1'\}$$

$$T_{\mathrm{SC}} = \{ua, gc, \underline{cg}, \underline{cg}\}.$$

After a new rule is created, the grammar is

$$S \rightarrow A_1uaagA_1A_2^*$$

$$A_1 \rightarrow gA_2^*$$

$$A_2 \rightarrow cg$$

and digram tables are

$$T_{\mathrm{TR}} = \{A_1u, aa, ag, gA_1, A_1A_2^*, gA_2^*\}$$

$$T_{\mathrm{RC}} = \{aA_1', uu, cu, A_1'c, A_2^*A_1', A_2^*c\}$$

$$T_{\mathrm{SC}} = \{ua, cg\}.$$

The next five letters can be concatenated to $S$ without any rule violations. The subsequent $u$ results in the grammar

$$S \rightarrow A_1ua\underline{ag}A_1A_2^*gcac\underline{cu}$$

$$A_1 \rightarrow gA_2^*$$

$$A_2 \rightarrow cg$$

and digram tables

$$T_{\text{TR}} = \{A_1u, aa, \mathbf{ag}, gA_1, A_1A_2^*, gA_2^*, A_2^*g, ca, ac, cc, \underline{cu}\}$$

$$T_{\text{RC}} = \{aA_1', uu, \underline{cu}, A_1'c, A_2^*A_1', A_2^*c, cA_2^*, ug, gu, gg, \mathbf{ag}\}$$

$$T_{\text{SC}} = \{ua, cg, gc\}.$$

After the violation is corrected, the next character is appended resulting in another digram uniqueness rule violation. The current grammar inference is

$$S \rightarrow A_1u\underline{aA_3}A_1A_2^*gcac\underline{A_3'u}$$

$$A_1 \rightarrow gA_2^*$$

$$A_2 \rightarrow cg$$

$$A_3 \rightarrow ag$$

and digram tables

$$T_{\text{TR}} = \{A_1u, \mathbf{aA_3}, A_3A_1, A_1A_2^*, gA_2^*, A_2^*g, ca, ac, cA_3', \underline{A_3'u}, ag\}$$

$$T_{\text{RC}} = \{aA_1', \underline{A_3'u}, A_1'A_3', A_2^*A_1', A_2^*c, cA_2^*, ug, gu, A_3g, \mathbf{aA_3}, cu\}$$

$$T_{\text{SC}} = \{ua, cg, gc\}.$$

After the new rule is added, $A_3$ appears only once and needs to be removed. After everything is cleaned up the grammar looks like

$$S \rightarrow A_1uA_4A_1A_2^*gcacA_4'$$

$$A_1 \rightarrow gA_2^*$$

$$A_2 \rightarrow cg$$

$$A_4 \rightarrow aag$$

and the digram tables are

$$T_{\text{TR}} = \{A_1u, uA_4, A_4A_1, A_1A_2^*, gA_2^*, A_2^*g, ca, ac, cA_4', aa, ag\}$$

$$T_{\text{RC}} = \{aA_1', A_4'a, A_1'A_4', A_2^*A_1', A_2^*c, cA_2^*, ug, gu, A_4g, uu, cu\}$$

$$T_{\text{SC}} = \{cg, gc\}.$$

The next two letters result in a digram uniqueness rule violation, causing the generation of another rule and resulting in the current grammar of

$$S \to A_1 u A_4 A_1 A_2^* gc A_5 A_4' A_5'$$

$$A_1 \to g A_2^*$$

$$A_2 \to cg$$

$$A_4 \to aag$$

$$A_5 \to ac$$

with digram tables

$$T_{\mathrm{TR}} = \{A_1 u, uA_4, A_4 A_1, A_1 A_2^*, gA_2^*, A_2^* g, cA_5, A_5 A_4', aa, ag, A_4' A_5', ac\}$$

$$T_{\mathrm{RC}} = \{aA_1', A_4' a, A_1' A_4', A_2^* A_1', A_2^* c, cA_2^*, A_5' g, A_4 A_5', uu, cu, A_5 A_4, gu\}$$

$$T_{\mathrm{SC}} = \{cg, gc\}.$$

The final two letters "grow" the $A_5$ rule by causing interleaved and repeated digram uniqueness violations and rule utility violations. That is, $A_5' g$ pairs with the earlier occurrence of $cA_5$ causing the formation of $A_6 \to cA_5$ which turns into $A_6 \to cac$. When the final $c$ is concatenated to $S$, $A_6' c$ pairs with the earlier occurrence of $gA_6$ causing the formation of $A_7 \to gA_6$ which becomes $A_7 \to gcac$. The final grammar is then given by

$$S \to A_1 u A_4 A_1 A_2^* A_7 A_4' A_7'$$

$$A_1 \to g A_2^*$$

$$A_2 \to cg$$

$$A_4 \to aag$$

$$A_7 \to gcac$$

with digram tables

$$T_{\mathrm{TR}} = \{A_1 u, uA_4, A_4 A_1, A_1 A_2^*, gA_2^*, A_2^* A_7, A_7 A_4', aa, ag, A_4' A_7', ca, ac\}$$

$$T_{\mathrm{RC}} = \{aA_1', A_4' a, A_1' A_4', A_2^* A_1', A_2^* c, A_7' A_2^*, A_4 A_7', uu, cu, A_7 A_4, ug, gu\}$$

$$T_{\mathrm{SC}} = \{cg, gc\}.$$

Consider the derivation of the inferred start rule, where we note the location of top-level variable boundaries with an explicit placement of the concatenation operator. We will refer to this type of derivation as the *landscape* of the sequence, and it indicates the inferred secondary structure by identifying the primary pieces that are repeated either directly or reverse complementarily:

$$\begin{aligned} S &\Rightarrow A_1 u A_4 A_1 A_2^* A_7 A_4' A_7' \\ &\Rightarrow A_1 \cdot u \cdot A_4 \cdot A_1 \cdot A_2^* \cdot A_7 \cdot A_4' \cdot A_7' \\ &\Rightarrow \underline{gcg} \cdot u \cdot aag \cdot g\underline{cg \cdot c}g \cdot \mathbf{gcac} \cdot cuu \cdot \mathbf{gugc}. \end{aligned}$$

We have indicated the location of the two real hairpin structures that actually occur in the sequence with an underline and boldface. For convenience, the known mechanical folding of the original sequence is as indicated in the following:

$$\underline{gcg} \cdot uaagg \cdot \underline{cgc} \cdot g \cdot \mathbf{gcac} \cdot cuu \cdot \mathbf{gugc}.$$

As seen in the landscape derivation, the second physical hairpin stem is correctly identified with the grammar-based reverse complement pairing of *gcac* and *gugc*. However, the first hairpin stem has not correctly been identified because the self-complementing *gc* digram caused the algorithm to greedily form a tandem-repeat rule when the second occurrence of *gc* appeared in the sequence. The resulting grammar implies the trigram is only repeated, and does not indicate any hairpin at all. While the DNASequiter algorithm compressed the sequence as intended, it failed to properly model the important secondary structure feature that occurs due to reverse complement palindromes. The algorithm fails in this regard due to the fact that it is attempting to infer a CFG via a greedy left-to-right parsing of the input sequence in order to maintain linear processing time.

### 6.2.2 Framework

As was the case of the algorithm presented in Chapter 5, the goal of the algorithm presented here is to infer a grammar from an unknown sequence with biological secondary structure in mind. The primary problem with the ICYK algorithm was it's polynomial processing time. Thus, the second goal of the algorithm presented here is for the algorithm to have a linear processing time, as was the case for Sequiter and DNASequiter.

Here we introduce Inferring via Sequiter (IVS), a linear time algorithm that builds a CFG by greedy left-to-right parsing of an input DNA/RNA sequence. Similar to DNASequiter, we modify the violations rules in order to build the set of variables in the inferred grammar. However, in order to focus on the reverse complement palindromes present in DNA/RNA, the governing set of algorithm rules is significantly different from those present in DNASequiter. The primary objective of IVS is not necessarily compression as was the case of DNASequiter. Instead, the real objective is to produce a landscape with the similar quality to those that result from the ICYK algorithm from Chapter 5.

The primary problem that occurred in DNASequiter was due to self-complementing digrams, such as $(gc)' = gc$. Consider the trigram $uau$ which has a reverse complement trigram $aua$. Suppose the latter happens to have a $u$ that occurs immediately to the left in the sequence, so we have the 4-gram $uaua$. Because of the self-complementing digrams $(ua)' = ua$ and $(au)' = au$, there are overlapping repeats in the 4-gram $\underline{u\mathbf{au}}\mathbf{a}$. These kinds of overlapping repeats can cause the DNASequiter algorithm to greedily match the leftmost tandem repeat, which is then replaced with a newly created variable head, $\underline{A_{\mathrm{TR}}\mathbf{u}}\mathbf{a}$. When this happens, the reverse complement trigram is no longer seen because a portion of it overlapped with tandem repeat trigram. This problem of tandem repeat fragments overlapping with reverse complement fragments is a result of the self-complementing digrams, and the reason for a new set of violations rules. Keep in mind that in order to capture a more accurate landscape, the following set of governing rules are meant to favor the identification of reverse complement fragment pairs over the discovery of tandem repeat fragments:

1. *Terminal reverse complement trigram uniqueness* requires that the reverse complement of three consecutive terminal symbols can not appear in the grammar.
2. *Terminal trigram uniqueness* requires that no three terminal symbols can appear next to each other more than once in the grammar when at least one of their occurrences is in the body of a variable other than the start rule, $S$.
3. *Variable reverse complement digram uniqueness* requires that the reverse complement of two consecutive symbols can not appear next to each other more than once in the grammar when at least one is a variable.
4. Rule utility.

These rules allow for the creation of a CFG with emphasis on capturing the reverse complement palindromes present in DNA/RNA sequences. Reconsider the example presented to demonstrate the downfall of the DNASequiter algorithm in regards to capturing an appropriate landscape,

$$\underline{gcg}uaagg\underline{cgc}g\mathbf{gcac}cuu\mathbf{gugc}.$$

The algorithm begins by appending three bases to $S$ before checking any rules

$$S \rightarrow gcg$$

with an initial table containing terminal reverse complement trigrams, $T_{\mathrm{RC3}} = \{cgc\}$. It turns out that the next seven terminals are added before anything interesting happens, at which time the grammar is

$$S \rightarrow gcguaaggcg$$

with trigram table

$$T_{\mathrm{RC3}} = \{cgc, acg, uac, uua, cuu, ccu, gcc\}.$$

At this time, a subsequent $c$ is appended to $S$, causing a terminal reverse complement trigram uniqueness violation since the trigram $cgc$ at the end of $S$ appears in $T_{\mathrm{RC3}}$. In response to the rule violation, the new variable $A_0 \rightarrow gcg$ is added to the grammar, the appropriate occurrences in $S$ are replaced with $A_0$, and two new tables are added; one contains the terminal tandem repeat trigrams, and the other contains the variable/terminal reverse complement digrams. The current grammar state is

$$S \rightarrow A_0uaaggA_0'$$
$$A_0 \rightarrow gcg$$

with tables

$$T_{\mathrm{RC3}} = \{cgc, uua, cuu, ccu\}$$
$$T_{\mathrm{TR3}} = \{gcg\}$$
$$T_{\mathrm{RC2}} = \{aA_0', A_0c\}.$$

Notice that the only entry in $T_{\mathrm{TR3}}$ is a terminal-only trigram that is in the production body of a variable other than that of $S$. As a result, only reverse complement structures are searched for in $S$–should another occurrence of a previously identified trigram appear later in $S$, it will be replaced with an associated variable. At this point, six more letters are concatenated to $S$ before another violation occurs. Before the violation happens, the grammar state is

$$S \rightarrow A_0 uaagg A'_0 ggcacc$$

$$A_0 \rightarrow gcg$$

with tables

$$T_{\mathrm{RC3}} = \{cgc, uua, cuu, ccu, gcc, ugc, gug, ggu\}$$

$$T_{\mathrm{TR3}} = \{gcg\}$$

$$T_{\mathrm{RC2}} = \{aA'_0, A_0c, cA_0\}.$$

At this point, $u$ is added to the body of $S$ making a $ccu$ trigram which is an entry in $T_{\mathrm{RC3}}$. Thus, a new rule is created resulting in the grammar state

$$S \rightarrow A_0 ua A_1 A'_0 ggca A'_1$$

$$A_0 \rightarrow gcg$$

$$A_1 \rightarrow agg$$

with tables

$$T_{\mathrm{RC3}} = \{cgc, ccu, gcc, ugc\}$$

$$T_{\mathrm{TR3}} = \{gcg, agg\}$$

$$T_{\mathrm{RC2}} = \{aA'_0, A'_1u, A_0A'_1, cA_0, A_1u\}.$$

The next letter, $u$, is added to $S$ resulting in the digram $A'_1u$, which is an entry in $T_{\mathrm{RC2}}$. Here is an example of variable reverse complement digram uniqueness being violated. In response, the algorithm creates a new rule and adds it to the grammar resulting in the current state

$$S \rightarrow A_0 u A_2 A'_0 ggca A'_2$$

$$A_0 \rightarrow gcg$$

$$A_1 \rightarrow agg$$

$$A_2 \rightarrow aA_1$$

with tables containing

$$T_{\text{RC3}} = \{cgc, ccu, gcc, ugc\}$$

$$T_{\text{TR3}} = \{gcg, agg\}$$

$$T_{\text{RC2}} = \{aA'_0, A'_2a, A_0A'_2, cA_0, A_2u, A'_1u\}.$$

Now $A_1$ appears in only one production body which violates rule utility and results in the grammar

$$S \rightarrow A_0uA_2A'_0ggcaA'_2$$

$$A_0 \rightarrow gcg$$

$$A_2 \rightarrow aagg$$

with tables

$$T_{\text{RC3}} = \{cgc, ccu, gcc, ugc, cuu\}$$

$$T_{\text{TR3}} = \{gcg, agg, aag\}$$

$$T_{\text{RC2}} = \{aA'_0, A'_2a, A_0A'_2, cA_0, A_2u\}.$$

The next three characters are appended without incident when the final $c$ is added. Prior to the final letter, the grammar state is

$$S \rightarrow A_0uA_2A'_0ggcaA'_2gug$$

$$A_0 \rightarrow gcg$$

$$A_2 \rightarrow aagg$$

with tables

$$T_{\text{RC3}} = \{cgc, ccu, gcc, ugc, cuu, cac\}$$

$$T_{\text{TR3}} = \{gcg, agg, aag\}$$

$$T_{\text{RC2}} = \{aA'_0, A'_2a, A_0A'_2, cA_0, A_2u, cA_2\}.$$

After the $c$ addition, the terminal-only trigram $ugc$ is found in $T_{\text{RC3}}$ causing a violation and resulting in a new rule addition giving

$$S \rightarrow A_0 u A_2 A'_0 g A_3 A'_2 g A'_3$$

$$A_0 \rightarrow gcg$$

$$A_2 \rightarrow aagg$$

$$A_3 \rightarrow gca$$

with tables

$$T_{\mathrm{RC3}} = \{cgc, ccu, ugc, cuu\}$$

$$T_{\mathrm{TR3}} = \{gcg, agg, gca, aag\}$$

$$T_{\mathrm{RC2}} = \{aA'_0, A'_2 a, A_0 A'_2, cA_0, A_2 A'_3, cA_2, A'_3 c, A_3 c\}.$$

As was done for the second DNASequiter example, consider the landscape of the inferred grammar, where we note the location of top-level variable boundaries with an explicit placement of the concatenation operator:

$$\begin{aligned} S &\Rightarrow A_0 u A_2 A'_0 g A_3 A'_2 g A'_3 \\ &\Rightarrow A_0 \cdot u \cdot A_2 \cdot A'_0 \cdot g \cdot A_3 \cdot A'_2 \cdot g \cdot A'_3 \\ &\Rightarrow \underline{gcg} \cdot u \cdot aagg \cdot \underline{cgc} \cdot g \cdot \mathbf{gca} \cdot \mathbf{c}cuu \cdot \mathbf{g} \cdot \mathbf{ugc}. \end{aligned}$$

Again, we have indicated the location of the two real hairpin structures that actually occur in the sequence with an underline and boldface. Recall the known mechanical folding of the original sequence is as indicated in the following:

$$\underline{gcg} \cdot uaagg \cdot \underline{cgc} \cdot g \cdot \mathbf{gcac} \cdot cuu \cdot \mathbf{gugc}.$$

In this landscape derivation, both physical hairpin stems are identified with the grammar-based reverse complement pairings of *gcg*-*cgc* and *gca*-*ugc*. The only mistake is the inferred length of the second hairpin stem being three bases instead of four. This error occurred as a result of the greedy left-to-right parsing where two variables are positional neighbors, and so the boundary between grammar pieces shifted so that

$A_2'$ contained the $c$ that should have been part of $A_3$. This is not a significant problem as the landscape was still able to show the presence of the two hairpin structures on either end of the string. Further, this problem could be addressed by processing a string twice; once in the forward direction as was done here, and once in the reverse direction. The resulting inferred grammars could be merged by comparing both landscapes and splitting variables apart wherever there is question about overlapping boundaries. We leave this problem for future research.

### IVS Implementation

Implementing the IVS algorithm is a straight-forward matter of managing three different tables containing $k$-grams and a list of grammar production rules. However, it is somewhat complicated by the fact that variable elements used to compose digrams in one table have a dependency with terminal elements used to compose trigrams in the other tables. Additionally, care must be given due to the reverse complementary nature of the table entries. As a result, the seemingly simple table searches and string manipulations are made more difficult. Thus, the initial design of the IVS implementation is somewhat involved; and so it is completely detailed with flowchart diagrams in Appendix A for documentation's sake.

### Algorithm Complexity

It was shown in [71] that the Sequiter algorithm is of linear order when the implementation uses doubly linked-lists with additional side information to manage the digram table. The IVS algorithm could be implemented with a similar technique in order to achieve a similar linear order. For simplicity, linked lists were not used in the IVS program. Each character, $s_i$ for $i \in \{1, ..., N\}$, where $N$ is the length of the sequence, is processed one at a time. After the letter is retrieved, it is used to form either a trigram consisting of terminals or a digram consisting of a variable and a terminal. The tables are searched for an occurrence of the newly formed $k$-gram; the worst-case being the trigram is not in the reverse complement table so the tandem repeat table is also checked. All tables are implemented using a binary search tree abstract data type; meaning each table containing $T$ $k$-grams can be searched on average in $\mathcal{O}(\log(T))$ with an upper-bound of $\mathcal{O}(T)$ occurring in a completely unbalanced tree. In the event the trigram is found in the tandem repeat table, the worst-case continues with a new variable being created and inserted into both the reverse complement and tandem repeat tables. Again, the binary search tree insertion procedure has an average time of $\mathcal{O}(\log(T))$ and an upper-bound of $\mathcal{O}(T)$. Finally, various $k$-grams can be deleted from the tables, a procedure that has an average time of $\mathcal{O}(\log(T))$ and an upper-bound of $\mathcal{O}(T)$. Thus, an upper-bound of $\mathcal{O}(NT)$ occurs with the selected implementation. However, this is a very loose upper-bound as many times the worst-case path is not taken. In particular, the tandem repeat table never contains any entries until the reverse complement table has been used to identify complementary repeats. Also, each time a digram search fails, the subsequent element is added to a new digram that contains only terminals, and so no table searches are performed at all. Further, it is somewhat unlikely to have completely unbalanced trees for all the $k$-gram tables. Thus, a much more likely order of complexity is given by $\mathcal{O}(N\log(T))$. Additionally, because $T < N$ this complexity can be restated as $\mathcal{O}(N\log(N))$, which is worse than linear but much better than polynomial-time, as was the case of the ICYK algorithm of Chapter 5.

### 6.2.3 Symbolic Sequence

As was the case for the ICYK program developed in Chapter 5, the inferred grammar is not directly applicable without further processing. As a result, the IVS implementation used in Section 6.2.4 outputs a symbolic sequence detailing the secondary structure in FASTA format.

The symbolic sequence is used to indicate where structural pieces have been identified within the input sequence. The symbols used are

- '.' = not contained in a structural element;
- $n$ = contained in a structural element of length $n$.

It was decided for this implementation to keep track of structural pieces without considering the nature of the component. The reason being due to the existence of dynamic structural elements such as riboswitches in which a structural piece can form chemical bonds with different pieces in the same sequence depending on various environmental conditions (e.g., regulations). So, instead of trying to infer the specific behavior of a structural piece, we opt to only identify the presence of a structural piece. Note that the inferred grammar does contain inferred behavior due to the complementary repeat fragments, which may be useful in future research.

### 6.2.4 Example Simulations

#### Comparison with ICYK

We begin with comparing the symbolic sequences inferred by IVS to those inferred via ICYK from Chapter 5. The first example is the RNA input sequence *gagc...gagc* taken from the Jena Library of Biological Macromolecules. This sequence is Chain W of the TRP RNA-binding attenuation protein (TRAP) bound to an RNA molecule containing 11 *gagc* repeats. It was included in the ICYK examples which was capable of identifying tandem repeated fragments. IVS was designed to identify only complementary repeated regions before any tandem repeats are found. As a result, IVS correctly identified no grammar-based fragments.

Next, consider the RNA input sequence *gguauuuugguacc*, which is Chain B of the crystal structure of a 14mer RNA containing double *uu* bulges. Applying the IVS algorithm results in the grammar $V(G) = \{A_0, A_1, A_2, A_3, A_5\}$, $T(G) = \{a, c, g, u, x\}$, $P(G) = \{A_0 \to A_5 A'_1 A'_1 A'_1 A'_1 A'_2 A'_2 A'_5, A_1 \to a, A_2 \to c, A_3 \to x, A_5 \to A'_2 A'_2 A'_1 A_1\}$, where $S = A_0$. Note the addition of production $A_3 \to x$ which is a type-1 rule used as a "catch-all" for any unknown bases present in an input sequence–a feature that was not available in the ICYK implementation. The symbolic sequence representing the landscape is shown in Figure 6.1 clearly showing the correct identification of the

```
gguauuuugguacc
4444......4444
```

Figure 6.1: Depiction of the IVS-inferred structure of a 14mer RNA containing double *uu* bulges.

4-gram complementary repeat regions at the ends. The inferred structure identified by the IVS algorithm matches that of the inferred structure present in the ICYK result. However, the IVS grammar is slightly more compact at a size of 15 body elements compared to 18 used by ICYK.

The next result comes from analyzing the RNA sequence

$$\underline{gcg}uaagg\underline{cgc}g\mathbf{gcac}cuu\mathbf{gugc},$$

which was previously used to demonstrate the operation of the IVS algorithm. The symbolic sequence representing the landscape is shown in Figure 6.2.

```
gcguaaggcgcggcaccuugugc
333.4444333.3334444.333
```

Figure 6.2: Depiction of the IVS-inferred structure of an RNA sequence containing two small hairpin structures.

Analyzing this sequence via RNAFold was already presented in Figure 6.3 as a reference for comparison, and is repeated in Figure 6.3 for convenience. Additionally,

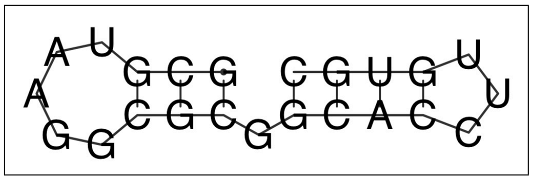


Figure 6.3: Depiction of the structure of an RNA sequence containing two small hairpin structures via the RNAFold software [43].

the parse tree indicating the landscape from the ICYK-inferred grammar is repeated in Figure 6.4 for convenience.

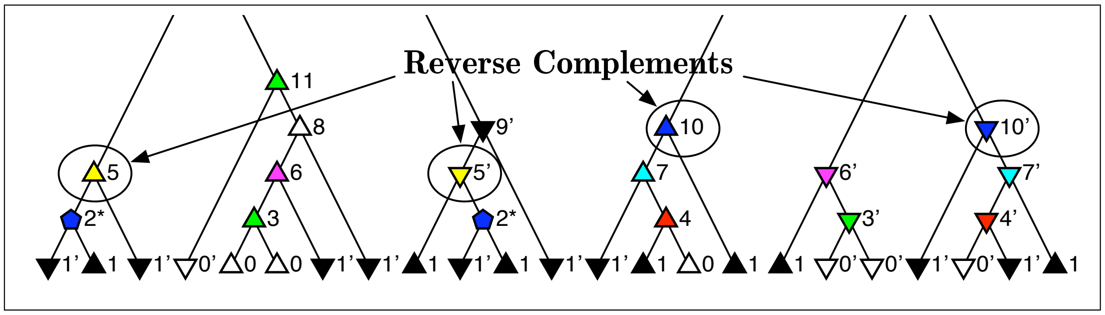


Figure 6.4: Depiction of the ICYK-inferred structure of an RNA sequence containing two small hairpin structures.

Notice the landscape of the symbolic sequence depicted in Figure 6.2 and that of the parse tree present in Figure 6.4. In fact, they are nearly identical with the exception of the number of bases present in the second hairpin structure. In the ICYK-inferred parse tree, rule $A_{10}$ represents the 4-base stem of the rightmost hairpin structure and rule $A_6$ derives the middle reverse complement piece consisting of only three bases. In terms of hand-analysis which agrees with the RNA-fold result in Figure 6.3, the middle structure is not considered meaningful, as the molecules are not able to physically fold into a shape that would allow the chemical bonding of all three reverse complement structures. Generally, mechanical methods assume the least amount of energy is used to generate the physical structures. As previously described, the IVS-inferred symbolic sequence shifts the size of the middle structure with that of the rightmost structure because the pieces overlap and IVS operates as a greedy algorithm which ultimately processed the middle structure before the rightmost structure was encountered. The presence of the rightmost hairpin structure is still intact.

As in the case of the first example, the final example included in the ICYK results will not be meaningful as the fabricated sequence consisted of embedded tandem repeats of the 7-gram *gagacat*. Thus, the results are omitted.

### 6.2.5 Application: Multiple Sequence Alignment

Now that we have an efficient algorithm for inferring secondary structural information present in DNA/RNA sequences, we would like to gather and use that information in order to guide MSAs. That is, secondary structure occurs in biological sequences for various reasons, one of which is to perform specific mechanical functions by chemically folding into three-dimensional shapes. Thus, as in the case of MSA, should sequences perform similar high-level functionality, then their secondary structures should be aligned as well as their primary structures. If we have knowledge of the location of structural pieces within the sequences we are aligning, then we should be able to apply that knowledge in the scoring scheme in order to improve the overall alignment.

We have created a modified version of GramAlign from Chapter 3 that allows for the input of an IVS-inferred symbolic sequence. If enabled to do so, the modified version of GramAlign adds the structural information to the pairwise alignment scoring. Recall the original GramAlign which utilizes three different scoring matrices in the Needleman-Wunsch dynamic programming method to perform pairwise alignment. During the forward phase of the alignment, two matrices are used to score the occurrence of gaps with the third used to score the base pair under scrutiny. The gap matrices were required to keep track of the sophisticated affine scoring that included different penalties marking the beginning of a gap, extending a gap, and noting the difference at the tail ends. The GramAlign mechanism is made more complicated by the fact that after two sequences are aligned, further pairwise alignments take place between a new sequence and the previously determined ensemble sequence. GramAlign has a mechanism in place to keep track of various scoring parameters for every aligned column associated with the ensemble sequence, generally based on the confidence of the column contents.

By comparison, the secondary structural modification to GramAlign is somewhat simplistic to demonstrate the viability of using the inferred structural information generated by IVS. GramAlign is modified in two ways. First, the symbolic sequence information is loaded per each sequence and a score is added to the substitution score matrix used in the forward phase of Needleman-Wunsch. For the base pair being compared, the following scoring is added to the substitution score:

- if neither base is in a grammar piece, nothing is added to the score;
- if only one base being compared is in a grammar piece, a user defined mismatch penalty is applied to the score;
- if both bases being compared are in a grammar piece, a user specified match benefit is added to the score.

In a sense, this simplified scoring mechanism is a “hard-decision” where the actual size of the grammar piece is not taken into account. Perhaps in future work a “soft-decision” scoring scheme could be used that took the grammar piece length into consideration.

The second modification to GramAlign is the addition of the grammar piece length information to the ensemble data structure. In particular, after two sequences have been aligned the ensemble sequence contains column-wise information including symbol percentages that lead to per-column scoring metrics. We now add a new metric that is simply the maximum grammar piece length found during pairwise alignment for the column. Then, subsequent pairwise alignment with the ensemble compares the new sequence symbolic grammar pieces with the maximum grammar piece length previously found in the associated column under scrutiny. Again, future work may focus on more sophisticated methods of keeping grammar pieces from splitting apart, or pre-aligning based on grammar pieces.

To demonstrate the benefit of using the IVS-inferred symbolic sequence information, we performed alignment experiments similar to those from Chapter 3. All results were generated by compiling and executing the respective MSA programs on the same computer; specifically, an Apple MacBook Pro with an Intel Core 2 Duo operating at 2.53 GHz with 4 Gb of system memory and a 3 Mb L2 cache. The experiments were conducted using the unaligned FASTA files from the BRAliBase 2.1 [104] data-set, a sequel that largely extended the original work of the BRAliBase II [34] data-set. Both data-sets constructed their reference alignments using Rfam [37, 38, 33] which is a database of sequence families of structural RNAs, including ncRNA genes as well as *cis*-regulatory RNA elements. Rfam release 9.0 contains 603 families, each represented by an MSA of known and predicted representative members of the family, annotated with a consensus base-paired secondary structure [33]. Compared to BRAliBase II, BRAliBase 2.1 used an updated Rfam version, 7.0, and includes many more RNA families and also varies the number of sequences. The resulting aligned FASTA files from each algorithm were scored using `compalignp`, one of two scoring programs provided with the BRAliBase 2.1 distribution that generates a modified sum-of-pairs score (SPS) defined as the fractional sequence-identity between a trusted reference alignment and a test alignment in [104].

**BRAliBase Experiments**

Alignment files in the BRAliBase 2.1 database are separated into six categories (k2 through k15), each exhibiting an increase in the number of sequences per alignment. In particular, each file within a subdirectory of k$n$ contains $n$ sequences to be aligned with each other. Each category is further divided into 36 directories, each representing an RNA family from the Rfam 7.0 database. The results presented in Table 6.1 detail the average SPS score over each category as aligned by GramAlign version 1.18 both with and without the symbolic sequence information generated by IVS version 0.1, ClustalW version 1.83, PSAlign using ProbCons as the tree generation (no version given, archive created on 8/19/2008), MAFFT version 6.821, and MUSCLE version 3.8.31. Additionally, a fast version was tested for ClustalW, MAFFT and MUSCLE. In particular, the command line options used were `clustalw -quicktree`, `mafft --retree 1` and `muscle -maxiters 1 -diags -sv -distance1 kbit20_3` to incorporate high-speed progressive options. In all cases the default parameters were used for each program.

Table 6.1: Average SPS score for each algorithm for each category offered by the BRAliBase 2.1 test suite.

| **Algorithm** | **k2** | **k3** | **k5** | **k7** | **k10** | **k15** |
|---|---|---|---|---|---|---|
| GramAlign | 0.8089 | 0.8145 | 0.8195 | 0.8222 | 0.8299 | 0.8235 |
| GramAlign w/ IVS | 0.8128 | 0.8176 | 0.8250 | 0.8265 | 0.8379 | 0.8354 |
| PSAlign | 0.6058 | 0.6292 | 0.6570 | 0.6748 | 0.7126 | 0.7169 |
| ClustalW (fast) | 0.7959 | 0.8064 | 0.8205 | 0.8235 | 0.8368 | 0.8455 |
| ClustalW | 0.7959 | 0.8084 | 0.8261 | 0.8337 | 0.8483 | 0.8517 |
| MAFFT (fast) | 0.8254 | 0.8360 | 0.8511 | 0.8584 | 0.8661 | 0.8739 |
| MAFFT | 0.8254 | 0.8396 | 0.8569 | 0.8671 | 0.8756 | 0.8836 |
| MUSCLE (fast) | 0.8332 | 0.8407 | 0.8515 | 0.8581 | 0.8679 | 0.8722 |
| MUSCLE | 0.8332 | 0.8462 | 0.8626 | 0.8759 | 0.8869 | 0.8971 |

Additional statistical information is provided in Table 6.2. This table contains relative score comparisons between alignments generated via GramAlign with no extra structural information to those of GramAlign with the IVS-inferred symbolic sequence information. In all categories, the average SPS score increased when using the IVS structural information. Additionally, the total number of alignments with an improved SPS score was always more than the total number of alignments with a decreased SPS score. Further, the average amount by which the SPS score increased was always more than the average SPS score decrease present in the alignments that had a worse result. The net result was at least an improvement or no change to the alignment in 80% of all categories with the greatest benefit occurring in the larger datasets of k10 and k15 which had an average alignment improvement in over 30% of the cases.

The preliminary results from all experiments show viable promise of the proposed IVS algorithm.

Table 6.2: Comparison of individual SPS scores for GramAlign with and without IVS symbolic information for each category offered by the BRAliBase 2.1 test suite.

| **Statistic** | **k2** | **k3** | **k5** | **k7** | **k10** | **k15** |
|---|---|---|---|---|---|---|
| SPS Increase | 1718 | 1104 | 639 | 397 | 276 | 156 |
| No Change | 5886 | 2856 | 1290 | 745 | 427 | 262 |
| SPS Decrease | 1372 | 875 | 476 | 284 | 142 | 85 |
| Mean SPS Increase | 0.0622 | 0.0480 | 0.0546 | 0.0472 | 0.0476 | 0.0634 |
| Mean SPS Decrease | 0.0526 | 0.0433 | 0.0453 | 0.0445 | 0.0452 | 0.0458 |
| % Dataset Improvement | 19% | 23% | 27% | 28% | 33% | 31% |
| % Dataset Improvement or No Change | 85% | 82% | 80% | 80% | 83% | 83% |

### 6.2.6 Future Research

The IVS algorithm is in its infancy. There are already places where improvements can be made. For example, there is the problem of neighboring and overlapping structural pieces which leads to incorrectly identified structure lengths. One solution to some of the cases is to perform a subsequent inference of the target sequence in reverse. The result could be compared to the original inference resulting in an identification of overlapping pieces. The overlaps could then be adjusted by an efficient post-processing algorithm that takes into account the location of pieces and discounts any that form pseudoknot structures.

As was the case for the ICYK algorithm in Chapter 5, the framework presented here may be useful as a structural distance metric. Where using ICYK may be prohibited by its polynomial order of execution time, the IVS algorithm or its future derivatives may be applied instead due to its low processing time. Thus, a second application is the identification of significant secondary structures within unknown sequences. Additionally, future work may focus on better application of the IVS-inferred symbolic information with regards to the MSA problem. The preliminary method presented here was merely a demonstration of the potential; perhaps even better results may come from re-working the scoring mechanisms in place in the current GramAlign.

## 6.3 Conclusions

This work has presented an efficient algorithm for inferring a novel CFG with the intent of modeling structural regions within biological sequences. Particular focus was applied to RNA data, resulting in a complementary method to thermodynamic modeling for predicting secondary structure. The Sequiter and DNASequiter algorithms for performing grammar-inference were detailed as the basis for the proposed algorithm, IVS. Preliminary results were provided to demonstrate the viability of the algorithm to generate useful structural information necessarily present in a biological sequence. Future research will focus on ways of applying the inference algorithm.

# Chapter 7

# Conclusions

The primary objective of this dissertation is to model biological data with abstract grammars. These grammar models are inferred and applied to efficiently solve various sequence analysis problems present in computational biology.

First we introduce sequence comparison algorithms using a grammar-based sequence distance predicated upon a classic text-based dictionary compression scheme. The grammar-based distance work presented in [79] is adapted to generate a numeric metric useful in each application, including: multiple sequence alignment; relative fragment assembly; and sequence clustering for the purpose of removing redundancy within a dataset. In Chapter 3, the grammar-based distance metric is first used in GramAlign, a multiple sequence alignment program, to guide the order in which sequences are progressively aligned. Subsequently, the same grammar-based distance metric is used in GramContig, a contig arrangement program, to identify the relative locations of fragments from a set; thereby performing fragment assembly relative to a reference sequence. Then, in Chapter 4, a modified version of the grammar-based distance metric is used in GramCluster, a sequence clustering program, to determine accurate partitioning of a set of related biological sequences.

Second we present the first of two grammar inference algorithms with the goal of capturing secondary structural information present in biological sequences. We introduce a novel framework for inferring context-free grammars and their associated parse trees in a polynomial-time algorithm. The grammar can be used to identify significant biological structural information present in the sequences without recourse to thermodynamic considerations. Chapter 5 details how the classic sequence classification method, CYK, is used as the inspiration for ICYK, a context-free grammar inference program. Resultant parse tree outputs demonstrate one possible visualization of significant structural pieces; comparisons show the pieces to be structural in both terms of grammar as well as molecular folding. Parenthetical sequence outputs demonstrate a second visualization of significant structural pieces; comparisons to thermodynamic methods provide validation of the grammar inference framework and more importantly the connection between linguistics and biological structures.

Finally, we present a second grammar inference algorithm that improves upon the first by reducing the algorithm order from polynomial-time to linear-time. In Chapter 6, we detail the novel framework for inferring context-free grammars based on Sequiter [71] in a linear-time algorithm. The grammar is used to identify significant biological structural information necessarily present in DNA or RNA sequences. The inferred secondary structural information is provided to a modified version of GramAlign as supplemental information with the intent on improving pairwise sequence alignment quality by modifying the dynamic programming scoring mechanism.

## 7.1 Future Research

It is shown in [79, 85] and in Chapters 3 and 4 that inferred grammars may be used as a viable biological sequence distance measure. The inference algorithms presented in Chapters 5 and 6 may be useful as a structural distance metric. In particular, a grammar may be inferred for each sequence in a set followed by applying a grammar comparison; for example, the classic CYK algorithm may be applied using one grammar and a different sequence. The metric might be the height of the resultant parse tree. Perhaps the grammars may be compared via parenthetical sequences; for example, the dot bracket output sequences from ICYK or IVS may be compared against each other in a dynamic programming algorithm such as Needleman-Wunsch or Smith-Waterman. In either case, the metric would be the distance that results due to the shortest path resulting from the forward trace; or in the Smith-Waterman case, local minima in the matrix would identify regions of structural similarity. Another possibility is comparing the same parenthetical sequences using the LZ dictionary compression method. As was done for the original sequences in Chapters 3 and 4, the distance would be a calculation that accounts for how a dictionary is extended in going from one parenthetical sequence to another. The biggest problem with this idea is the lack of unique symbols, only '.', '(', and ')'. It turns out the interesting information is the length and position of the structural pieces. Maybe the dot bracket sequences can be encoded first, thereby indicating the run lengths of each symbol. The resulting compressed sequences may provide more useful comparison information.

A second application of the grammar inference work is the identification of significant secondary structures within unknown biological sequences. However, as detailed in Section 5.2.4, neither ICYK nor IVS provide a means for modeling the $G{\cdot}U$ wobble pair, a fundamental unit of RNA secondary structure [101]. Future research should consider how the wobble pair may be included within a grammar-based model; perhaps an edit grammar would be the best suited to manage the possible "wild-card" behavior of the pairing between a $G$ and both a $C$ and $T/U$.

Similar to the wobble pair, differences can occur at the molecular level that are not accounted for by a fixed grammar. When either ICYK or IVS infers a grammar from a sequence it assumes there are no errors, and there is no allowance for small variabilities. An edit grammar allows for production rule modifications, or edits, throughout a derivation. In this way, similar segments within a sequence can be represented with the same grammar production rules, where each rule might have an edit associated with it. This powerful grammar would be able to capture the secondary structures even in the presence of small differences throughout the pieces. The primary challenge here is creating an efficient inference algorithm.

Regarding the linear-time IVS algorithm from Chapter 6, there are areas where improvements can be made. For example, neighboring and overlapping structural pieces can lead to incorrectly identified structure lengths. A solution to many of the cases is to perform a subsequent inference of the target sequence in reverse. Then the forward and reverse inferred landscapes could be compared to identify overlapping structural pieces. The overlaps could be adjusted by an efficient post-processing algorithm that breaks longer runs into common shorter runs.

There is a language governing life–there can not be order without underlying rules. We have shown many connections and relationships between our simple abstract grammar models and physical molecular strands of information. We know that biological sequences obey higher-levels of grammar due to the presence of pseudoknot structures. The challenge is in developing models and inference algorithms that function within a reasonable order of execution time yet are still able to capture the majority of structural information.

# Appendix A

# IVS Diagrams

Initially, the IVS algorithm seemed simple enough to implement directly in ANSI C. However, it quickly spiraled into a large, unmanageable mess. It was clear that implementing the IVS algorithm required careful consideration. As a result, a significant set of flowcharts were constructed to help guide the eventual program. The development flowcharts are included in this appendix in Figures A.1-A.9.

The symbols used within the figures are defined as follows

- tRC3, tTR3, and tRC2 are the $k$-gram tables;
- $S$ is the start rule;
- $Y$ and $Z$ are the penultimate and last indices used to form trigrams and digrams;
- $N$ is the sequence length;
- $i$ and $e_i$ are the current index and the current terminal of the sequence;
- $R$ is the index of the next variable to be added;
- $*x$ implies the terminal or variable within $S$ at position $x$;
- $RC(x)$ implies the reverse complement of $x$;

- $*x.len$ implies the fully-derived length of variable $x$;

- $[h, b]$ is the grammar location of variable $h$ body-index $b$.

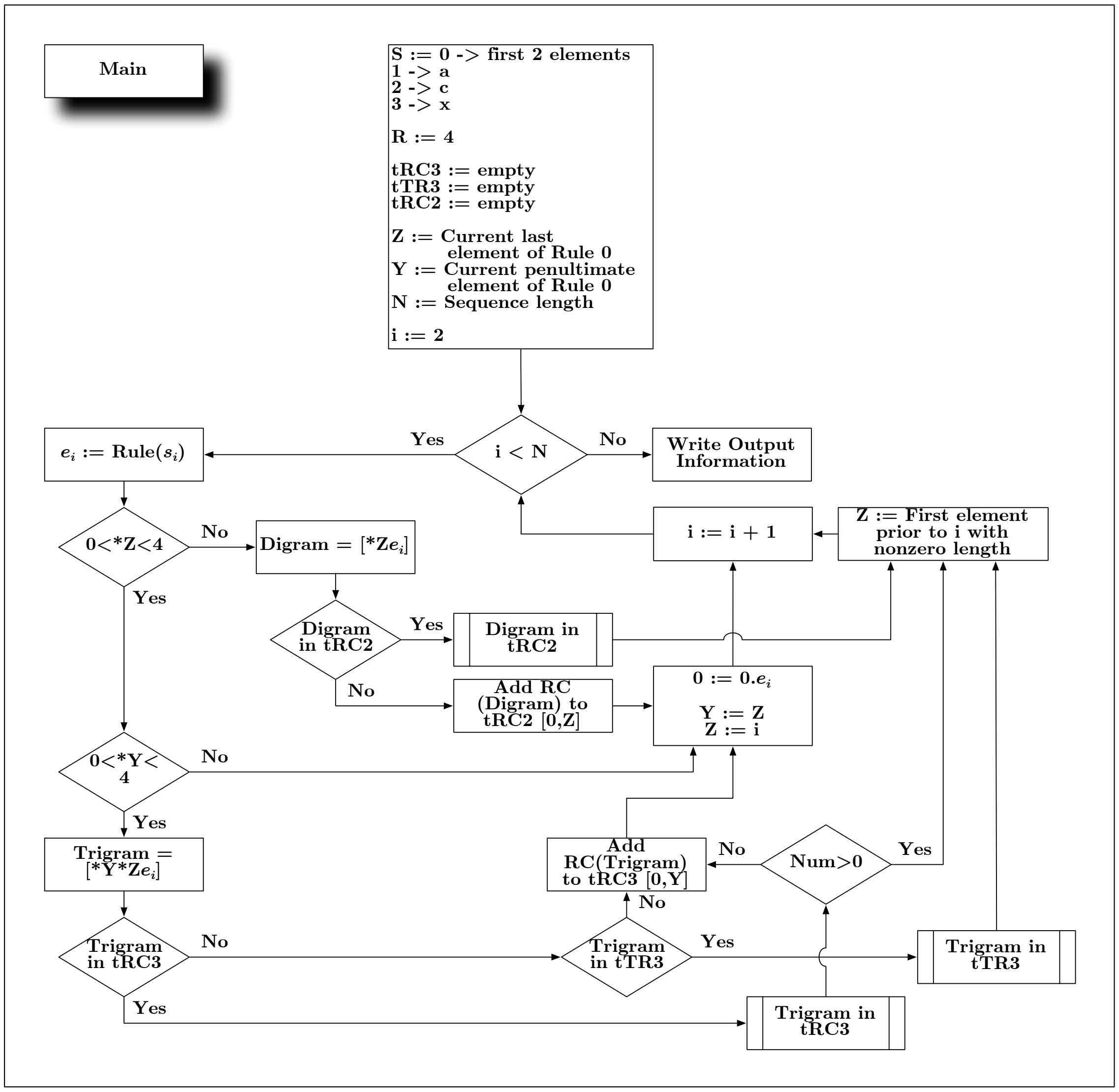


Figure A.1: IVS diagram page 1–main.

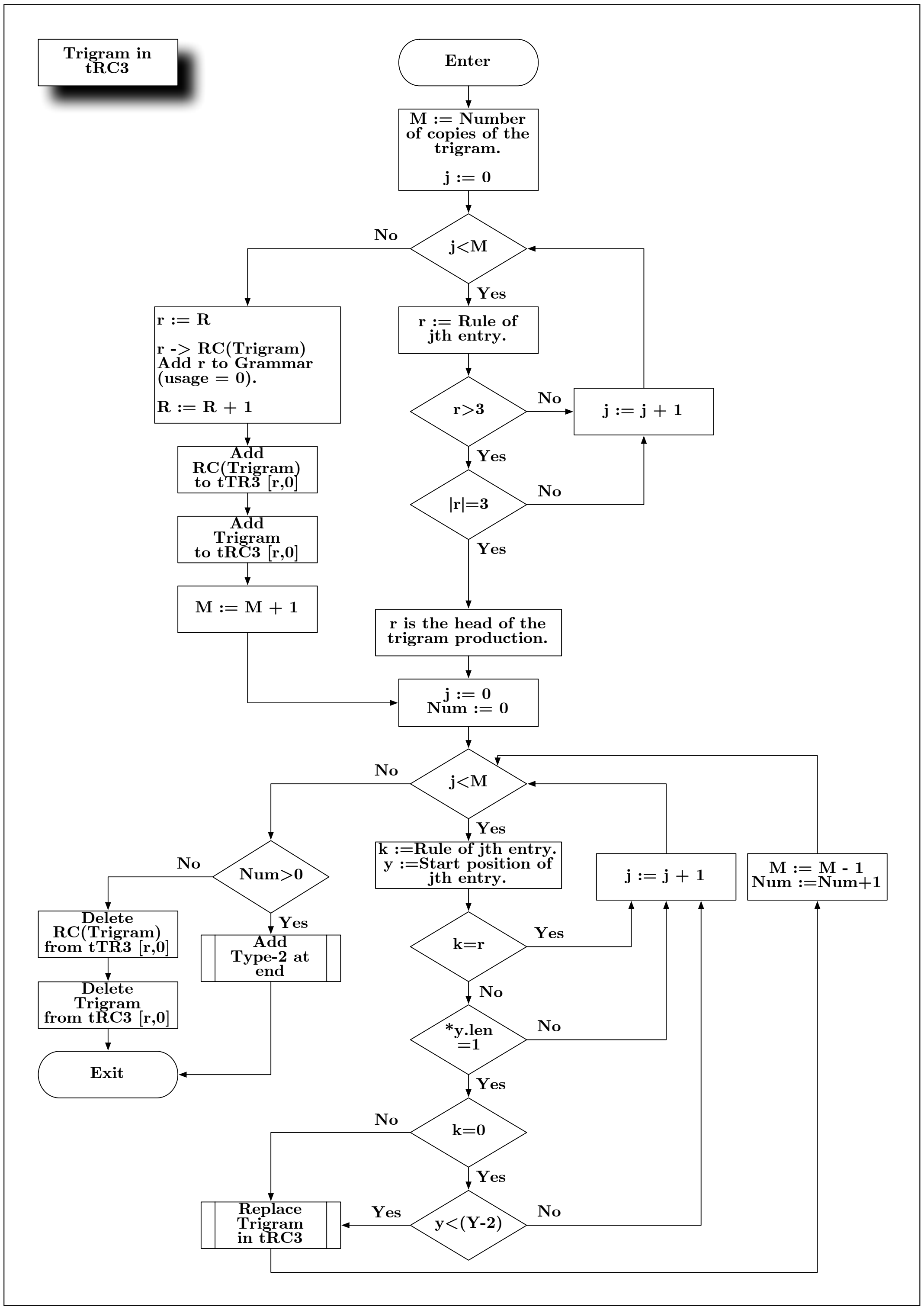


Figure A.2: IVS diagram page 2–trigram in reverse complement trigram table.

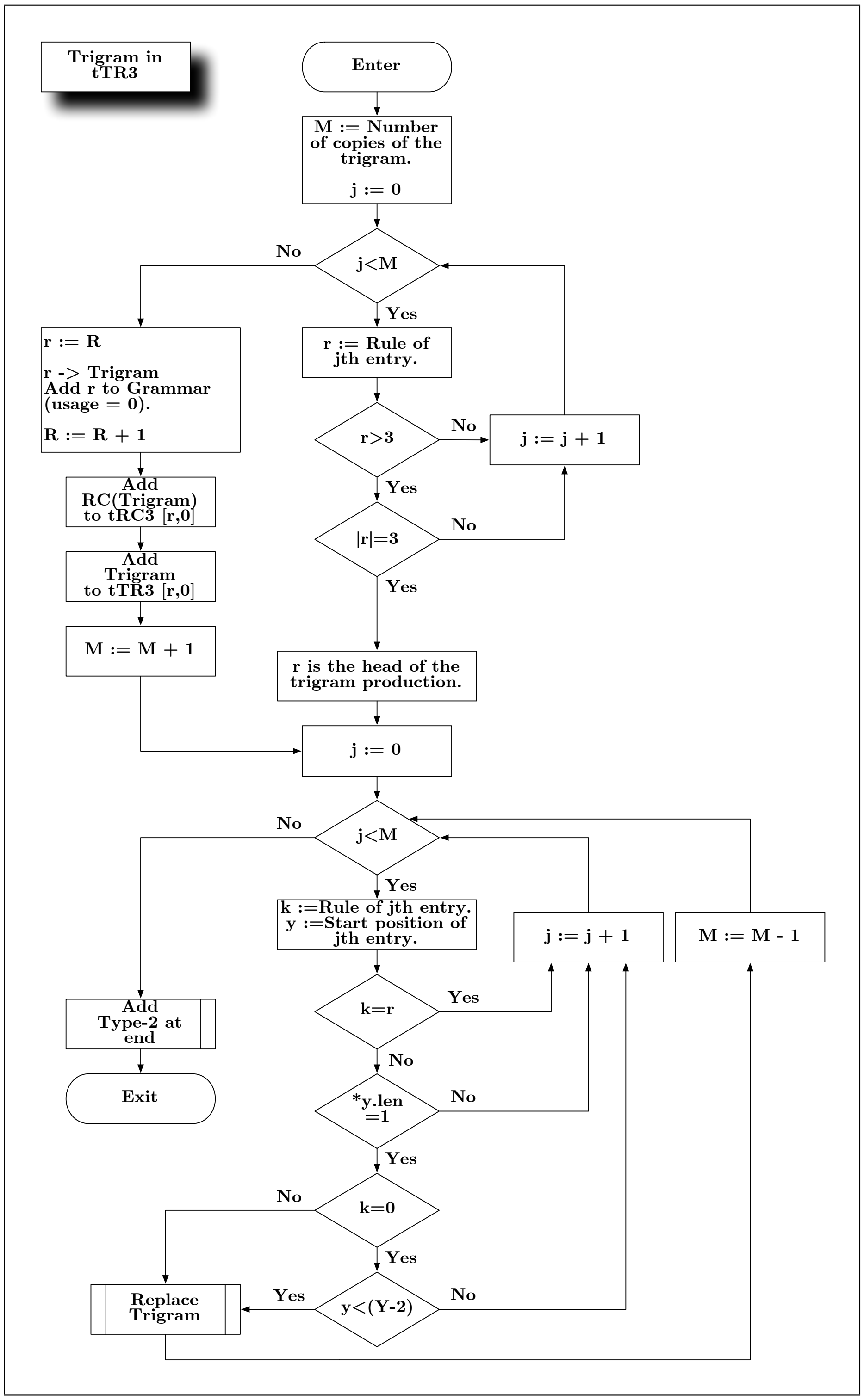


Figure A.3: IVS diagram page 3–trigram in tandem repeat trigram table.

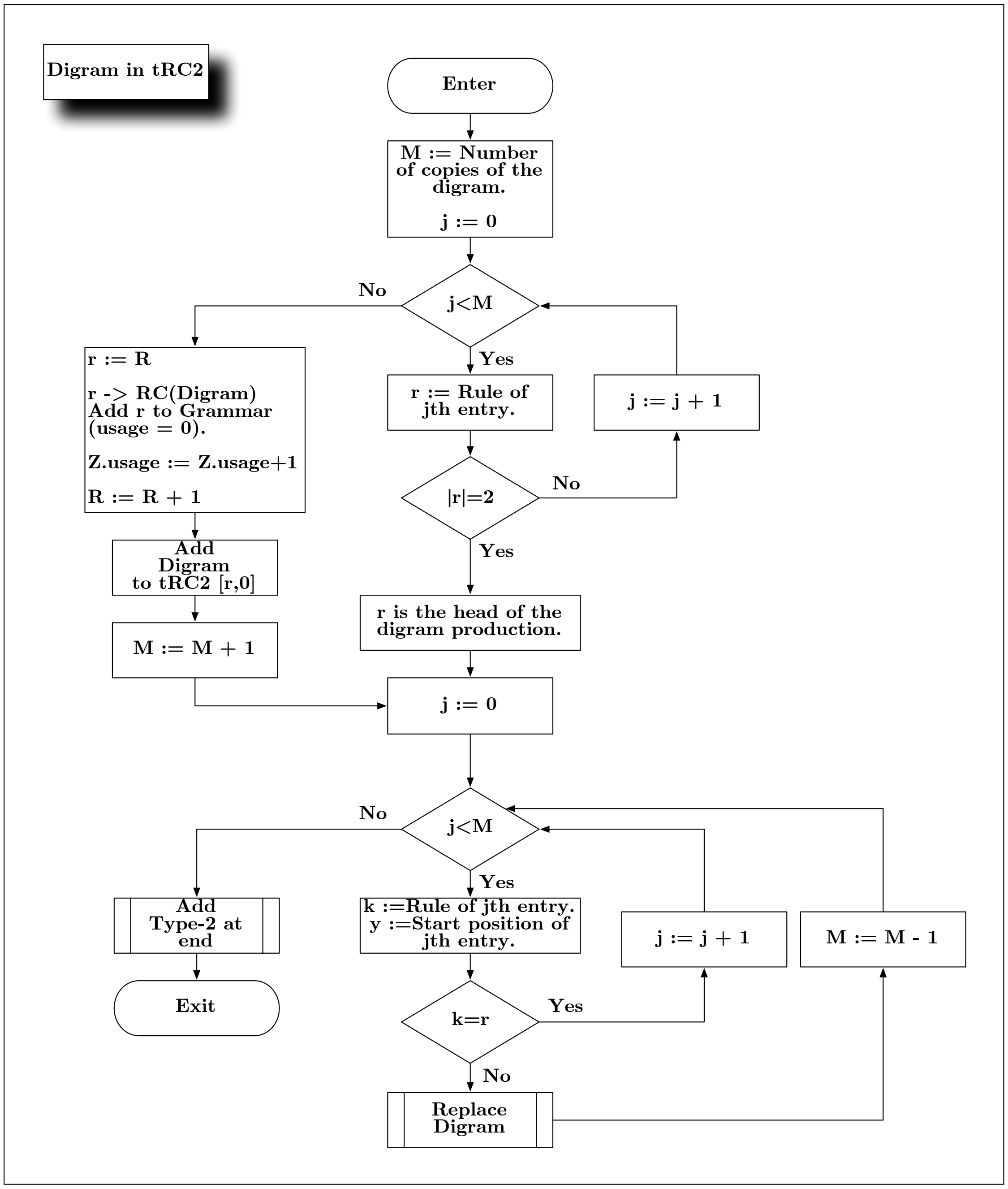


Figure A.4: IVS diagram page 4–digram in reverse complement digram table.

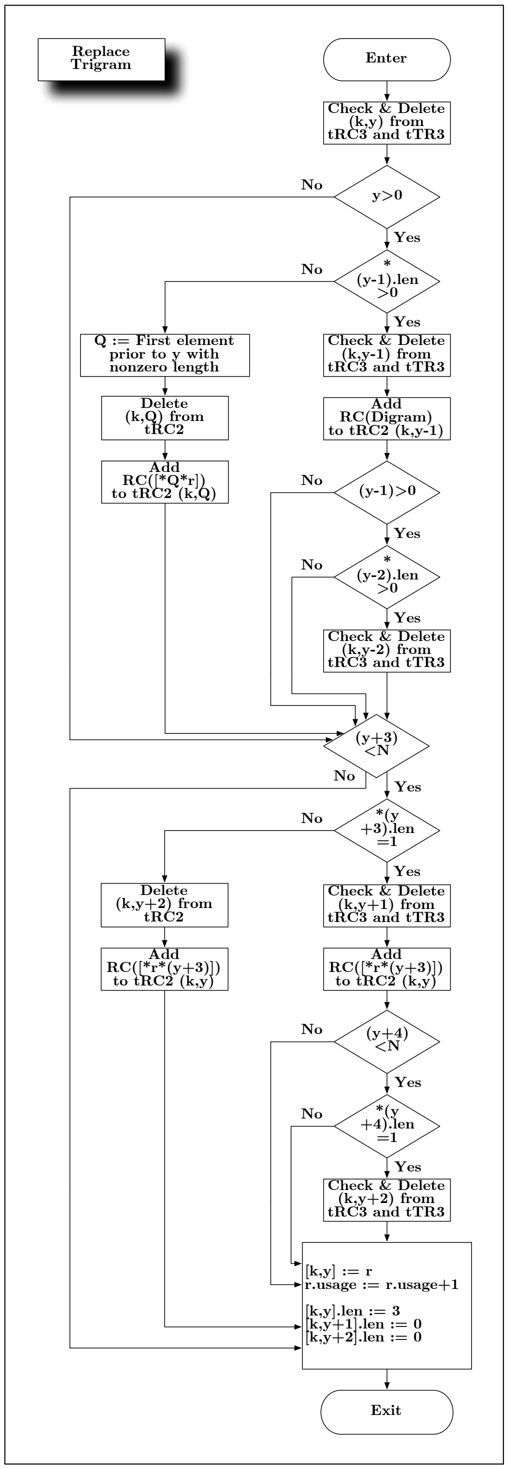

Replace Trigram
Enter
Check & Delete (k,y) from tRC3 and tTR3
y>0
No
Yes
*(y-1).len >0
No
Yes
Q := First element prior to y with nonzero length
Delete (k,Q) from tRC2
Add RC([*Q*r]) to tRC2 (k,Q)
Check & Delete (k,y-1) from tRC3 and tTR3
Add RC(Digram) to tRC2 (k,y-1)
(y-1)>0
No
Yes
*(y-2).len >0
No
Yes
Check & Delete (k,y-2) from tRC3 and tTR3
(y+3) <N
No
Yes
*(y+3).len =1
No
Yes
Delete (k,y+2) from tRC2
Add RC([*r*(y+3)]) to tRC2 (k,y)
Check & Delete (k,y+1) from tRC3 and tTR3
Add RC([*r*(y+3)]) to tRC2 (k,y)
(y+4) <N
No
Yes
*(y+4).len =1
No
Yes
Check & Delete (k,y+2) from tRC3 and tTR3
[k,y] := r
r.usage := r.usage+1
[k,y].len := 3
[k,y+1].len := 0
[k,y+2].len := 0
Exit

Figure A.5: IVS diagram page 5–replace trigram occurrence.

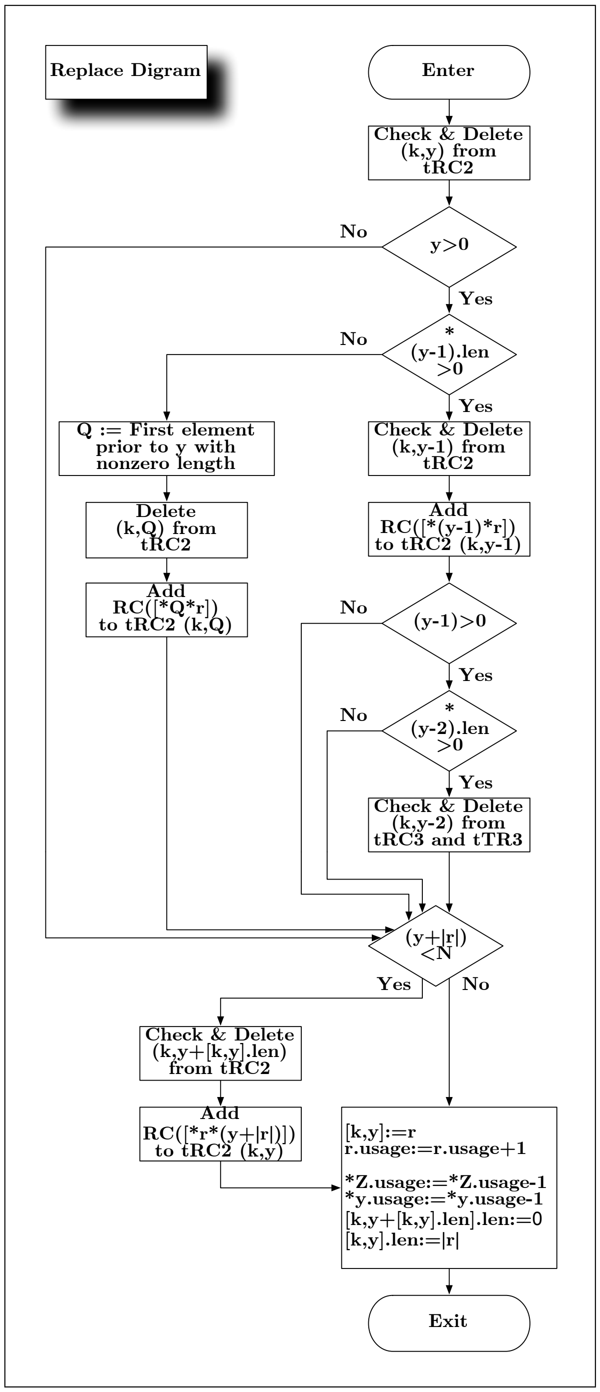

Replace Digram
Enter
Check & Delete (k,y) from tRC2
y>0
No
Yes
* (y-1).len >0
No
Yes
Q := First element prior to y with nonzero length
Delete (k,Q) from tRC2
Add RC([*Q*r]) to tRC2 (k,Q)
Check & Delete (k,y-1) from tRC2
Add RC([*(y-1)*r]) to tRC2 (k,y-1)
(y-1)>0
No
Yes
* (y-2).len >0
No
Yes
Check & Delete (k,y-2) from tRC3 and tTR3
(y+|r|) <N
Yes
No
Check & Delete (k,y+[k,y].len) from tRC2
Add RC([*r*(y+|r|)]) to tRC2 (k,y)
[k,y]:=r
r.usage:=r.usage+1
*Z.usage:=*Z.usage-1
*y.usage:=*y.usage-1
[k,y+[k,y].len].len:=0
[k,y].len:=|r|
Exit

Figure A.6: IVS diagram page 6–replace digram occurrence.

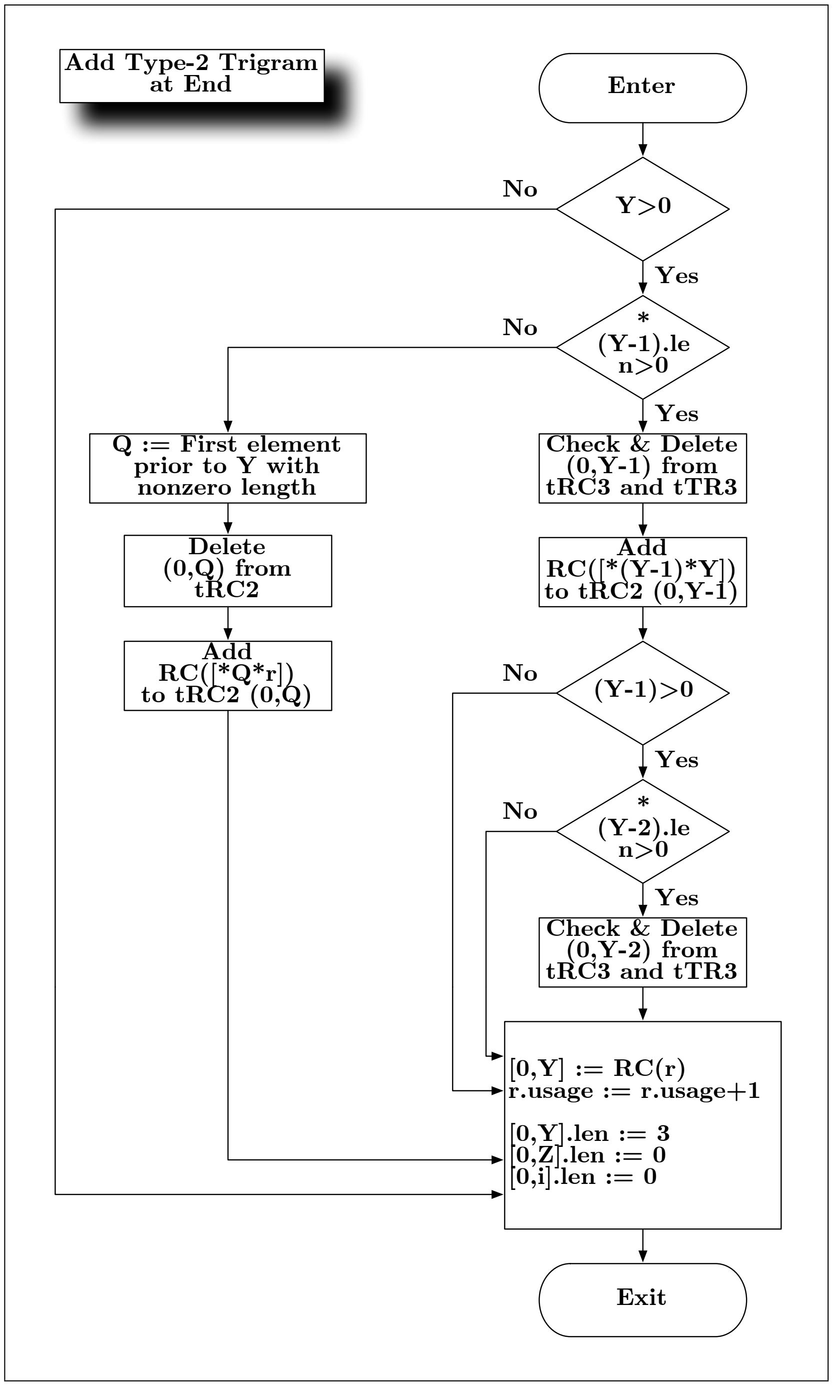

Add Type-2 Trigram at End
Enter
Y>0
No
Yes
*
(Y-1).le
n>0
No
Yes
Q := First element prior to Y with nonzero length
Delete (0,Q) from tRC2
Add RC([*Q*r]) to tRC2 (0,Q)
Check & Delete (0,Y-1) from tRC3 and tTR3
Add RC([*(Y-1)*Y]) to tRC2 (0,Y-1)
(Y-1)>0
No
Yes
*
(Y-2).le
n>0
No
Yes
Check & Delete (0,Y-2) from tRC3 and tTR3
[0,Y] := RC(r)
r.usage := r.usage+1
[0,Y].len := 3
[0,Z].len := 0
[0,i].len := 0
Exit

Figure A.7: IVS diagram page 7–add type-2 trigram rule at the end of the start rule.

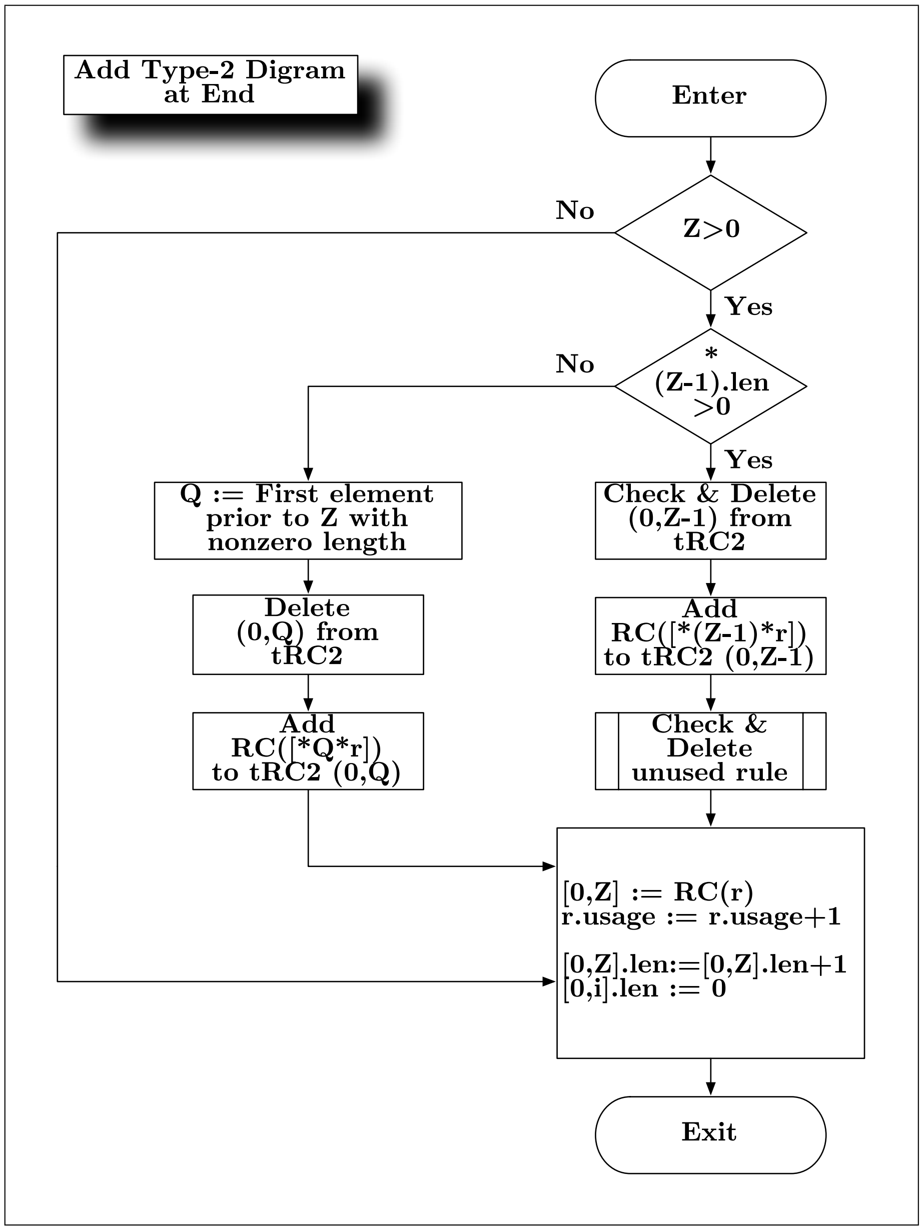

Add Type-2 Digram at End
Enter
Z>0
No
Yes
*(Z-1).len >0
No
Yes
Q := First element prior to Z with nonzero length
Delete (0,Q) from tRC2
Add RC([*Q*r]) to tRC2 (0,Q)
Check & Delete (0,Z-1) from tRC2
Add RC([*(Z-1)*r]) to tRC2 (0,Z-1)
Check & Delete unused rule
[0,Z] := RC(r)
r.usage := r.usage+1
[0,Z].len:=[0,Z].len+1
[0,i].len := 0
Exit

Figure A.8: IVS diagram page 8–add type-2 digram rule at the end of the start rule.

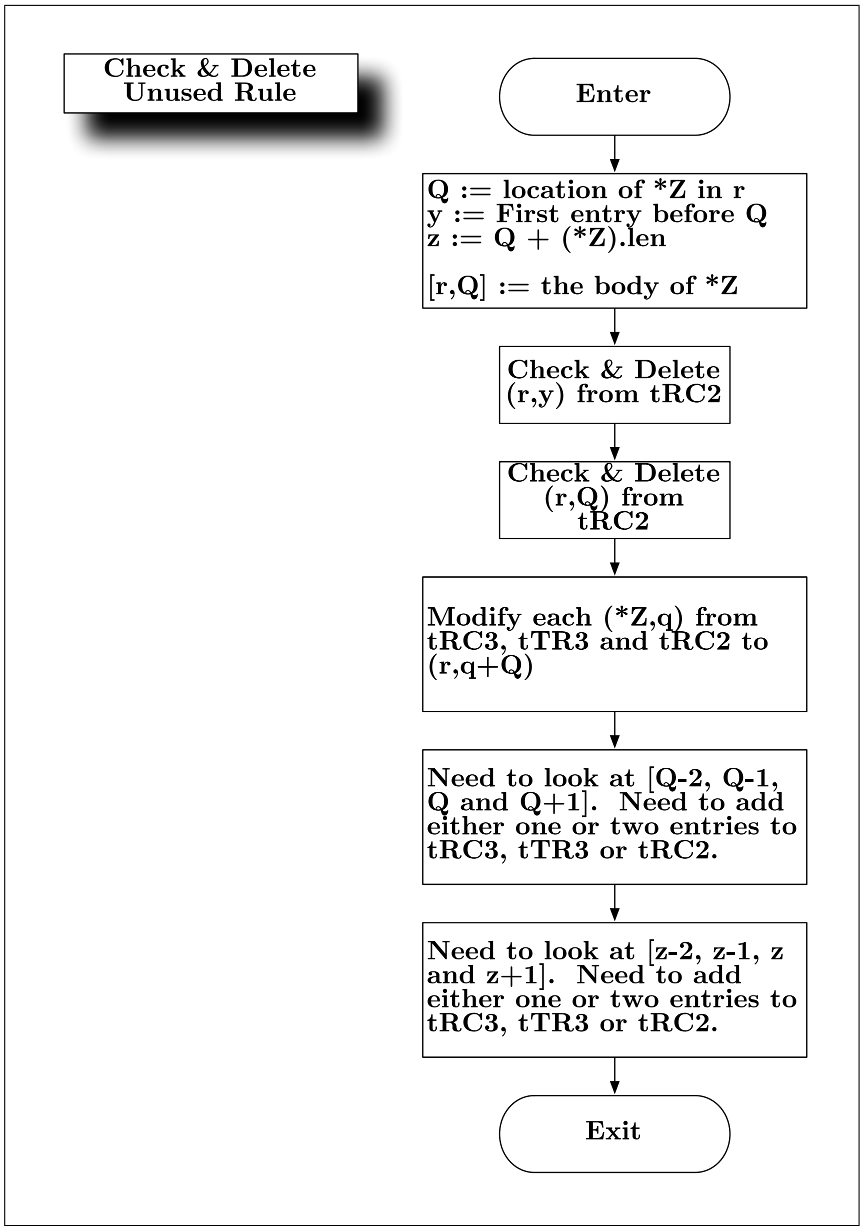

Check & Delete Unused Rule
Enter
Q := location of *Z in r
y := First entry before Q
z := Q + (*Z).len
[r,Q] := the body of *Z
Check & Delete (r,y) from tRC2
Check & Delete (r,Q) from tRC2
Modify each (*Z,q) from tRC3, tTR3 and tRC2 to (r,q+Q)
Need to look at [Q-2, Q-1, Q and Q+1]. Need to add either one or two entries to tRC3, tTR3 or tRC2.
Need to look at [z-2, z-1, z and z+1]. Need to add either one or two entries to tRC3, tTR3 or tRC2.
Exit

Figure A.9: IVS diagram page 9–check and delete unused rule.

# Bibliography

[1] N. Abe and H. Mamitsuka. Predicting Protein Secondary Structure Using Stochastic Tree Grammars. *Machine Learning*, 29:275–301, July 1997.

[2] M. O. Albertson and J. P. Hutchinson. *Discrete Mathematics with Algorithms.* John Wiley & Sons, Inc., New York, 1988.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[5] E. Bareinboim and A. T. R. Vasconcelos. Grammatical Inference Applied to Linguistic Modeling of Biological Regulation Networks. *Electronic Journal of Communication, Information & Innovation in Health*, 1(2):Sup329–Sup333, 2007.

[6] D. R. Bastola, H. H. Otu, S. E. Doukas, K. Sayood, S. H. Hinrichs, and P. C. Iwen. Utilization of the Relative Complexity Measure to Construct a Phylogenetic Tree for Fungi. *Mycological Research*, 108(2):117–125, February 2004.

[7] D. Benedetto, E. Caglioti, and V. Loreto. Language Trees and Zipping. *Physical Review Letters*, 88(4), January 2002.

[8] N. N. Biswas. Maximum Compatible Classes from Compatibility Matrices. *S-adhana*, 14(3):213–218, December 1989.

[9] G. Blackshields, F. Sievers, W. Shi, A. Wilm, and D. G. Higgins. Sequence Embedding for Fast Construction of Guide Trees for Multiple Sequence Alignment. *Algorithms for Molecular Biology*, 5(21), 2010.

[10] V. Brendel and H. G. Busse. Genome Structure Described by Formal Languages. *Nucleic Acids Research*, 12(5):2561–2568, February 1984.

[11] D. G. Brown, M. Li, and B. Ma. A Tutorial of Recent Developments in the Seeding of Local Alignment. *Journal of Bioinformatics and Computational Biology*, 2(4):819–842, 2004.

[12] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the Smallest Grammar: Kolmogorov Complexity in Natural Models. In *STOC '02: Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 792–801, New York, NY, USA, 2002. ACM.

[13] N. Cherniavsky and R. E. Ladner. Grammar-based Compression of DNA Sequences. In *DIMACS Working Group on the Burrows-Wheeler Transform*, August 2004.

[14] D. Chiang, A. K. Joshi, and D. B. Searls. Grammatical Representations of Macromolecular Structure. *Journal of Computational Biology*, 13(5):1077–1100, February 2006.

[15] N. Chomsky. *Logical Structure of Linguistic Theory.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1955.

[16] N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.

[17] P. Clote and R. Backofen. *Computational Molecular Biology, An Introduction.* Cambridge University Press, New York, NY, 1998.

[18] J. Collado-Vides. A Transformational-Grammar Approach to the Study of the Regulation of Gene Expression. *Journal of Theoretical Biology*, 136:403–425, 1989.

[19] E. K. Costello, C. L. Lauber, M. Hamady, N. Fierer, J. I. Gordon, and R. Knight. Bacterial Community Variation in Human Body Habitats Across Space and Time. *Science*, 326:1694–1697, December 2009.

[20] F. H. C. Crick. Codon-Anticodon Pairing: The Wobble Hypothesis. *Journal of Molecular Biology*, 19:548–555, 1966.

[21] M. O. Dayhoff and R. M. Schwartz. A Model of Evolutionary Change in Proteins. In *Atlas of Protein Sequence and Structure*, 1978.

[22] C. B. Do, M. S. P. Mahabhashyam, M. Brudno, and S. Batzoglou. ProbCons: Probabilistic Consistency-Based Multiple Sequence Alignment. *Genome Research*, 15(2):330–340, February 2005.

[23] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis, Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, New York, NY, 1998.

[24] S. R. Eddy. Profile Hidden Markov Models. *Bioinformatics*, 14(9):755–763, 1998.

[25] S. R. Eddy. A Memory-Efficient Dynamic Programming Algorithm for Optimal Alignment of a Sequence to an RNA Secondary Structure. *BMC Bioinformatics*, 3(18), July 2002.

[26] S. R. Eddy. Computational Analysis of RNAs. *Cold Spring Harbor Symposia Quantitative Biology*, 71:117–128, 2006.

[27] R. C. Edgar. MUSCLE: A Multiple Sequence Alignment Method with Reduced Time and Space Complexity. *BMC Bioinformatics*, 5(113), August 2004.

[28] R. C. Edgar. MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput. *Nucleic Acids Research*, 32(5):1792–1797, March 2004.

[29] R. C. Edgar. Search and Clustering Orders of Magnitude Faster than BLAST. *Bioinformatics*, Advance Access published August 12, 2010.

[30] R. C. Edgar and S. Batzoglou. Multiple Sequence Alignment. *Current Opinion in Structural Biology*, 16:368–373, June 2006.

[31] J. Felsenstein. PHYLIP (Phylogeny Inference Package) version 3.6. Distributed by the author, 2005.

[32] G. E. Fox and C. R. Woese. 5S RNA Secondary Structure. *Nature*, 256(5517):505–507, August 1975.

[33] P. P. Gardner, J. Daub, J. G. Tate, E. P. Nawrocki, D. L. Kolbe, S. Lindgreen, A. C. Wilkinson, R. D. Finn, S. Griffiths-Jones, S. R. Eddy, and A. Bateman. Rfam: Updates to the RNA Families Database. *Nucleic Acids Research*, 37(Database issue):D136–D140, 2009.

[34] P. P. Gardner, A. Wilm, and S. Washietl. A Benchmark of Multiple Sequence Alignment Programs upon Structural RNAs. *Nucleic Acids Research*, 33(8):2433–2439, 2005.

[35] M. Gheorghe and V. Mitrana. A Formal Language-Based Approach in Biology. *Comparative and Functional Genomics*, 5:91–94, 2004.

[36] G. H. Gonnet, M. A. Cohen, and S. A. Benner. Exhaustive Matching of the Entire Protein Sequence Database. *Science*, 256(5062):1443–1445, June 1992.

[37] S. Griffiths-Jones, A. Bateman, M. Marshall, A. Khanna, and S. R. Eddy. Rfam: an RNA Family Database. *Nucleic Acids Research*, 31(1):439–441, 2003.

[38] S. Griffiths-Jones, S. Moxon, M. Marshall, A. Khanna, S. R. Eddy, and A. Bateman. Rfam: Annotating Non-Coding RNAs in Complete Genomes. *Nucleic Acids Research*, 33(Database issue):D121–D124, 2005.

[39] V. D. Gusev, L. A. Nemytikova, and N. A. Chuzhanova. On the Complexity Measures of Genetic Sequences. *Bioinformatics*, 15(12):994–999, December 1999.

[40] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On Clustering Validation Techniques. *Journal of Intelligent Information Systems*, 17(2-3):107–145, December 2001.

[41] T. Head. Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[42] S. Henikoff and J. G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, November 1992.

[43] I. L. Hofacker. Vienna RNA Secondary Structure Server. *Nucleic Acids Research*, 31(13):3429–3431, 2003.

[44] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte für Chemie*, 125:167–188, 1994.

[45] L. Holm and C. Sander. Removing Near-Neighbour Redundancy from Large Protein Sequence Collections. *Bioinformatics*, 14(5):423–429, 1998.

[46] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[47] L. Hunter, editor. *Artificial Intelligence and Molecular Biology*, chapter 2, pages 47–120. American Association for Artificial Intelligence, Menlo Park, CA, 1993.

[48] M. Itoh, S. Goto, T. Akutsu, and M. Kanehisa. Fast and Accurate Database Homology Search using Upper Bounds of Local Alignment Scores. *Bioinformatics*, 21(7):912–921, 2005.

[49] D. T. Jones. Protein Secondary Structure Prediction Based on Position-specific Scoring Matrices. *Journal of Molecular Biology*, 292:195–202, 1999.

[50] K. Katoh, K. Kuma, H. Toh, and T. Miyata. MAFFT version 5: Improvement in Accuracy of Multiple Sequence Alignment. *Nucleic Acids Research*, 33(2):511–518, January 2005.

[51] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. MAFFT: A Novel Method for Rapid Multiple Sequence Alignment Based on Fast Fourier Transform. *Nucleic Acids Research*, 30(14):3059–3066, July 2002.

[52] K. Katoh and H. Toh. PartTree: an Algorithm to Build an Approximate Tree from a Large Number of Unaligned Sequences. *Bioinformatics*, 23(3):372–374, 2007.

[53] J. C. Kieffer and E. Yang. Grammar-Based Codes: A New Class of Universal Lossless Source Codes. *IEEE Transactions on Information Theory*, 46(3):737–754, May 2000.

[54] T. Lassmann and E. L. L. Sonnhammer. Kalign - an Accurate and Fast Multiple Sequence Alignment Algorithm. *BMC Bioinformatics*, 6(298), December 2005.

[55] A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, January 1976.

[56] S. Leung, C. Mellish, and D. Robertson. Basic Gene Grammars and DNA-ChartParser for Language Processing of Escherichia coli Promotor DNA Sequences. *Bioinformatics*, 17(3):226–236, 2001.

[57] M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II: Highly Sensitive and Fast Homology Search. *Journal of Bioinformatics and Computational Biology*, 2(3):417–439, 2004.

[58] W. Li. Analysis and Comparison of Very Large Metagenomes with Fast Clustering and Functional Annotation. *BMC Bioinformatics*, 10(359), October 2009.

[59] W. Li and A. Godzik. Cd-hit: a Fast Program for Clustering and Comparing Large Sets of Protein or Nucleotide Sequences. *Bioinformatics*, 22(13):1658–1659, 2006.

[60] W. Li, L. Jaroszewski, and A. Godzik. Clustering of Highly Homologous Sequences to Reduce the Size of Large Protein Databases. *Bioinformatics*, 17(3):282–283, 2001.

[61] W. Li, L. Jaroszewski, and A. Godzik. Tolerating some Redundancy Significantly Speeds up Clustering of Large Protein Databases. *Bioinformatics*, 18(1):77–82, 2002.

[62] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A Tool for Multiple Sequence Alignment. *Proceedings of the National Academy of Sciences of the United States of America*, 86(12):4412–4415, June 1989.

[63] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(3):440–445, 2002.

[64] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[65] J. Meiler and D. Baker. Coupled Prediction of Protein Secondary and Tertiary Structure. *Proceedings of the National Academy of Sciences of the United States of America*, 100(21):12105–12110, October 2003.

[66] J. Meiler, M. Müller, and A. Zeidler. Generation and Evaluation of Dimension-reduced Amino Acid Parameter Representations by Artificial Neural Networks. *Journal of Molecular Modeling*, 7:360–369, 2001.

[67] A. Y. Mitrophanov and M. Borodovsky. Statistical Significance in Biological Sequence Analysis. *Briefings in Bioinformatics*, 7(1):2–24, March 2006.

[68] K. Nakamura and M. Matsumoto. Incremental Learning of Context Free Grammars Based on Bottom-up Parsing and Search. *Pattern Recognition*, 38:1384–1392, 2005.

[69] E. P. Nawrocki and S. R. Eddy. Computational Identification of Functional RNA Homologs in Metagenomic Data. http://selab.janelia.org/publications.html, 2009.

[70] S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[71] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, New Zealand, May 1996.

[72] C. G. Nevill-Manning and I. H. Witten. Compression and Explanation using Hierarchical Grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[73] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, September 1997.

[74] C. Notredame. Recent Progress in Multiple Sequence Alignment: a Survey. *Pharmacogenomics*, 3(1), 2002.

[75] C. Notredame. Recent Evolutions of Multiple Sequence Alignment Algorithms. *PLoS Computational Biology*, 3(8):1405–1408, August 2007.

[76] C. Notredame, D. G. Higgins, and J. Heringa. T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment. *Journal of Molecular Biology*, 302(1):205–217, September 2000.

[77] P. A. Nuin, Z. Wang, and E. R. Tillier. The Accuracy of Several Multiple Sequence Alignment Programs for Proteins. *BMC Bioinformatics*, 7(471), October 2006.

[78] H. H. Otu and K. Sayood. A Divide-and-Conquer Approach to Fragment Assembly. *Bioinformatics*, 19(1):22–29, January 2003.

[79] H. H. Otu and K. Sayood. A New Sequence Distance Measure for Phylogenetic Tree Construction. *Bioinformatics*, 19(16):2122–2130, November 2003.

[80] J. D. Parsons. Improved Tools for DNA Comparison and Clustering. *Computer Applications in the Biosciences*, 11(6):603–613, 1995.

[81] W. R. Pearson. Rapid and Sensitive Sequence Comparison with FASTP and FASTA. *Methods in Enzymology*, 183:63–98, 1990.

[82] T. Przytycka, R. Srinivasan, and G. D. Rose. Recursive Domains in Proteins. *Protein Science*, 11:409–417, November 2002.

[83] A. Puglisi, D. Benedetto, E. Caglioti, V. Loreto, and A. Vulpiani. Data Compression and Learning in Time Sequences Analysis. *Physica D: Nonlinear Phenomena*, 180:92–107, June 2003.

[84] D. A. Rosenblueth, D. Thieffry, A. M. Huerta, H. Salgado, and J. Collado-Vides. Syntactic Recognition of Regulatory Regions in Escherichia coli. *Computer Applications in the Biosciences*, 12(5):415–422, 1996.

[85] D. J. Russell, H. H. Otu, and K. Sayood. Grammar-Based Distance in Progressive Multiple Sequence Alignment. *BMC Bioinformatics*, 9(306), July 2008.

[86] N. Saitou and M. Nei. The Neighbor-joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[87] Y. Sakakibara. Learning Context-Free Grammars using Tabular Representations. *Pattern Recognition*, 38:1372–1383, 2005.

[88] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler. Stochastic Context-Free Grammars for tRNA Modeling. *Nucleic Acids Research*, 22(23):5112–5120, 1994.

[89] D. B. Searls. Investigating the Linguistics of DNA with Definite Clause Grammars. In E. Lusk and R. Overbeek, editors, *Logic Programming: Proceedings North American Conference*, pages 189–208, Cambridge, MA, 1989. MIT Press.

[90] D. B. Searls. The Linguistics of DNA. *American Scientist*, 80:579–591, November-December 1992.

[91] D. B. Searls. The Language of Genes. *Nature*, 420:211–217, November 2002.

[92] V. A. Simossis and J. Heringa. PRALINE: a Multiple Seqeunce Alignment Toolbox that Inegrates Homology-Extended and Secondary Structure Information. *Nucleic Acids Research*, 33(Web Server Issue):W289–W294, July 2005.

[93] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[94] J. Stoye, D.Evers, and F. Meyer. Rose: Generating Sequence Families. *Bioinformatics*, 14(2):157–163, 1998.

[95] A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner, and S. Batzoglou. Whole-Genome Sequencing and Assembly with High-Throughput, Short-Read Technologies. *PLoS ONE*, 2(5), May 2007.

[96] S. Sze, Y. Lu, and Q. Yang. A Polynomial Time Solvable Formulation of Multiple Sequence Alignment. *Journal of Computational Biology*, 13(2):309–319, 2006.

[97] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Research*, 22(22):4673–4680, November 1994.

[98] J. D. Thompson, F. Plewniak, and O. Poch. A Comprehensive Comparison of Multiple Sequence Alignment Programs. *Nucleic Acids Research*, 27(13):2682–2690, July 1999.

[99] J. D. Thompson, F. Plewniak, and O. Poch. BAliBASE: a Benchmark Alignment Database for the Evaluation of Multiple Alignment Programs. *Bioinformatics*, 15(1):87–88, 1999.

[100] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, September 1995.

[101] G. Varani and W. H. McClain. The GxU Wobble Base Pair. A Fundamental Building Block of RNA Structure Crucial to RNA Function in Diverse Biological Systems. *EMBO Reports*, 1(1):18–23, 2000.

[102] P. Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, October 1973.

[103] W. J. Wilbur and D. J. Lipman. Rapid Similarity Searches of Nucleic Acid and Protein Data Banks. *Proceedings of the National Academy of Sciences of the United States of America*, 80:726–730, February 1983.

[104] A. Wilm, I. Mainz, and G. Steger. An Enhanced RNA Alignment Benchmark for Sequence Alignment Programs. *Algorithms for Molecular Biology*, 1(19), October 2006.

[105] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[106] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

[107] M. Zuker. Mfold Web Server for Nucleic Acid Folding and Hybridization Prediction. *Nucleic Acids Research*, 31(13):3406–3415, 2003.

[108] M. Zuker and P. Stiegler. Optimal Computer Folding of Large RNA Sequences using Thermodynamics and Auxiliary Information. *Nucleic Acids Research*, 9(1):133–148, 1981.