Lecture Notes on
Biological Sequence Analysis [1]

Martin Tompa

Technical Report #2000-06-01

Winter 2000

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, Washington, U.S.A. 98195-2350

© Martin Tompa, 2000

[1]This material is based upon work supported in part by the National Science Foundation and DARPA under grant DBI-9601046, and by the National Science Foundation under grant DBI-9974498.

Contents

Preface

These are the lecture notes from CSE 527, a graduate course on computational molecular biology I taught at the University of Washington in Winter 2000. The topic of the course was Biological Sequence Analysis. These notes are not intended to be a survey of that area, however, as there are numerous important results that I would have liked to cover but did not have time.

I am grateful to Phil Green, Dick Karp, Rune Lyngsø, Larry Ruzzo, and Rimli Sengupta, who helped me both with overview and with technical points. I am thankful for the students who attended faithfully, served as notetakers, asked embarassing questions, made perceptive comments, carried out exciting projects, and generally make teaching exciting and rewarding.

— Martin Tompa

Lecture 1

Basics of Molecular Biology

January 4, 2000
Notes: Michael Gates

We begin with a review of the basic molecules responsible for the functioning of all organisms' cells. Much of the material here comes from the introductory textbooks by Drlica [14], Lewin [31], and Watson *et al.* [52]. Later in the course, when we discuss the computational aspects of molecular biology, some useful textbooks will be those by Gusfield [20], Salzberg *et al.* [42], Setubal and Meidanis [43], and Waterman [51].

What sorts of molecules perform the required functions of the cells of organisms? Cells have a basic tension in the roles they need those molecules to fulfill:

1. The molecules must perform the wide variety of chemical reactions necessary for life. To perform these reactions, cells need diverse three-dimensional structures of interacting molecules.
2. The molecules must pass on the instructions for creating their constitutive components to their descendents. For this purpose, a simple one-dimensional information storage medium is the most effective.

We will see that *proteins* provide the three-dimensional diversity required by the first role, and *DNA* provides the one-dimensional information storage required by the second. Another cellular molecule, *RNA*, is an intermediary between DNA and proteins, and plays some of each of these two roles.

1.1. Proteins

Proteins have a variety of roles that they must fulfill:

1. They are the enzymes that rearrange chemical bonds.
2. They carry signals to and from the outside of the cell, and within the cell.
3. They transport small molecules.
4. They form many of the cellular structures.
5. They regulate cell processes, turning them on and off and controlling their rates.

This variety of roles is accomplished by the variety of proteins, which collectively can assume a variety of three-dimensional shapes.

A protein's three-dimensional shape, in turn, is determined by the particular one-dimensional composition of the protein. Each protein is a linear sequence made of smaller constituent molecules called *amino acids*. The constituent amino acids are joined by a "backbone" composed of a regularly repeating sequence of bonds. (See [31, Figure 1.4].) There is an asymmetric orientation to this backbone imposed by its chemical structure: one end is called the *N-terminus* and the other end the *C-terminus*. This orientation imposes directionality on the amino acid sequence.

There are 20 different types of amino acids. The three-dimensional shape the protein assumes is determined by the specific linear sequence of amino acids from N-terminus to C-terminus. Different sequences of amino acids *fold* into different three-dimensional shapes. (See, for example, [10, Figure 1.1].)

Protein size is usually measured in terms of the number of amino acids that comprise it. Proteins can range from fewer than 20 to more than 5000 amino acids in length, although an average protein is about 350 amino acids in length.

Each protein that an organism can produce is encoded in a piece of the DNA called a "gene" (see Section 1.6). To give an idea of the variety of proteins one organism can produce, the single-celled bacterium *E. coli* has about 4300 different genes. Humans are believed to have about 50,000 different genes (the exact number as yet unresolved), so a human has only about 10 times as many genes as *E. coli*. The number of proteins that can be produced by humans greatly exceeds the number of genes, however, because a substantial fraction of the human genes can each produce many different proteins through a process called "alternative splicing".

1.1.1. Classification of the Amino Acids

Each of the 20 amino acids consists of two parts:

1. a part that is identical among all 20 amino acids; this part is used to link one amino acid to another to form the backbone of the protein.
2. a unique *side chain* (or "R group") that determines the distinctive physical and chemical properties of the amino acid.

Although each of the 20 different amino acids has unique properties, they can be classified into four categories based upon their major chemical properties. Below are the names of the amino acids, their 3 letter abbreviations, and their standard one letter symbols.

1. Positively charged (and therefore basic) amino acids (3).

Arginine	Arg	R
Histidine	His	H
Lysine	Lys	K

2. Negatively charged (and therefore acidic) amino acids (2).

Aspartic acid	Asp	D
Glutamic acid	Glu	E

3. Polar amino acids (7). Though uncharged overall, these amino acids have an uneven charge distribution. Because of this uneven charge distribution, these amino acids can form hydrogen bonds with water. As a consequence, polar amino acids are called *hydrophilic*, and are often found on the outer surface of folded proteins, in contact with the watery environment of the cell.

Asparagine	Asn	N
Cysteine	Cys	C
Glutamine	Gln	Q
Glycine	Gly	G
Serine	Ser	S
Threonine	Thr	T
Tyrosine	Tyr	Y

4. Nonpolar amino acids (8). These amino acids are uncharged and have a uniform charge distribution. Because of this, they do not form hydrogen bonds with water, are called *hydrophobic*, and tend to be found on the inside surface of folded proteins.

Alanine	Ala	A
Isoleucine	Ile	I
Leucine	Leu	L
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Tryptophan	Trp	W
Valine	Val	V

Although each amino acid is different and has unique properties, certain pairs have more similar properties than others. The two nonpolar amino acids leucine and isoleucine, for example, are far more similar to each other in their chemical and physical properties than either is to the charged glutamic acid. In algorithms for comparing proteins to be discussed later, the question of amino acid similarity will be important.

1.2. DNA

DNA contains the instructions needed by the cell to carry out its functions. DNA consists of two long interwoven strands that form the famous "double helix". (See [14, Figure 3-3].) Each strand is built from a small set of constituent molecules called *nucleotides*.

1.2.1. Structure of a Nucleotide

A nucleotide consist of three parts [14, Figure 3-2]. The first two parts are used to form the ribbon-like backbone of the DNA strand, and are identical in all nucleotides. These two parts are (1) a *phosphate group* and (2) a sugar called *deoxyribose* (from which DNA, DeoxyriboNucleic Acid, gets its name). The third part of the nucleotide is the *base*. There are four different bases, which define the four different nucleotides: thymine (T), cytosine (C), adenine (A), and guanine (G).

Note in [14, Figure 3-2] that the five carbon atoms of the sugar molecule are numbered $C_{1'}, C_{2'}, C_{3'}, C_{4'}, C_{5'}$. The base is attached to the $1'$ carbon. The two neighboring phosphate groups are attached to the $5'$ and $3'$ carbons. As is the case in the protein backbone (Section 1.1), the asymmetry of the sugar molecule imposes an orientation on the backbone, one end of which is called the *$5'$ end* and the other the *$3'$ end*. (See [14, Figure 3-4(a)].)

1.2.2. Base Pair Complementarity

Why is DNA double-stranded? This is due to *base pair complementarity*. If specific bases of one strand are aligned with specific bases on the other strand, the aligned bases can *hybridize* via hydrogen bonds, weak attractive forces between hydrogen and either nitrogen or oxygen. The specific complementary pairs are

- A with T
- G with C

Two hydrogen bonds form between A and T, whereas three form between C and G. (See [14, Figure 3-5].) This makes C-G bonds stronger than A-T bonds.

If two DNA strands consist of complementary bases, under "normal" cellular conditions they will hybridize and form a stable double helix. However, the two strands will only hybridize if they are in "antiparallel configuration". This means that the sequence of one strand, when read from the $5'$ end to the $3'$ end, must be complementary, base for base, to the sequence of the other strand read from $3'$ to $5'$. (See [14, Figure 3-4(b) and 3-3].)

1.2.3. Size of DNA molecules

An *E. coli* bacterium contains one circular, double-stranded molecule of DNA consisting of approximately 5 million nucleotides. Often the length of double-stranded DNA is expressed in the units of basepairs (bp), kilobasepairs (kb), or megabasepairs (Mb), so that this size could be expressed equivalently as 5×10^6 bp, 5000 kb, or 5 Mb.

Each human cell contains 23 pairs of *chromosomes*, each of which is a long, double-stranded DNA molecule. Collectively, the 46 chromosomes in one human cell consist of approximately 3×10^9 bp of DNA. Note that a human has about 1000 times more DNA than *E. coli* does, yet only about 10 times as many genes. (See Section 1.1.) The reason for this will be explained shortly.

1.3. RNA

Chemically, RNA is very similar to DNA. There are two main differences:

1. RNA uses the sugar *ribose* instead of deoxyribose in its backbone (from which RNA, RiboNucleic Acid, gets its name).
2. RNA uses the base uracil (U) instead of thymine (T). U is chemically similar to T, and in particular is also complementary to A.

RNA has two properties important for our purposes. First, it tends to be single-stranded in its "normal" cellular state. Second, because RNA (like DNA) has base-pairing capability, it often forms intramolecular hydrogen bonds, partially hybridizing to itself. Because of this, RNA, like proteins, can fold into complex three-dimensional shapes. (For an example, see `http://www.ibc.wustl.edu/~zuker/rna/hammerhead.html`.)

RNA has some of the properties of both DNA and proteins. It has the same information storage capability as DNA due to its sequence of nucleotides. But its ability to form three-dimensional structures allows it to have enzymatic properties like those of proteins. Because of this dual functionality of RNA, it has been conjectured that life may have originated from RNA alone, DNA and proteins having evolved later.

1.4. Residues

The term *residue* refers to either a single base constituent from a nucleotide sequence, or a single amino acid constituent from a protein. This is a useful term when one wants to speak collectively about these two types of biological sequences.

1.5. DNA Replication

What is the purpose of double-strandedness in DNA? One answer is that this redundancy of information is key to how the one-dimensional instructions of the cell are passed on to its descendant cells. During the cell cycle, the DNA double strand is split into its two separate strands. As it is split, each individual strand is used as a template to synthesize its complementary strand, to which it hybridizes. (See [14, Figure 5-2 and 5-1].) The result is two exact copies of the original double-stranded DNA.

In more detail, an enzymatic protein called *DNA polymerase* splits the DNA double strand and synthesizes the complementary strand of DNA. It synthesizes this complementary strand by adding *free nucleotides* available in the cell onto the $3'$ end of the new strand being synthesized [14, Figure 5-3]. The DNA polymerase will only add a nucleotide if it is complementary to the opposing base on the template strand. Because the DNA polymerase can only add new nucleotides to the $3'$ end of a DNA strand (i.e., it can only synthesize DNA in the $5'$ to $3'$ direction), the actual mechanism of copying both strands is somewhat more complicated. One strand can be synthesized continuously in the $5'$ to $3'$ direction. The other strand must be synthesized in short $5'$-to-$3'$ fragments. Another enzymatic protein, *DNA ligase*, glues these synthesized fragments together into a single long DNA molecule. (See [14, Figure 5-4].)

1.6. Synthesis of RNA and Proteins

The one-dimensional storage of DNA contains the information needed by the cell to produce all its RNA and proteins. In this section, we describe how the information is encoded, and how these molecules are synthesized.

Proteins are synthesized in a two-step process. First, an RNA "copy" of a portion of the DNA is synthesized in a process called *transcription*, described in Section 1.6.1. Second, this RNA sequence is read and interpreted to synthesize a protein in a process called *translation*, described in Section 1.6.2. Together, these two steps are called *gene expression*.

A *gene* is a sequence of DNA that encodes a protein or an RNA molecule. Gene structure and the exact expression process are somewhat dependent on the organism in question. The *prokaryotes*, which consist of the *bacteria* and the *archaea*, are single-celled organisms lacking nuclei. Because prokaryotes have the simplest gene structure and gene expression process, we will start with them. The *eukaryotes*, which include plants and animals, have a somewhat more complex gene structure that we will discuss after.

1.6.1. Transcription in Prokaryotes

How do prokaryotes synthesize RNA from DNA? This process, called transcription, is similar to the way DNA is replicated (Section 1.5). An enzyme called *RNA polymerase*, copies one strand of the DNA gene into a *messenger RNA* (*mRNA*), sometimes called the *transcript*. The RNA polymerase temporarily splits the double-stranded DNA, and uses one strand as a template to build the complementary strand of RNA. (See [14, Figure 4-1].) It incorporates U opposite A, A opposite T, G opposite C, and C opposite G. The RNA polymerase begins this transcription at a short DNA pattern it recognizes called the *transcription start site*. When the polymerase reaches another DNA sequence called the *transcription stop site*, signalling the end of the gene, it drops off.

1.6.2. Translation

How is protein synthesized from mRNA? This process, called translation, is not as simple as transcription, because it proceeds from a 4 letter alphabet to the 20 letter alphabet of proteins. Because there is not a one-to-one correspondence between the two alphabets, amino acids are encoded by consecutive sequences of 3 nucleotides, called *codons*. (Taking 2 nucleotides at a time would give only $4^2 = 16$ possible permutations, whereas taking 3 nucleotides yields $4^3 = 64$ possible permutations, more than sufficient to encode the 20 different amino acids.) The decoding table is given in Table 1.1, and is called the *genetic code*. It is rather amazing that this same code is used almost universally by all organisms.

	U	C	A	G	
U	UUU Phe [F]	UCU Ser [S]	UAU Tyr [Y]	UGU Cys [C]	U
	UUC Phe [F]	UCC Ser [S]	UAC Tyr [Y]	UGC Cys [C]	C
	UUA Leu [L]	UCA Ser [S]	UAA *STOP*	UGA *STOP*	A
	UUG Leu [L]	UCG Ser [S]	UAG *STOP*	UGG Trp [W]	G
C	CUU Leu [L]	CCU Pro [P]	CAU His [H]	CGU Arg [R]	U
	CUC Leu [L]	CCC Pro [P]	CAC His [H]	CGC Arg [R]	C
	CUA Leu [L]	CCA Pro [P]	CAA Gln [Q]	CGA Arg [R]	A
	CUG Leu [L]	CCG Pro [P]	CAG Gln [Q]	CGG Arg [R]	G
A	AUU Ile [I]	ACU Thr [T]	AAU Asn [N]	AGU Ser [S]	U
	AUC Ile [I]	ACC Thr [T]	AAC Asn [N]	AGC Ser [S]	C
	AUA Ile [I]	ACA Thr [T]	AAA Lys [K]	AGA Arg [R]	A
	AUG Met [M]	ACG Thr [T]	AAG Lys [K]	AGG Arg [R]	G
G	GUU Val [V]	GCU Ala [A]	GAU Asp [D]	GGU Gly [G]	U
	GUC Val [V]	GCC Ala [A]	GAC Asp [D]	GGC Gly [G]	C
	GUA Val [V]	GCA Ala [A]	GAA Glu [E]	GGA Gly [G]	A
	GUG Val [V]	GCG Ala [A]	GAG Glu [E]	GGG Gly [G]	G

Table 1.1: The Genetic Code

There is a necessary redundancy in the code, since there are 64 possible codons and only 20 amino acids. Thus each amino acid (with the exceptions of Met and Trp) is encoded by *synonymous codons*, which are interchangeable in the sense of producing the same amino acid. Only 61 of the 64 codons are used to encode amino acids. The remaining 3, called *STOP codons*, signify the end of the protein.

Ribosomes are the molecular structures that read mRNA and produce the encoded protein according to the genetic code. Ribosomes are large complexes consisting of both proteins and a type of RNA called *ribosomal RNA* (*rRNA*).

The process by which ribosomes translate mRNA into protein makes use of yet a third type of RNA called *transfer RNA* (*tRNA*). There are 61 different transfer RNAs, one for each nontermination codon. Each tRNA folds (see Section 1.3) to form a cloverleaf-shaped structure. This structure produces a pocket that complexes uniquely with the amino acid encoded by the tRNA's associated codon, according to Table 1.1. The unique fit is accomplished analogously to a key and lock mechanism. Elsewhere on the tRNA is the *anticodon*, three consecutive bases that are complementary and antiparallel to the associated codon, and exposed for use by the ribosome. The ribosome brings together each codon of the mRNA with its corresponding anticodon on some tRNA, and hence its encoded amino acid. (See [14, Figure 4-4].)

Lecture 2

Basics of Molecular Biology (continued)

January 6, 2000
Notes: Tory McGrath

2.1. Course Projects

A typical course project might be to take some existing biological sequences from the public databases on the web, and design and run some sequence analysis experiments, using either publicly available software or your own program. For example, there is reason to believe that some of the existing bacterial genomes may be misannotated, in the sense that the identified genes are not actually located exactly as annotated. There is existing software to identify gene locations. We will discuss more such suggested projects as the course proceeds, but the choice of topic is quite flexible, and is open to suggestion, provided there is a large computational aspect. You will be required to check your project topic with the instructor before embarking.

You may work on the project in groups of up to four people. For maximum effectiveness, it is recommended to have a mix of biology and math/computer participants in each group.

The project will entail a short write-up as well as a short presentation of your problem, methods, and results.

2.2. Translation (continued)

In prokaryotes, which have no cell nucleus, translation begins while transcription is still in progress, the $5'$ end of the transcript being translated before the RNA polymerase has transcribed the $3'$ end. (See Drlica [14, Figure 4-4].) In eukaryotes, the DNA is inside the nucleus, whereas the ribosomes are in the *cytoplasm* outside the nucleus. Hence, transcription takes place in the nucleus, the completed transcript is exported from the nucleus, and translation then takes place in the cytoplasm.

The ribosome forms a complex near the $5'$ end of the mRNA, binding around the *start codon*, also called the *translation start site*. The start codon is most often $5'$-AUG-$3'$, and the corresponding anticodon is $5'$-CAU-$3'$. (Less often, the start codon is $5'$-GUG-$3'$or $5'$-UUG-$3'$.) The ribosome now brings together this start codon on the mRNA and its exposed anticodon on the corresponding tRNA, which hybridize to each other. (See [14, Figure 4-4].) The tRNA brings with it the encoded amino acid; in the case of the usual start codon $5'$-AUG-$3'$, this is methionine.

Having incorporated the first amino acid of the synthesized protein, the ribosome shifts the mRNA three bases to the next codon. A second tRNA complexed with its specific amino acid hybridizes to the second codon via its anticodon, and the ribosome bonds this second amino acid to the first. At this point the ribosome releases the first tRNA, moves on to the third codon, and repeats. (See [14, Figure 4-5].) This process continues until the ribosome detects one of the STOP codons, at which point it releases the mRNA and the completed protein.

2.3. Prokaryotic Gene Structure

Recall from Section 1.6 that a gene is a relatively short sequence of DNA that encodes a protein or RNA molecule. In this section we restrict our attention to protein-coding genes in prokaryotes.

The portion of the gene containing the codons that ultimately will be translated into the protein is called the *coding region*, or *open reading frame*. The transcription start site (see Section 1.6.1) is somewhat *upstream* from the start codon, where "upstream" means "in the $5'$ direction". Similarly, the transcription stop site is somewhat *downstream* from the stop codon, where "downstream" means "in the $3'$ direction". That is, the mRNA transcript contains sequence at both its ends that has been transcribed, but will not be translated. The sequence between the transcription start site and the start codon is called the *$5'$ untranslated region*. The sequence between the stop codon and the transcription stop site is called the *$3'$ untranslated region*.

Upstream from the transcription start site is a relatively short sequence of DNA called the *regulatory region*. It contains *promoters*, which are specific DNA sites where certain regulatory proteins bind and regulate expression of the gene. These proteins are called *transcription factors*, since they regulate the transcription process. A common way in which transcription factors regulate expression is to bind to the DNA at a promoter and from there affect the ability (either positively or negatively) of RNA polymerase to perform its task of transcription. (There is also the analogous possibility of *translational regulation*, in which regulatory factors bind to the mRNA and affect the ability of the ribosome to perform its task of translation.)

2.4. Prokaryotic Genome Organization

The *genome* of an organism is the entire complement of DNA in any of its cells. In prokaryotes, the genome typically consists of a single chromosome of double-stranded DNA, and it is often circularized (its $5'$ and $3'$ ends attached) as opposed to being linear. A typical prokaryotic genome size would be in the millions of base pairs.

Typically 90% of the prokaryotic genome consists of coding regions. For instance, the *E. coli* genome has size about 5 Mb and approximately 4300 coding regions, each of average length around 1000 bp. The genes are relatively densely and uniformly distributed throughout the genome.

2.5. Eukaryotic Gene Structure

An important difference between prokaryotic and eukaryotic genes is that the latter may contain "introns". In more detail, the transcribed sequence of a general eukaryotic gene is an alternation between DNA sequences called *exons* and *introns*, where the introns are sequences that ultimately will be spliced out of the mRNA before it leaves the nucleus. Transcription in the nucleus produces an RNA molecule called *pre-mRNA*, produced as described in Section 1.6.1, that contains both the exons and introns. The introns are spliced out of the pre-mRNA by structures called *spliceosomes* to produce the *mature mRNA* that will be transported out of the nucleus for translation. A eukaryotic gene may contain numerous introns, and each intron may be many kilobases in size. One fact that is relevant to our later computational studies is that the presence of introns makes it much more difficult to identify the locations of genes computationally, given the genome sequence.

Another important difference between prokaryotic and higher eukaryotic genes is that, in the latter, there can be multiple regulatory regions that can be quite far from the coding region, can be either upstream or downstream from it, and can even be in the introns.

2.6. Eukaryotic Genome Organization

Unlike prokaryotic genomes, many eukaryotic genomes consist of multiple linear chromosomes as opposed to single circular chromosomes. Depending on how simple the eukaryote is, very little of the genome may be coding sequence. In humans, less than 3% of the genome is believed to be coding sequence, and the genes are distributed quite nonuniformly over the genome.

2.7. Goals and Status of Genome Projects

Molecular biology has the following two broad goals:

1. Identify all key molecules of a given organism, particularly the proteins, since they are responsible for the chemical reactions of the cells.
2. Identify all key interactions among molecules.

Traditionally, molecular biologists have tackled these two goals simultaneously in selected small systems within selected model organisms. The genome projects today differ by focusing primarily on the first goal, but for *all* the systems of a given model organism. They do this by *sequencing* the genome, which means determining the entire DNA sequence of the organism. They then perform a computational analysis (to be discussed in later lectures) on the genome sequence to identify (most of) the genes. Having done this, (most of) the proteins of the organism will have been identified.

With recent advances in sequencing technology, the genome projects have progressed very rapidly over the past five years. The first free-living organism to be completely sequenced was the bacterium *H. influenzae* [15], with a genome of size 1.8 Mb. Since that time, 18 bacterial, 6 archaeal, and 2 eukaryotic genomes have been sequenced. Presently there are approximately an additional 95 prokaryotic and 27 eukaryotic genomes in the process of being sequenced. (See, for example, the Genomes On Line Database at `http://geta.life.uiuc.edu/~nikos/genomes.html` for the status of ongoing genome projects.)

The human genome is expected to be sequenced within the next two years or so. Although every human is a unique individual, the genome sequences of any two humans are about 99.9% identical, so that it makes some sense to talk about sequencing *the* human genome, which will really be an amalgamation of a small collection of individuals. Once that is done, one of the interesting challenges is to identify the common *polymorphisms*, which are genomic variations that occur in a nonnegligible fraction of the population.

2.8. Sequence Analysis

Once a genome is completely sequenced, what sorts of analyses are performed on it? Some of the goals of *sequence analysis* are the following:

1. Identify the genes.
2. Determine the function of each gene. One way to hypothesize the function is to find another gene (possibly from another organism) whose function is known and to which the new gene has high sequence similarity. This assumes that sequence similarity implies functional similarity, which may or may not be true.
3. Identify the proteins involved in the regulation of gene expression.
4. Identify sequence repeats.
5. Identify other functional regions, for example *origins of replication* (sites at which DNA polymerase binds and begins replication; see Section 1.5), *pseudogenes* (sequences that look like genes but are not expressed), sequences responsible for the compact folding of DNA, and sequences responsible for nuclear anchoring of the DNA.

Many of these tasks are computational in nature. Given the incredible rate at which sequence data is being produced, the integration of computer science, mathematics, and biology will be integral to analyzing those sequences.

Lecture 3

Introduction to Sequence Similarity

January 11, 2000
Notes: Martin Tompa

3.1. Sequence Similarity

The next few lectures will deal with the topic of "sequence similarity", where the sequences under consideration might be DNA, RNA, or amino acid sequences. This is likely the most frequently performed task in computational biology. Its usefulness is predicated on the assumption that a high degree of similarity between two sequences often implies similar function and/or three-dimensional structure. Most of the content of these lectures on sequence similarity is from Gusfield [20].

Why are we starting here, rather than with a discussion of how biologists determine the sequence in the first place? The reason is that the problems and algorithms of sequence similarity are reasonably simple to state. This makes it a good context in which to ensure that we agree on the language we will be using to discuss computing and algorithms.

To begin that process, the word *algorithm* simply means an unambiguously specified method for solving a problem. In this context, an algorithm may be thought of as a computer program, although algorithms are usually expressed in a somewhat more abstract language than real programming languages.

3.2. Biological Motivation for Studying Sequence Similarity

We start with two motivating applications in which sequence similarity is utilized.

3.2.1. Hypothesizing the Function of a New Sequence

When a new genome is sequenced, the usual first analysis performed is to identify the genes and hypothesize their functions. Hypothesizing their functions is most often done using sequence similarity algorithms, as follows. One first translates the coding regions into their corresponding amino acid sequences, using the genetic code of Table 1.1. One then searches for similar sequences in a protein database that contains sequenced proteins (from related organisms) and their functions. Close matches allow one to make strong conjectures about the function of each matched gene. In a similar way, sequence similarity can be used to predict the three-dimensional structure of a newly sequenced protein, given a database of known protein sequences and their structures.

3.2.2. Researching the Effects of Multiple Sclerosis

Multiple sclerosis is an autoimmune disease in which the immune system attacks nerve cells in the patient. More specifically, the immune system's T-cells, which normally identify foreign bodies for immune system attacks, mistakenly identify proteins in the nerves' myelin sheaths as foreign.

It was conjectured that the myelin sheath proteins identified by the T-cells were similar to viral and/or bacterial sheath proteins from an earlier infection. In order to test this hypothesis, the following steps were carried out:

- the myelin sheath proteins were sequenced,
- a protein database was searched for similar bacterial and viral sequences, and
- laboratory tests were performed to determine if the T-cells attacked these same proteins.

The result was the identification of certain bacterial and viral proteins that were confused with the myelin sheath proteins.

3.3. The String Alignment Problem

The first task is to make the problem of sequence similarity more precise. A *string* is a sequence of characters from some alphabet. Given two strings `acbcdb` and `cadbd`, how should we measure how similar they are? Similarity is witnessed by finding a good "alignment" between two strings. Here is one possible alignment of these two strings.

```
a c - - b c d b
- c a d b - d -
```

The special character "$-$" represents the insertion of a *space*, representing a deletion from its sequence (or, equivalently, an insertion in the other sequence). We can evaluate the goodness of such an alignment using a scoring function. For example, if an exact match between two characters scores $+2$, and every mismatch or deletion (space) scores -1, then the alignment above has score

$$3 \cdot (2) + 5 \cdot (-1) = 1.$$

This example shows only one possible alignment for the given strings. For any pair of strings, there are many possible alignments.

The following definitions generalize this example.

Definition 3.1: If x and y are each a single character or space, then $\sigma(x, y)$ denotes the *score* of aligning x and y. σ is called the *scoring function*.

In the example above, for any two distinct characters a and c, $\sigma(c, c) = +2$, and $\sigma(c, a) = \sigma(c, -) = \sigma(-, c) = -1$. If one were designing a scoring function for comparing amino acid sequences, one would certainly want to incorporate into it the physico-chemical similarities and differences among the amino acids, such as those described in Section 1.1.1.

Definition 3.2: If S is a string, then $|S|$ denotes the length of S and $S[i]$ denotes the ith character of S (where the first character is $S[1]$ rather than, say, $S[0]$).

For example, if $S = \texttt{acbcdb}$, then $|S| = 6$ and $S[3] = \texttt{b}$.

Definition 3.3: Let S and T be strings. An *alignment* A maps S and T into strings S' and T' that may contain space characters, where

1. $|S'| = |T'|$, and
2. the removal of spaces from S' and T' (without changing the order of the remaining characters) leaves S and T, respectively.

The *value* of the alignment A is

$$\sum_{i=1}^{l} \sigma\left(S'[i], T'[i]\right),$$

where $l = |S'| = |T'|$.

In the example alignment above, if $S = \texttt{acbcdb}$ and $T = \texttt{cadbd}$, then $S' = \texttt{ac--bcdb}$ and $T' = \texttt{-cadb-d-}$.

Definition 3.4: An *optimal alignment* of S and T is one that has the maximum possible value for these two strings.

Finding an optimal alignment of S and T is the way in which we will measure their similarity. For the two strings given in the example above, is the alignment shown optimal? We will next present some algorithms for computing optimal alignments, which will allow us to answer that question.

3.4. An Obvious Algorithm for Optimal Alignment

The most obvious algorithm is to try all possible alignments, and output any alignment with maximum value. We will examine this approach in more detail.

A *subsequence* of a string S means a sequence of characters of S that need not be consecutive in S, but do retain their order as given in S. For instance, $\texttt{acd}$ is a subsequence of $\texttt{acbcdb}$.

Suppose we are given strings S and T, and assume for the moment that $|S| = |T| = n$. Also, consider an arbitrary scoring function $\sigma(x, y)$, subject to the reasonable restriction that $\sigma(-,-) \leq 0$. With this restriction, there is never a reason to align a pair of spaces.

The obvious algorithm for optimal alignment is given in Figure 3.1. This algorithm works correctly, but is it a good algorithm? If you tried running this algorithm on a pair of strings each of length 20 (which is ridiculously modest by biology standards), you would find it much too slow to be practical. The program would run for an hour on such inputs, even if the computer can perform a billion basic operations per second.

for all i, $0 \le i \le n$, **do**
  for all subsequences A of S with $|A| = i$ **do**
    for all subsequences B of T with $|B| = i$ **do**
      Form an alignment that matches $A[k]$ with $B[k]$, $1 \le k \le i$,
        and matches all other characters with spaces;
      Determine the value of this alignment;
      Retain the alignment with maximum value;
    end ;
  end ;
end ;

Figure 3.1: Enumerating all Alignments to Find the Optimal

The running time analysis of this algorithm proceeds as follows. A string of length n has $\binom{n}{i}$ subsequences of length i. [1] Thus, there are $\binom{n}{i}^2$ pairs (A, B) of subsequences each of length i. Consider one such pair. Since there are n characters in S, only i of which are matched with characters in T, there will be $n - i$ characters in S unmatched to characters in T. Thus, the alignment has length $n + (n - i) = 2n - i$. We must look up and add the score of each pair in the alignment, so the total number of basic operations is at least

$$\sum_{i=0}^{n} \binom{n}{i}^2 (2n - i) \ge n \sum_{i=0}^{n} \binom{n}{i}^2 = n\binom{2n}{n} > 2^{2n}, \text{ for } n > 3.$$

(The equality has a pretty combinatorial explanation that is worth discovering. The last inequality follows from Stirling's approximation [39].) Thus, for $n = 20$, this algorithm requires more than $2^{2n} = 2^{40}$ basic operations.

3.5. Asymptotic Analysis of Algorithms

In Section 4.1 we will see a cleverer algorithm that runs in time proportional to n^2. For large n, it is clear that 2^{2n} is greater than n^2. As a demonstration that an algorithm that requires time proportional to n^2 is far more desirable than one that requires time 2^{2n}, consider at what value of n these two functions cross. Suppose the actual running time of the cleverer algorithm is $100n^2 + 100n + 100$. The value of 2^{2n} already exceeds this quadratic at $n = 7$. Suppose instead that the running time is $10,000n^2 + 100n + 100$. Despite the fact that we increased the constant of proportionality by a factor of 100, 2^{2n} already exceeds this quadratic at $n = 10$. This demonstration should make it clear that the rate of growth of the high order term is the most important determinant of how long an algorithm runs on large inputs, independent of the constant of proportionality and any lower order terms.

To formalize this notion, we introduce "big O" notation.

Definition 3.5: Let $f(n)$ and $g(n)$ be functions. Then $f(n) = O(g(n))$ if and only if there is a constant c such that, for all n sufficiently large, $|f(n)| \le cg(n)$.

[1] The notation $\binom{n}{i}$ denotes the number of combinations of n distinguishable objects taken i at a time. See any textbook on combinatorial mathematics, for instance Roberts [39].

For example, $100n^2 + 100n + 100$ and $10,000n^2 + 100n + 100$ are both $O(n^2)$. For the former, $c = 101$ works, and for the latter, $c = 10,001$ works.

Lecture 4

Alignment by Dynamic Programming

January 13, 2000
Notes: Martin Tompa

4.1. Computing an Optimal Alignment by Dynamic Programming

Given strings S and T, with $|S| = n$ and $|T| = m$, our goal is to compute an optimal alignment of S and T. Toward this goal, define $V(i,j)$ as the value of an optimal alignment of the strings $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$.

The value of an optimal alignment of S and T is then $V(n,m)$. The crux of dynamic programming is to solve the more general problems of computing *all* values $V(i,j)$ with $0 \leq i \leq n$ and $0 \leq j \leq m$, in order of increasing i and j. Each of these will be relatively simple to compute, given the values already computed for smaller i and/or j, using a "recurrence relation". To start the process, we need a "basis" for $i = 0$ and/or $j = 0$.

BASIS :

$$
\begin{aligned}
V(0,0) &= 0 \\
V(i,0) &= V(i-1,0) + \sigma(S[i],-), \text{ for } i > 0 \\
V(0,j) &= V(0,j-1) + \sigma(-,T[j]), \text{ for } j > 0
\end{aligned}
$$

The basis for $V(i,0)$ says that if i characters of S are to be aligned with 0 characters of T, then they must all be matched with spaces. The basis for $V(0,j)$ is analogous.

RECURRENCE : For $i > 0$ and $j > 0$,

$$
\begin{array}{llll}
V(i,j) = \max(& V(i-1,j-1) & + \quad \sigma(S[i],T[j]) & , \\
 & V(i-1,j) & + \quad \sigma(S[i],-) & , \\
 & V(i,j-1) & + \quad \sigma(-,T[j]) &)
\end{array}
$$

This formula can be understood by considering an optimal alignment of the first i characters from S and the first j characters from T. In particular, consider the last aligned pair of characters in such an alignment. This last pair must be one of the following:

1. $(S[i],T[j])$, in which case the remaining alignment excluding this pair must be an optimal alignment of $S[1] \cdots S[i-1]$ and $T[1] \cdots T[j-1]$ (i.e., must have value $V(i-1,j-1)$), or

2. $(S[i],-)$, in which case the remaining alignment excluding this pair must have value $V(i-1,j)$, or

3. $(-, T[j])$, in which case the remaining alignment excluding this pair must must have value $V(i, j-1)$.

The optimal alignment chooses whichever among these three possibilities has the greatest value.

4.1.1. Example

In aligning `acbcdb` and `cadbd`, the dynamic programming algorithm fills in the following values for $V(i, j)$ from top to bottom and left to right, simply applying the basis and recurrence formulas. (As in the example of Section 3.3, assume that matches score $+2$, and mismatches and spaces score -1.) For instance, in the table below, the entry in row 4 and column 1 is obtained by computing $\max(-3+2, 0-1, -4-1) = -1$.

i \ j		0	1	2	3	4	5
			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1	-1	1	0	-1	-2
2	c	-2	1	0	0	-1	-2
3	b	-3	0	0	-1	2	1
4	c	-4	-1	-1	-1	1	1
5	d	-5	-2	-2	1	0	3
6	b	-6	-3	-3	0	3	2

The value of the optimal alignment is $V(n, m)$, and so can be read from the entry in the last row and last column. Thus, there is an alignment of `acbcdb` and `cadbd` that has value 2, so the alignment proposed in Section 3.3 with value 1 is not optimal. But how can one determine the optimal alignment itself, and not just its value?

4.1.2. Recovering the Alignments

The solution is to retrace the dynamic programming steps back from the (n, m) entry, determining which preceding entries were responsible for the current one. For instance, in the table below, the (4,2) entry could have followed from either the (3,1) or (3,2) entry; this is denoted by the two arrows pointing to those entries. We can then follow any of these paths from (n, m) to $(0, 0)$, tracing out an optimal alignment:

i \ j		0	1	2	3	4	5
			c	a	d	b	d
0		0	$\leftarrow -1$	-2	-3	-4	-5
1	a	$\uparrow -1$	-1	$\nwarrow$ 1	0	-1	-2
2	c	-2	$\nwarrow$ 1	0	$\nwarrow$ 0	-1	-2
3	b	-3	$\uparrow$ 0	$\nwarrow$ 0	-1	$\nwarrow$ 2	1
4	c	-4	-1	$\nwarrow\uparrow -1$	-1	$\uparrow$ 1	1
5	d	-5	-2	-2	$\nwarrow$ 1	0	$\nwarrow$ 3
6	b	-6	-3	-3	0	$\nwarrow$ 3	$\leftarrow\uparrow$ 2

The optimal alignments corresponding to these three paths are

a	c	b	c	d	b	-
-	c	-	a	d	b	d

,

a	c	b	c	d	b	-
-	c	a	-	d	b	d

, and

-	a	c	b	c	d	b
c	a	d	b	-	d	-

.

Each of these has three matches, one mismatch, and three spaces, for a value of $3 \cdot (2) + 4 \cdot (-1) = 2$, the optimal alignment value.

4.1.3. Time Analysis

Theorem 4.1: The dynamic programming algorithm computes an optimal alignment in time $O(nm)$.

Proof: This algorithm requires an $(n+1) \times (m+1)$ table to be completed. Any particular entry is computed with a maximum of 6 table lookups, 3 additions, and a three-way maximum, that is, in time c, a constant. Thus, the complexity of the algorithm is at most $c(n+1)(m+1) = O(nm)$. Reconstructing a single alignment can then be done in time $O(n+m)$. □

4.2. Searching for Local Similarity

Next we will discuss some variants of the dynamic programming approach to string alignment. We do this to demonstrate the versatility of the approach, and because the variants themselves arise in biological applications.

In the variant called "local similarity", we are searching for regions of similarity between two strings, within contexts that may be dissimilar. An example in which this arises is if we have two long DNA sequences that each contain a given gene, or perhaps closely related genes. Certainly the "global" alignment problem of Definitions 3.3 and 3.4 will not in general identify these genes.

We can formulate this problem as the *local alignment problem*: Given two strings S and T, with $|S| = n$ and $|T| = m$, find substrings (i.e., contiguous subsequences) A of S and B of T such that the optimal (global) alignment of A and B has the maximum value over all such substrings A and B. In other words, the optimal alignment of A and B must have at least as great a value as the optimal alignment of any other substrings A' of S and B' of T.

4.2.1. An Obvious Local Alignment Algorithm

The definition above immediately suggests an algorithm for local alignment:

for all substrings A of S **do**
    for all substrings B of T **do**
        Find an optimal alignment of A and B by dynamic programming;
        Retain A and B with maximum alignment value, and their alignment;
    end ;
end ;
Output the retained A, B, and alignment;

There are $\binom{n+1}{2}$ choices of A, and $\binom{m+1}{2}$ choices of B (excluding the length 0 substrings as choices). Using Theorem 4.1, it is not difficult to show that the time taken by this algorithm is $O(n^3m^3)$. We will see in Section 5.1, however, that it is possible to compute the optimal local alignment in time $O(nm)$, that is, the same time used for the optimal global alignment.

4.2.2. Set-Up for Local Alignment by Dynamic Programming

Definition 4.2: The *empty string* λ is the string with $|\lambda| = 0$.

Definition 4.3: U is a *prefix* of S if and only if $U = S[1] \cdots S[k]$ or $U = \lambda$, for some $1 \leq k \leq n$, where $n = |S|$.

Definition 4.4: U is a *suffix* of S if and only if $U = S[k] \cdots S[n]$ or $U = \lambda$, for some $1 \leq k \leq n$, where $n = |S|$.

For example, let $S =$ `abcxdex`. The prefixes of S include `ab`. The suffixes of S include `xdex`. The empty string λ is both a prefix and a suffix of S.

Definition 4.5: Let S and T be strings with $|S| = n$ and $|T| = m$. For $0 \leq i \leq n$ and $0 \leq j \leq m$, let $v(i, j)$ be the maximum value of an optimal (global) alignment of α and β over all suffixes α of $S[1] \cdots S[i]$ and all suffixes β of $T[1] \cdots T[j]$.

For example, suppose $S =$ `abcxdex` and $T =$ `xxxcde`. Score a match as $+2$ and a mismatch or space as -1. Then $v(5, 5) = 3$, with $\alpha =$ `cxd`, $\beta =$ `cd`, and alignment

c	x	d
c	−	d
+2	−1	+2

The dynamic programming algorithm for optimal local alignment is similar to the dynamic programming algorithm for optimal global alignment given in Section 4.1. It proceeds by filling in a table with the values of $v(i, j)$, with i, j increasing. The value of each entry is calculated according to a new basis and recurrence for $v(i, j)$, given in Section 5.1. Unlike the global alignment algorithm, however, the value of the optimal local alignment can be any entry, whichever contains the maximum of all $(n + 1)(m + 1)$ values of $v(i, j)$. The reason for this is that each $v(i, j)$ entry represents an optimal pair (α, β) of suffixes of a given pair $(S[1] \cdots S[i], T[1] \cdots T[j])$ of prefixes. Since a suffix of a prefix is just a substring, we find the optimal pair of substrings by maximizing $v(i, j)$ over all possible pairs (i, j).

Lecture 5

Local Alignment, and Gap Penalties

January 18, 2000
Notes: Martin Tompa

5.1. Computing an Optimal Local Alignment by Dynamic Programming

BASIS : For simplicity, we will make the reasonable assumption that $\sigma(x, -) \leq 0$ and $\sigma(-, x) \leq 0$. Then

$$\begin{aligned} v(i, 0) &= 0, \text{ and} \\ v(0, j) &= 0, \end{aligned}$$

since the optimal suffix to align with a string of length 0 is the empty suffix.

RECURRENCE : for $i > 0$ and $j > 0$,

$$v(i,j) = \max(\begin{array}{lll} 0 & & , \\ v(i-1, j-1) & + \ \sigma(S[i], T[j]) & , \\ v(i-1, j) & + \ \sigma(S[i], -) & , \\ v(i, j-1) & + \ \sigma(-, T[j]) &) \end{array}$$

The formula looks very similar to the recurrence for the optimal global alignment in Section 4.1. Of course, the meaning is somewhat different and we have an additional term in the $\max$ function. The recurrence is explained as follows. Consider an optimal alignment A of a suffix α of $S[1] \cdots S[i]$ and a suffix β of $T[1] \cdots T[j]$. There are four possible cases:

1. $\alpha = \lambda$ and $\beta = \lambda$, in which case the alignment has value 0.
2. $\alpha \neq \lambda$ and $\beta \neq \lambda$, and the last matched pair in A is $(S[i], T[j])$, in which case the remainder of A has value $v(i-1, j-1)$.
3. $\alpha \neq \lambda$, and the last matched pair in A is $(S[i], -)$, in which case the remainder of A has value $v(i-1, j)$.
4. $\beta \neq \lambda$, and the last matched pair in A is $(-, T[j])$, in which case the remainder of A has value $v(i, j-1)$.

The optimal alignment chooses whichever of these cases has greatest value.

5.1.1. Example

For example, let S = `abcxdex` and T = `xxxcde`, and suppose a match scores $+2$, and a mismatch or a space scores -1. The dynamic programming algorithm fills in the table of $v(i, j)$ values from top to bottom and left to right, as follows:

	j	0	1	2	3	4	5	6
i			x	x	x	c	d	e
0		0	0	0	0	0	0	0
1	a	0	0	0	0	0	0	0
2	b	0	0	0	0	0	0	0
3	c	0	0	0	0	2	1	0
4	x	0	2	2	2	1	1	0
5	d	0	1	1	1	1	3	2
6	e	0	0	0	0	0	2	5
7	x	0	2	2	2	1	1	4

The value of the optimal local alignment is $v(6, 6) = 5$. We can reconstruct optimal alignments as in Section 4.1.2, by retracing from any maximum entry to any zero entry:

	j	0	1	2	3	4	5	6
i			x	x	x	c	d	e
0		0	0	0	0	0	0	0
1	a	0	0	0	0	0	0	0
2	b	0	0	0	0	0	0	0
3	c	0	0	0	0	$\nwarrow$ 2	1	0
4	x	0	2	2	$\nwarrow$ 2	$\hookleftarrow\uparrow$ 1	1	0
5	d	0	1	1	1	1	$\nwarrow$ 3	2
6	e	0	0	0	0	0	2	$\nwarrow$ 5
7	x	0	2	2	2	1	1	4

The optimal local alignments corresponding to these paths are

```
c x d e          x - d e
c - d e   and    x c d e   .
```

Both alignments have three matches and one space, for a value of $3 \cdot (2) + 1 \cdot (-1) = 5$. You can also see from this diagram how the value was derived in the example following Definition 4.5, which said that for the same strings S and T, $v(5, 5) = 3$.

5.1.2. Time Analysis

Theorem 5.1: The dynamic programming algorithm computes an optimal local alignment in time $O(nm)$.

Proof: Computing the value for each of the $(n + 1)(m + 1)$ entries requires at most 6 table lookups, 3 additions, and 1 max calculation. Reconstructing a single alignment can then be done in time $O(n + m)$. □

5.2. Space Analysis

The space required for either the global or local optimal alignment algorithm is also quadratic in the length of the strings being compared. This could be prohibitive for comparing long DNA sequences. There is a modification of the dynamic programming algorithm that computes an optimal alignment in $O(n+m)$ space and still runs in $O(nm)$ time. If one were interested only in the value of an optimal alignment, this could be done simply by retaining only two consecutive rows of the dynamic programming table at any time. Reconstructing an alignment is somewhat more complicated, but can be done in $O(n+m)$ space and $O(nm)$ time with a divide and conquer approach (Hirschberg [24], Myers and Miller [36]).

5.3. Optimal Alignment with Gaps

Definition 5.2: A *gap* in an alignment of S and T is a maximal substring of either S' or T' consisting only of spaces. (Recall from Definition 3.3 that S' and T' are S and T with spaces inserted as dictated by the alignment.)

5.3.1. Motivations

In certain applications, we may not want to have a penalty proportional to the length of a gap.

1. Mutations causing insertion or deletion of large substrings may be considered a single evolutionary event, and may be nearly as likely as insertion or deletion of a single residue.

2. cDNA matching: Biologists are very interested in learning which genes are expressed in which types of specialized cells, and where those genes are located in the chromosomal DNA. Recall from Section 2.5 that eukaryotic genes often consist of alternating exons and introns. The mature mRNA that leaves the nucleus after transcription has the introns spliced out. To study gene expression within specialized cells, one procedure is as follows:

    (a) Capture the mature mRNA as it leaves the nucleus.

    (b) Make *complementary DNA* (abbreviated *cDNA*) from the mRNA using an enzyme called *reverse transcriptase*. The cDNA is thus a concatenation of the gene's exons.

    (c) Sequence the cDNA.

    (d) Match the sequenced cDNA against sequenced chromosomal DNA to find the region of chromosomal DNA from which the cDNA derives. In this process we do not want to penalize heavily for the introns, which will match gaps in the cDNA.

In general, the gap penalty may be some arbitrary function $g(q)$ of the gap length q. The best choice of this function, like the best choice of a scoring function, depends on the application. In the cDNA matching application, we would like the penalty to reflect what is known about the common lengths of introns. In the next section we will see an $O(nm)$ time algorithm for the case when $g(q)$ is an arbitrary linear affine function, and this is adequate for many applications. There are programs that use piecewise linear functions as gap penalties, and these may be more suitable in the cDNA matching application. There are $O(nm \log m)$ time algorithms for the case when $g(q)$ is concave downward (Galil and Giancarlo [17], Miller and Myers [35]). We could even implement an arbitrary function as a gap penalty function, but the known algorithm for this requires cubic time (Needleman and Wunsch [37]), and such an algorithm is probably not useful in practice.

5.3.2. Affine Gap Model

We will study a model in which the penalty for a gap has two parts: a penalty for initiating a gap, and another penalty that depends linearly on the length of a gap. That is, the gap penalty is $W_g + qW_s$ where W_g and W_s are both constants, $W_g \geq 0$, $W_s \geq 0$, and $q \geq 1$ is the length of the gap. (Note that the model with a constant penalty regardless of gap length is the special case with $W_s = 0$.)

For simplicity, assume we are modifying the global alignment algorithm of Section 4.1 to accommodate an affine gap penalty. Similar ideas would work for local alignment as well.

We will assume $\sigma(x, -) = \sigma(-, x) = 0$, since the spaces will be penalized as part of the gap. Our goal then is to maximize

$$\sum_{i=1}^{\ell} \sigma(S'[i], T'[i]) - W_g(\text{\# gaps}) - W_s(\text{\# spaces}),$$

where S' and T' are S and T with spaces inserted, and $|S'| = |T'| = l$.

5.3.3. Dynamic Programming Algorithm

Once again, the algorithm proceeds by aligning $S[1] \cdots S[i]$ with $T[1] \cdots T[j]$. For these prefixes of S and T, define the following variables:

1. $V(i, j)$ is the value of an optimal alignment of $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$.
2. $G(i, j)$ is the value of an optimal alignment of $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$ whose last pair matches $S[i]$ with $T[j]$.
3. $F(i, j)$ is the value of an optimal alignment of $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$ whose last pair matches $S[i]$ with a space.
4. $E(i, j)$ is the value of an optimal alignment of $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$ whose last pair matches a space with $T[j]$.

BASIS :

$$\begin{aligned}
V(0,0) &= 0, \\
V(i,0) &= -W_g - iW_s, \text{ for } i > 0, \\
V(0,j) &= -W_g - jW_s, \text{ for } j > 0, \\
E(i,0) &= -\infty, \text{ for } i > 0, \\
F(0,j) &= -\infty, \text{ for } j > 0.
\end{aligned}$$

RECURRENCE : For $i > 0$ and $j > 0$,

$$V(i,j) = \max(G(i,j),\ F(i,j),\ E(i,j)),$$

$$
\begin{aligned}
G(i,j) &= V(i-1,j-1)+\sigma(S[i],T[j]),\\
F(i,j) &= \max(F(i-1,j)-W_s,\, V(i-1,j)-W_g-W_s),\\
E(i,j) &= \max(E(i,j-1)-W_s,\, V(i,j-1)-W_g-W_s).
\end{aligned}
$$

The equation for $F(i,j)$ (and analogously $E(i,j)$) can be understood as taking the maximum of two cases: adding another space to an existing gap, and starting a new gap. To understand why starting a new gap can use $V(i-1,j)$, which includes the possibility of an alignment ending in a gap, consider that $V(i-1,j) = \max(G(i-1,j),\ F(i-1,j),\ E(i-1,j))$, so that $F(i-1,j)-W_g-W_s$ is always dominated by $F(i-1,j)-W_s$, so will never be chosen by the max.

5.3.4. Time Analysis

Theorem 5.3: An optimal global alignment with affine gap penalty can be computed in time $O(nm)$.

Proof: The algorithm proceeds as those we have studied before, but in this case there are three or four matrices to fill in simultaneously, depending on whether you store the values of $V(i,j)$ or calculate them from the other three matrices when needed. □

5.4. Bibliographic Notes on Alignments

Bellman [6] began the systematic study of dynamic programming. The original paper on global alignment is that of Needleman and Wunsch [37]. Smith and Waterman [45] introduced the local alignment problem, and the $O(nm)$ algorithm to solve it. A number of authors have studied the question of how to construct a good scoring function for sequence comparison, including Karlin and Altschul [27] and Altschul [3].

Lecture 6

Multiple Sequence Alignment

January 20, 2000
Notes: Martin Tompa

While previous lectures discussed the problem of determining the similarity between two strings, this lecture turns to the problem of determining the similarity among multiple strings.

6.1. Biological Motivation for Multiple Sequence Alignment

6.1.1. Representing Protein Families

An important motivation for studying the similarity among multiple strings is the fact that protein databases are often categorized by protein families. A *protein family* is a collection of proteins with similar structure (i.e., three-dimensional shape), similar function, or similar evolutionary history. When we have a newly sequenced protein, we would like to know to which family it belongs, as this provides hypotheses about its structure, function, or evolutionary history. (See Section 3.2.1.) The new protein might not be particularly similar to a single protein in the database, yet might still share considerable similarity with the collective members of a family of proteins. One approach is to construct a representation for each protein family, for example a good multiple sequence alignment of all its members. Then, when we have a newly sequenced protein and want to find its family, we only have to compare it to the representation of each family.

Common structure, function, or origin of a molecule may only be weakly reflected in its sequence. For example, the three-dimensional structure of a protein is very difficult to infer from its sequence, and yet is very important to predict its function. Multiple sequence comparisons may help highlight weak sequence similarity, and shed light on structure, function, or origin.

6.1.2. Repetitive Sequences in DNA

In the DNA domain, a motivation for multiple sequence alignment arises in the study of *repetitive sequences*. These are sequences of DNA, often without clearly understood biological function, that are repeated many times throughout the genome. The repetitions are generally not exact, but differ from each other in a small number of insertions, deletions, and substitutions. As an example, the *Alu* repeat is approximately 300 bp long, and appears over 600,000 times in the human genome. It is believed that as much as 60% of the human genome may be attributable to repetitive sequences without known biological function. (See Jurka and Batzer [26].)

In order to highlight the similarities and differences among the instances of such a repeat family, one would like to display a good multiple sequence alignment of its constituent sequences.

6.2. Formulation of the Multiple String Alignment Problem

We now define the problem more precisely.

Definition 6.1: Given strings $S_1, S_2, \ldots, S_k$ a *multiple (global) alignment* maps them to strings $S'_1, S'_2, \ldots, S'_k$ that may contain spaces, where

1. $|S'_1| = |S'_2| = \cdots = |S'_k|$, and
2. the removal of spaces from S'_i leaves S_i, for $1 \le i \le k$.

The question that arises next is how to assign a value to such an alignment. In a pairwise alignment, we simply summed the similarity score of corresponding characters. In the case of multiple string alignment, there are various scoring methods, and controversy around the question of which is best. We focus here on a scoring method called the "sum-of-pairs" score. Other methods are explored in the homework.

Until now, we have been using a scoring function that assigns higher values to better alignments and lower values to worse alignments, and we have been trying to find alignments with maximum value. For the remainder of this lecture, we will switch to a function $\delta(x, y)$ that measures the *distance* between characters x and y. That is, it will assign higher values the more distant two strings are. In the case of two strings, we will thus be trying to *minimize*

$$\sum_{i=1}^{l} \delta(S'[i], T'[i]),$$

where $l = |S'| = |T'|$.

Definition 6.2: The *sum-of-pairs (SP) value* for a multiple global alignment A of k strings is the sum of the values of all $\binom{k}{2}$ pairwise alignments induced by A.

In this definition we assume that the scoring function is symmetric. For simplicity, we will not discuss the issue of a separate gap penalty.

Example 6.3: Consider the following alignment:

```
a c - c d b -
- c - a d b d
a - b c d a d
```

Using the distance function $\delta(x, x) = 0$, and $\delta(x, y) = 1$ for $x \neq y$, this alignment has a sum-of-pairs value $3 + 5 + 4 = 12$.

Definition 6.4: An *optimal SP (global) alignment* of strings $S_1, S_2, \ldots, S_k$ is an alignment that has the minimum possible sum-of-pairs value for these k strings.

6.3. Computing an Optimal Multiple Alignment by Dynamic Programming

Given k strings each of length n, there is a generalization of the dynamic programming algorithm of Section 4.1 that finds an optimal SP alignment. Instead of a 2-dimensional table, it fills in a k-dimensional table. This table has dimensions

$$\underbrace{(n+1) \times (n+1) \cdots \times (n+1)}_{k},$$

that is, $(n+1)^k$ entries. Each entry depends on $2^k - 1$ adjacent entries, corresponding to the possibilities for the last match in an optimal alignment: any of the 2^k subsets of the k strings could participate in that match, except for the empty subset. The details of the algorithm itself and the recurrence are left as exercises for the reader.

Because each of the $(n+1)^k$ entries can be computed in time proportional to 2^k, the running time of the algorithm is $O((2n)^k)$. If $n \approx 350$ (as is typical for the length of proteins), it would be practical only for very small values of k, perhaps 3 or 4. However, typical protein families have hundreds of members, so this algorithm is of no use in the motivational problem posed in Section 6.1. We would like an algorithm that works for k in the hundreds too, which would be possible only if the running time were polynomial in both n and k. (In particular, k should not appear in the exponent as it does in the expression $(2n)^k$.) Unfortunately, we are very unlikely to find such an algorithm, which is a consequence of the following theorem:

Theorem 6.5 (Wang and Jiang [50]): The optimal SP alignment problem is *NP*-complete.

What *NP*-completeness means and what its consequences are will be discussed in the following section.

6.4. *NP*-completeness

In this section we give a brief introduction to *NP*-completeness, and how problems can be proved to be *NP*-complete.

Definition 6.6: A problem has a *polynomial time solution* if and only if there is some algorithm that solves it in time $O(n^c)$, where c is a constant and n is the size of the input.

Many familiar computational problems have polynomial time solutions:

1. two-string optimal alignment problem: $O(n^2)$ (Theorem 4.1) ,
2. sorting: $O(n \log n)$ [12],
3. two-string alignment with arbitrary gap penalty function: $O(n^3)$ (Section 5.3.1),
4. 100-string optimal alignment problem: $O(n^{100})$ (Section 6.3).

The last entry illustrates that having a polynomial time solution does not mean that the algorithm is practical. In most cases the converse, though, is true: an algorithm whose running time is not polynomial is likely to be impractical for all but the smallest size inputs.

NP-complete problems are equivalent in the sense that if any one of them has a polynomial time solution, then all of them do. One of the major open questions in computer science is whether there is a polynomial time solution for any of the *NP*-complete problems. Almost all experts conjecture strongly that the answer to this question is “no”. The bulk of the evidence supporting this conjecture, however, is only the failure to find such a polynomial time solution in thirty years.

In 1971, Cook defined the notion of *NP*-completeness and showed the *NP*-completeness of a small collection of problems, most of them from the domain of mathematical logic [11]. Roughly speaking, he defined *NP*-complete problems to be problems that have the property that we can verify in polynomial time whether a supplied solution is correct. For instance, if you did not have to *compute* an optimal SP alignment, but simply had to *verify* that a given alignment had SP value at most x, a given integer, it would be easy to write a polynomial time algorithm to do so.

Shortly after Cook’s work, Karp recognized the wide applicability of the concept of *NP*-completeness. He showed that a diverse host of problems are each *NP*-complete [28]. Since then, many hundreds of natural problems from many areas of computer science and mathematics such as graph theory, combinatorial optimization, scheduling, and symbolic computation have been proven *NP*-complete; see Garey and Johnson [18] for details.

Proving a problem Q to be *NP*-complete proceeds in the following way. Choose a known *NP*-complete problem A. Show that A has a polynomial time algorithm if it is allowed to invoke a polynomial time subroutine for Q, and vice versa.

There are many computational biology problems that are *NP*-complete, yet in practice we still need to solve them somehow. There are different ways to deal with an *NP*-complete problem:

1. We might give up on the possibility of solving the problem on anything but small inputs, by using an exhaustive (nonpolynomial time) search algorithm. We can sometimes use dynamic programming or branch-and-bound techniques to cut down the running time of such a brute force exhaustive search.

2. We might give up guaranteed efficiency by settling for an algorithm that is sufficiently efficient on inputs that arise in practice, but is nonpolynomial on some worst-case inputs that (hopefully) do not arise in practice. There may be an algorithm that runs in polynomial time on average inputs, being careful to define the input distribution so that the practical inputs are highly probable.

3. We might give up guaranteed optimality of solution quality by settling for an approximate algorithm that gives a suboptimal solution, especially if the suboptimal solution is provably not much worse than the optimal solution. (An example is given in Section 6.5.)

4. Heuristics (local search, simulated annealing, “genetic” algorithms, and many others) can also be used to improve the quality of solution or running time in practice. We will see several examples throughout the remaining lectures. However, rigorous analysis of heuristic algorithms is generally unavailable.

5. The problem to be solved in practice may be more specialized than the general one that was proved *NP*-complete.

In the following section we will look at the approximation approach to find a solution for the multiple string alignment problem.

6.5. An Approximation Algorithm for Multiple String Alignment

In this section we will show that there is a polynomial time algorithm (called the *Center Star Alignment Algorithm*) that produces multiple string alignments whose SP values are less than twice that of the optimal solutions. This result is due to Gusfield [19]. Although the factor of 2 may be unacceptable in some applications, the result will serve to illustrate how approximation algorithms work.

In this section we will make the following assumptions about the distance function:

1. $\delta(x, x) = 0$, for all characters x.
2. Triangle Inequality: $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$, for all characters x, y, and z, and

The triangle inequality says that the distance along one edge of a triangle is at most the sum of the distances along the other two edges. Although intuitively plausible, be aware that not all distance measures used in biology obey the triangle inequality.

6.5.1. Algorithm

Definition 6.7: For strings S and T, define $D(S, T)$ to be the value of the minimum (global) alignment distance of S and T.

The approximation algorithm is as follows. The input is a set $\mathcal{T}$ of k strings. First find $S_1 \in \mathcal{T}$ that minimizes

$$\sum_{S \in \mathcal{T} - \{S_1\}} D(S_1, S).$$

This can be done by running the dynamic programming algorithm of Section 4.1 on each of the $\binom{k}{2}$ pairs of strings in $\mathcal{T}$. Call the remaining strings in $\mathcal{T}$ $S_2, \ldots, S_k$. Add these strings $S_2, \ldots, S_k$ one at a time to a multiple alignment that initially contains only S_1, as follows.

Suppose $S_1, S_2, \ldots, S_{i-1}$ are already aligned as $S'_1, S'_2, \ldots, S'_{i-1}$. To add S_i, run the dynamic programming algorithm of Section 4.1 on S'_1 and S_i to produce S''_1 and S'_i. Adjust $S'_2, \ldots, S'_{i-1}$ by adding spaces to those columns where spaces were added to get S''_1 from S'_1. Replace S'_1 by S''_1.

6.5.2. Time Analysis

Theorem 6.8: The approximation algorithm of Section 6.5.1 runs in time $O(k^2 n^2)$ when given k strings each of length at most n.

Proof: By Theorem 4.1, each of the $\binom{k}{2}$ values $D(S, T)$ required to compute S_1 can be computed in time $O(n^2)$, so the total time for this portion is $O(\binom{k}{2} n^2) = O(k^2 n^2)$. After adding S_i to the multiple string alignment, the length of S'_1 is at most in, so the time to add all n strings to the multiple string alignment is

$$\sum_{i=1}^{k-1} O((in) \cdot n) = O(k^2 n^2).$$

□

6.5.3. Error Analysis

What remains to be shown is that the algorithm produces a solution that is less than a factor of 2 worse than the optimal solution. Let M be the alignment produced by this algorithm, let $d(i,j)$ be the distance M induces on the pair S_i, S_j, and let

$$v(M) = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(i,j).$$

Note that $v(M)$ is exactly twice the SP score of M, since every pair of strings is counted twice.

Then $d(1,l) = D(S_1, S_l)$ for all l. This is because the algorithm used an optimal alignment of S_1' and S_l, and $D(S_1', S_l) = D(S_1, S_l)$, since $\delta(-,-) = 0$. If the algorithm later adds spaces to both S_1' and S_l', it does so in the same columns.

Let M^* be the optimal alignment, $d^*(i,j)$ be the distance M^* induces on the pair S_i, S_j, and

$$v(M^*) = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(i,j).$$

Theorem 6.9: $\frac{v(M)}{v(M^*)} \leq \frac{2(k-1)}{k} < 2$. That is, the algorithm of Section 6.5.1 produces an alignment whose SP value is less than twice that of the optimal SP alignment.

Proof: We will derive an upper bound on $v(M)$ and a lower bound on $v(M^*)$, and then take their quotient.

$$\begin{aligned}
v(M) &= \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(i,j) \\
&\leq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} (d(i,1) + d(1,j)) && \text{(triangle inequality)} \\
&= 2(k-1) \sum_{l=2}^{k} d(1,l) && \text{(explained below)} \\
&= 2(k-1) \sum_{l=2}^{k} D(S_1, S_l)
\end{aligned}$$

The third line follows because each $d(l,1) = d(1,l)$ occurs in $2(k-1)$ terms of the second line.

$$\begin{aligned}
v(M^*) &= \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(i,j) \\
&\geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(S_i, S_j) && \text{(Definition 6.7)}
\end{aligned}$$

$$\geq \sum_{i=1}^{k}\sum_{j=2}^{k} D(S_1, S_j) \qquad \text{(definition of } S_1\text{)}$$

$$= k\sum_{l=2}^{k} D(S_1, S_l)$$

Combining these inequalities,

$$\frac{v(M)}{v(M^*)} \leq \frac{2(k-1)}{k} < 2.$$

□

Note that for small values of k, the approximation is significantly better than a factor of 2. Furthermore, the error analysis does not mean that the approximation solution is always $2(k-1)/k$ times the optimal solution. It means that the quality of the solution is never worse than this, and may be better in practice.

6.5.4. Other Approaches

In the Center Star Algorithm discussed in Section 6.5.1, we always try to align the chosen center string S_1 with the unaligned strings. However, there might be cases in which some of the strings are very "near" to each other and form "clusters". It might be an advantage to align strings in the same cluster first, and then merge the clusters of strings. The problem with this is how to define "near" and how to define "clusters".

There are many variants on this idea, which sometimes are called *iterative pairwise alignment* methods. Here is one version: an unaligned string nearest to any aligned string is picked and aligned with the previously aligned group. (For those who have seen it before, note the similarity to Prim's minimum spanning tree algorithm [12].) The "nearest" string is chosen based on optimal pairwise alignments between individual strings in the multiple alignment and unaligned strings, without regard to spaces inserted in the multiple alignment. Now the problem is to specify how to align a string with a *group* of strings. One possible method is to mimic the technique that was used to add S_i to the center star alignment in Section 6.5.1.

6.6. The Consensus String

Given a multiple string alignment, it is sometimes useful to derive from it a "consensus string" that can be used to represent the entire set of strings in the alignment.

Definition 6.10: Given a multiple alignment M of strings $S_1, S_2, \ldots, S_k$, the *consensus character* of column i of M is the character c_i that minimizes the sum of distances to it from all the characters in column i; that is, it minimizes $\sum_{j=1}^{k} \delta(S'_j[i], c_i)$. Let $d(i)$ be this minimum sum. The *consensus string* is the concatenation $c_1c_2\cdots c_l$ of all the consensus characters, where $l = |S'_1| = \cdots = |S'_k|$. The *alignment error* of M is then defined to be $\sum_{i=1}^{l} d(i)$.

For instance, the consensus string for the multiple string alignment in Example 6.3 is `ac-cdbd`, and its alignment error is 6, the number of characters in the aligned strings that differ from the consensus character in the corresponding position.

6.7. Summary

Multiple sequence alignment is a very important problem in computational biology. It appears to be impossible to obtain exact solutions in polynomial time, even with very simple scoring functions. A variety of (provably) bounded approximation algorithms are known, and a number of heuristic algorithms have been suggested, but it still remains largely an open problem.

Lecture 7

Finding Instances of Known Sites

January 25, 2000
Notes: Elisabeth Rosenthal

With this lecture we begin a study of how to identify functional regions from biological sequence data. This includes the problem of how to identify relatively long functional regions such as genes, but we begin instead with the problem of identifying shorter functional regions.

A *site* is a short sequence that contains some signal, that signal often being recognized by some enzyme. Examples of nucleotide sequence sites include the following:

1. origins of replication, where DNA polymerase initially binds (Section 1.5),
2. transcription start and stop sites (Section 1.6.1),
3. ribosome binding sites in prokaryotes (Section 2.2),
4. promoters, or transcription factor binding sites (Section 2.3), and
5. intron splice sites (Section 2.5).

We will further subdivide the problem of identifying sites into the problems of finding instances of a known site, and finding instances of unknown sites. We begin with the former. What makes all these problems interesting and challenging is that instances of a single site will generally not be identical, but will instead vary slightly.

7.1. How to Summarize Known Sites

Suppose that we have a large sample $\mathcal{A}$ of length n sites, and a large sample $\mathcal{B}$ of length n nonsites. Given a new sequence $s = s_1 s_2 \cdots s_n$ of length n, is s more likely to be a site or a nonsite? If we can derive an efficient way to determine this, we can screen an entire genome, testing every length n sequence, and thereby generate a "complete" list of candidate sites (excepting sequences where the test gives the wrong answer).

To illustrate, the *cyclic AMP receptor protein* (CRP) is a transcription factor (see Section 2.3) in *E. coli*. Its binding sites are DNA sequences of length approximately 22. Table 7.1, taken from Stormo and Hartzell [47], shows just positions 3–9 (out of the 22 sequence positions) in 23 *bona fide* CRP binding sites.

The "signal" in Table 7.1 is not easy to detect at first glance. Notice, though, that in the second column T predominates and in the third column G predominates, for example. Our first goal is to capture the most relevant information from these 23 sites in a concise form. (This would clearly be more important if we

```
TTGTGGC
TTTTGAT
AAGTGTC
ATTTGCA
CTGTGAG
ATGCAAA
GTGTTAA
ATTTGAA
TTGTGAT
ATTTATT
ACGTGAT
ATGTGAG
TTGTGAG
CTGTAAC
CTGTGAA
TTGTGAC
GCCTGAC
TTGTGAT
TTGTGAT
GTGTGAA
CTGTGAC
ATGAGAC
TTGTGAG
```

Table 7.1: Positions 3–9 from 23 CRP Binding Sites [47]

A	0.35	0.043	0	0.043	0.13	0.83	0.26
C	0.17	0.087	0.043	0.043	0	0.043	0.3
G	0.13	0	0.78	0	0.83	0.043	0.17
T	0.35	0.87	0.17	0.91	0.043	0.087	0.26

Table 7.2: Profile for CRP Binding Sites Given in Table 7.1

were given thousands of sites rather than just 23.) In order to do this, suppose that the sequence residues are from an alphabet of size c. Consider a $c \times n$ matrix A where $A_{r,j}$ is the fraction of sequences in $\mathcal{A}$ that have residue r in position j. Table 7.2 shows the 4×7 matrix A for the CRP sites given in Table 7.1. Such a matrix is called a *profile*. The profile shows the distribution of residues in each of the n positions. For example, in column 1 of the matrix the residues are quite mixed, in column 2, T occurs 87% of the time, etc.

7.2. Using Probabilities to Test for Sites

An alternative way to think of $A_{r,j}$ is in terms of probability. Let $t = t_1 t_2 \cdots t_n$ be chosen randomly and uniformly from $\mathcal{A}$. Then $A_{r,j} = \Pr(t_j = r \mid t \in \mathcal{A})$. In words, this says, "$A_{r,j}$ is the probability that the j-th residue of t is the residue r, given that t is chosen randomly from $\mathcal{A}$." For instance, $A_{T,2} = \Pr(t_2 = T \mid t \in \mathcal{A}) = 0.87$.

For the time being, we will make the following *Independence Assumption*: which residue occurs at position j is independent of the residues occurring at other positions. In other words, residues at any two different positions are uncorrelated. Although this assumption is not always realistic, it can be justified in some circumstances. The first justification is that it keeps the model and resulting analysis simple. The second justification is its predictive power in some (but admittedly not all) situations.

The independence assumption can be made precise in probabilistic terms:

Definition 7.1: Two probabilistic events E and F are said to be *independent* if the probability that they both occur is the product of their individual probabilities, that is, $\Pr(E \,\&\, F) = \Pr(E) \cdot \Pr(F)$.

Under the independence assumption, the probability that a randomly chosen site has a specified sequence $r_1 r_2 \cdots r_n$ is determined by Definition 7.1 as follows:

$$\begin{aligned}
\Pr(t = r_1 r_2 \cdots r_n \mid t \text{ is a site}) &= \Pr(t_1 = r_1 \,\&\, t_2 = r_2 \,\&\, \cdots \,\&\, t_n = r_n \mid t \text{ is a site}) \\
&= \prod_{j=1}^{n} \Pr(t_j = r_j \mid t \text{ is a site}) \\
&= \prod_{j=1}^{n} A_{r_j,j}. \qquad (7.1)
\end{aligned}$$

For example, suppose we want to know the probability that a randomly chosen CRP binding site will be TTGTGAC. By using Equation (7.1) and Table 7.2,

$$\Pr(t = \text{TTGTGAC} \mid t \text{ is a site}) = (.35)(.87)(.78)(.91)(.83)(.83)(.3) = 0.045.$$

Although this probability is small, it is the largest probability of any site sequence, because each position contains the most probable residue.

Now form the $c \times n$ profile B from the sample $\mathcal{B}$ of nonsites in the same way. Using the profiles A and B, let us return to the question of whether a given sequence s is more likely to be a site or nonsite. In order to do this, we define the likelihood ratio.

Definition 7.2: Given the sequence $s = s_1 s_2 \cdots s_n$, the *likelihood ratio*, denoted by $LR(A, B, s)$, is defined to be

$$\frac{\Pr(t = s \mid t \text{ is a site})}{\Pr(t = s \mid t \text{ is a nonsite})} = \frac{\prod_{j=1}^{n} A_{s_j,j}}{\prod_{j=1}^{n} B_{s_j,j}} = \prod_{j=1}^{n} \frac{A_{s_j,j}}{B_{s_j,j}}.$$

A	0.48	-2.5	$-\infty$	-2.5	-0.94	1.7	0.061
C	-0.52	-1.5	-2.5	-2.5	$-\infty$	-2.5	0.28
G	-0.94	$-\infty$	1.6	$-\infty$	1.7	-2.5	-0.52
T	0.48	1.8	-0.52	1.9	-2.5	-1.5	0.061

Table 7.3: Log Likelihood Weight Matrix for CRP Binding Sites

To illustrate, let $\mathcal{B} = \{A, C, T, G\}^7$, the set of all length seven sequences. The corresponding profile B has $B_{r,j} = 0.25$ for all r and j. Then for $s =$ TTGTGAC,

$$LR(A, B, s) = \frac{\prod_{j=1}^{n} A_{s_j,j}}{\prod_{j=1}^{n} B_{s_j,j}} = \frac{0.045}{(0.25)^7} = 732.$$

To test a sequence s, compare $LR(A, B, s)$ to a prespecified constant "cutoff" L, and declare s more likely to be a site if $LR(A, B, s) \geq L$.

If n is not small and some entries in A and B are small, then the likelihood ratio may be intractably large or small, causing numerical problems in the calculation. To alleviate this, we define the log likelihood ratio.

Definition 7.3: Given the sequence $s = s_1 s_2 \cdots s_n$, the *log likelihood ratio*, denoted by $LLR(A, B, s)$, is defined to be

$$\log_2 LR(A, B, s) = \log_2 \prod_{j=1}^{n} \frac{A_{s_j,j}}{B_{s_j,j}} = \sum_{j=1}^{n} \log_2 \frac{A_{s_j,j}}{B_{s_j,j}}.$$

The corresponding test of s is that s is more likely to be a site if $LLR(A, B, s) \geq \log_2 L$.

To test for sites, it is convenient to create a scoring matrix W whose entries are the log likelihood ratios, that is, $W_{r,j} = \log_2 \frac{A_{r,j}}{B_{r,j}}$. Table 7.3 shows the weight matrix for the example CRP samples $\mathcal{A}$ and $\mathcal{B}$ we have been discussing. In order to compute $LLR(A, B, s)$, Definition 7.3 says to add the corresponding scores from W: $LLR(A, B, s) = \sum_{j=1}^{n} W_{s_j,j}$.

A technical difficulty arises when an entry $A_{r,j}$ is 0, because the corresponding entry $W_{r,j}$ is then $-\infty$. If the residue r cannot possibly occur in position j of any site for biological reasons, then there is no problem. More often, though, this is a result of having too small a sample $\mathcal{A}$ of sites. In this case, there are various "small sample correction" formulas, which replace $A_{r,j}$ by a small positive number (see, for example, Lawrence *et al.* [29]), but we will not discuss them here.

Lecture 8

Relative Entropy

January 27, 2000
Notes: Anne-Louise Leutenegger

8.1. Weight Matrices

A *weight matrix* is any $c \times n$ matrix W that assigns a score to each sequence $s = s_1 s_2 \cdots s_n$ according to the formula $\sum_{j=1}^{n} W_{s_j,j}$. The log likelihood ratio matrix described at the end of Section 7.2, and illustrated in Table 7.3, is an example of a weight matrix.

In computing log likelihood ratios, we often take $B_{r,j}$ to be the "background" distribution of residue r in the entire genome, or a large portion of the genome. That is, $B_{r,j}$ is the frequency with which residue r appears in the genome as a whole. In this case, $B_{r,j}$ is independent of j, that is, $B_{r,j} = B_{r,j'}$ for all j and j'. Note, however, that this does not mean that $B_{r,j} = 0.25$ in the case of nucleotides. Although this *uniform* distribution is a fair estimate for the nucleotide composition of *E. coli*, it is not for other organisms. For instance, the nucleotide composition for the archaeon *M. jannaschii* is approximately $B_{A,j} = B_{T,j} = 0.34$ and $B_{C,j} = B_{G,j} = 0.16$.

8.2. A Simple Site Example

Example 8.1: As a simpler example of a collection of sites than the CRP binding sites of Table 7.1, Table 8.1 shows eight hypothetical translation start sites. For this example, we will assume a uniform background distribution $B_{r,j} = 0.25$. Table 8.2(a) shows the site profile matrix, and Table 8.2(b) the log likelihood ratio weight matrix, for this example. As illustrations of the log likelihood ratio calculations,

ATG
ATG
ATG
ATG
ATG
GTG
GTG
TTG

Table 8.1: Eight Hypothetical Translation Start Sites

A	0.625	0	0
C	0	0	0
G	0.25	0	1
T	0.125	1	0

(a)

A	1.32	$-\infty$	$-\infty$
C	$-\infty$	$-\infty$	$-\infty$
G	0	$-\infty$	2
T	-1	2	$-\infty$

(b)

	0.701	2	2

(c)

Table 8.2: (a) Profile, (b) Log Likelihood Weight Matrix, and (c) Positional Relative Entropies, for the Sites in Table 8.1, with Respect to Uniform Background Distribution

$W_{T,2} = \log_2 \frac{A_{T,2}}{B_{T,2}} = \log_2 \frac{1}{0.25} = \log_2 4 = 2$, and $W_{G,1} = \log_2 \frac{0.25}{0.25} = 0$, meaning both distributions have the same frequency for G in position 1.

8.3. How Informative is the Log Likelihood Ratio Test?

The next question to ask is how informative is a given weight matix W for distinguishing between sites and nonsites. If the distributions for sites and nonsites were identical, then every entry in the weight matrix would be 0, and it would be totally uninformative.

Definition 8.2: A *sample space* S is the set of all possible values of some random variable s.

Definition 8.3: A *probability distribution* P for a sample space S assigns a probability $P(s)$ to every $s \in S$, satisfying

1. $0 \leq P(s) \leq 1$, and
2. $\sum_{s \in S} P(s) = 1$.

In our application, the sample space is the set of all length n sequences. The site profile A induces a probability distribution on this sample space according to Equation (7.1), as does the nonsite profile B.

Definition 8.4: Let P and Q be probability distributions on the same sample space S. The *relative entropy* (or "information content", or "Kullback-Leibler measure") of P with respect to Q is denoted $D_b(P||Q)$ and is defined as follows:

$$D_b(P||Q) = \sum_{s \in S} P(s) \log_b \frac{P(s)}{Q(s)}.$$

By convention, we define $P(s) \log_b \frac{P(s)}{Q(s)}$ to be 0 whenever $P(s) = 0$, in agreement with the fact from calculus that $\lim_{x \to 0} x \log x = 0$.

Since $\log \frac{P(s)}{Q(s)}$ is the log likelihood ratio, $D_b(P||Q)$ is a weighted average of the log likelihood ratio with weights $P(s)$.

Definition 8.5: The *expected value* of a function $f(s)$ with respect to probability distribution P on sample space S is

$$E(f(s)) = \sum_{s \in S} P(s) f(s).$$

In these terms, the relative entropy is the expected value of $LLR(P, Q, s)$ when s is picked randomly according to $P(s)$. That is, it is the expected log likelihood score of a randomly chosen site.

Note that when P and Q are the same distribution, the relative entropy will be zero. In general, the relative entropy measures how different the distributions P and Q are. Since we want to be able to distinguish between sites and nonsites, we want the relative entropy to be large, and will use relative entropy as our measure of how informative the log likelihood ratio test is.

When the sample space is all length n sequences, and we assume independence of the n positions, it is not difficult to prove that the relative entropy satisfies

$$D_b(P||Q) = \sum_{j=1}^{n} D_b(P_j||Q_j),$$

where P_j is the distribution P imposes on the jth position and Q_j is the distribution Q imposes on the jth position.

When $b = 2$, the relative entropy is measured in "bits". This will be the usual case, unless specifically stated otherwise.

Continuing Example 8.1, Table 8.2(c) shows the relative entropies $D_2(P_j||Q_j)$ for each nucleotide position j separately. For instance, looking at position 2, residues A, C, and G do not contribute to the relative entropy (see Table 8.2(a)). Residue T contributes $1 \cdot W_{T,2} = 2$ (see Tables 8.2(a) and (b)). Hence, $D_2(P_2||Q_2) = 2$. This means that there are 2 bits of information in position 2. If the residues were coded with 0 and 1 so that 00 = A, 01 = C, 10 = G, and 11 = T, only 2 bits (11) would be necessary to encode the fact that this residue is always T. Position 3 has the same relative entropy of 2. For position 1, the relative entropy is 0.7 so there are 0.7 bits of information, indicating that column 1 of Table 8.2(a) is more similar to the background distribution than columns 2 and 3 are. The total relative entropy of all three positions is 4.7.

Example 8.6: Let us now modify Example 8.1 to see the effect of a nonuniform background distribution. Consider the same eight translation start sites of Table 8.1, but change the background distribution to $B_{A,j} = B_{T,j} = 0.375$, $B_{C,j} = B_{G,j} = 0.125$. The site profile matrix remains unchanged (Table 8.2(a)). The new weight matrix and relative entropies are given in Table 8.3.

Note that the relative entropy of each position has changed and, in particular, the last two columns no longer have equal relative entropy. The site distribution in position 2 is now more similar to the background distribution than the site distribution in position 3 is, since G is rarer in the background distribution. Thus, the relative entropy of position 3 is greater than that of position 2. An interpretation of $D_2(P_3||Q_3) = 3$ is that the residue G is $2^3 = 8$ times more likely to occur in the third position of a site than a nonsite. The total relative entropy of all three positions is 4.93.

A	0.737	$-\infty$	$-\infty$
C	$-\infty$	$-\infty$	$-\infty$
G	1	$-\infty$	3
T	-1.58	1.42	$-\infty$

(b)

	0.512	1.42	3

(c)

Table 8.3: (b) Log Likelihood Weight Matrix, and (c) Positional Relative Entropies, for the Sites in Table 8.1, with Respect to a Nonuniform Background Distribution

0.12	1.3	1.1	1.5	1.2	1.1	0.027

Table 8.4: Positional Relative Entropy for CRP Binding Sites of Tables 7.1 – 7.3

Example 8.7: Finally, returning to the more interesting CRP binding sites of Table 7.1, the seven positional relative entropies are given in Table 8.4. Note that 1.5 (middle position) is the highest relative entropy and corresponds to the most biased column (see Table 7.2). The value 0.027 (last position) is the lowest relative entropy because the distribution in this last position is the closest to the uniform background distribution (see Table 7.2).

8.4. Nonnegativity of Relative Entropy

In these examples, the relative entropy has always been nonnegative. It is by no means obvious that this should be, since it is the expected value of the log likelihood ratio, which can take negative values. For instance, why should the expected value of the last column of Table 7.3 be positive (0.027, according to Table 8.4)? The following theorem demonstrates that this must, indeed, be the case.

Theorem 8.8: For any probability distributions P and Q over a sample space S, $D_b(P||Q) \geq 0$, with equality if and only if P and Q are identical.

Proof: First, it is true that $\ln x \leq x - 1$ for all real numbers x, with equality if and only if $x = 1$. The reason is that the curve $y = \ln x$ is concave downward, and its tangent at $x = 1$ is the straight line $y = x - 1$. Thus, $\ln \frac{1}{x} = \ln(x^{-1}) = -\ln x \geq 1 - x$. In the following derivation, we will use this inequality with $x = \frac{Q(s)}{P(s)}$:

$$
\begin{aligned}
D_b(P||Q) &= \sum_{s \in S} P(s) \log_b \frac{P(s)}{Q(s)} \\
&= \frac{1}{\ln b} \sum_{s \in S} P(s) \ln \frac{P(s)}{Q(s)} \\
&\geq \frac{1}{\ln b} \sum_{s \in S} P(s)(1 - \frac{Q(s)}{P(s)}) \\
&= \frac{1}{\ln b} \sum_{s \in S} (P(s) - Q(s)) \\
&= \frac{1}{\ln b} (\sum_{s \in S} P(s) - \sum_{s \in S} Q(s)) \\
&= 0,
\end{aligned}
$$

since $\sum_{s \in S} P(s) = \sum_{s \in S} Q(s) = 1$, by Definition 8.3. Note that the relative entropy is equal to 0 if and only if $x = Q(s)/P(s) = 1$ for all $s \in S$, that is, P and Q are identical probability distributions. $\square$

Lecture 9

Relative Entropy and Binding Energy

February 1, 2000
Notes: Neil Spring

Binding energy is a measure of the affinity between two molecules. Because it is an expression of free energy released rather than absorbed, a large negative number conventionally represents a strong affinity, and suggests that these molecules are likely to bind. The binding energy depends on a number of factors such as temperature and salinity, which we will assume are not varying.

This lecture describes a paper of Stormo and Fields [46], which investigates the binding energy between a given DNA-binding protein and various short DNA sequences. In particular, it discusses an interesting relationship between binding energy and log likelihood weight matrices, shedding a new light on the relative entropy.

9.1. Experimental Determination of Binding Energy

Given a DNA-binding protein P, we would like to determine with what binding energy P binds to all possible length n DNA sequences. The "binary" question of whether or not P will bind to a particular DNA sequence oversimplifies a more complicated process: more realistically, P binds to most such sequences, but will occupy preferred sites for a greater fraction of time than others. Binding energies reflect this reality more clearly.

If c is the alphabet size, then one cannot hope to perform all the experiments to measure the binding energy of P with each of the possible c^n sequences of length n. Instead, Stormo and Fields proposed the following experimental method for estimating the binding energy of P with each length n sequence.

1. Choose some good site S of length n.
2. Construct all sequences of length n that differ from S in only one residue. There are $(c-1)n$ such sequences.
3. For each such sequence S', experimentally measure the difference in binding energy between P binding with S and P binding with S'.
4. Record the results in a $c \times n$ matrix G, where $G_{r,j}$ is the change in binding energy when residue r is substituted at position j in S.

Stormo and Fields then make the approximating assumption that changes in energy are additive. That is, the change in binding energy for any collection of substitutions is the sum of the changes in binding energy of those individual substitutions. With this assumption, one can predict the binding energy of P to any length n sequence $s = s_1 s_2 \cdots s_n$ by the following formula:

$$\sum_{j=1}^{n} G_{s_j,j}.$$

Thus, G is a weight matrix that assigns a score to each sequence s according to the usual weight matrix formula given in Section 8.1.

9.2. Computational Estimation of Binding Energy

Unfortunately, creating the matrix G for every DNA-binding protein in every organism of interest still requires an infeasible amount of experimental work. This motivated Stormo and Fields to ask how to approximate G computationally, given a collection $\mathcal{A}$ of good binding sites for P and a collection $\mathcal{B}$ of nonsites.

Choosing W to be the log likelihood ratio weight matrix for $\mathcal{A}$ with respect to $\mathcal{B}$ assigns the highest scores to the sites in $\mathcal{A}$. Since G also assigns high (negative) scores to the sites in $\mathcal{A}$, there is good reason to expect that W approximates G well (after the appropriate scaling).

Recall from Section 8.3 that the relative entropy $D_2(A||B)$ is the expected score assigned by W to a randomly chosen site. If W approximates G well, the relative entropy $D_2(A||B)$ then approximates the expected binding energy of P to a randomly chosen site. This provides us a new interpretation of relative entropy.

It also provides an estimate of how great we should expect the relative entropy to be for a good collection of binding sites. There is some probability that a good site will appear in the genomic background simply by chance. This probability increases with the size Γ of the genome. If the relative entropy is too small with respect to Γ, the expected binding energy at true sites will be too small, and the protein will spend too much time occupying nonsites.

Stormo and Fields suggest from experience that the relative entropy for binding sites will be close to $\log_2 \Gamma$. A simple scenario suggests some intuition for this particular estimate: Assume a uniform background distribution $B_{r,j} = 0.25$, and assume that the site profile A has a 1 in each column, that is, all sites are identical. This imples that the relative entropy is 2 bits per position (as in two of the columns of Table 8.2), so the total relative entropy is $D_2(A||B) = 2n$. In a random sequence generated according to B, one would expect this site sequence to appear once every 4^n residues. In order for P not to bind to too many random locations in the background, $4^n = 2^{2n} = 2^{D_2(A||B)}$ must be not much less than Γ, so $D_2(A||B)$ must be not much less than $\log_2 \Gamma$.

9.3. Finding Instances of an Unknown Site

This leads us into our next topic. Suppose we are not given a sample $\mathcal{A}$ of known sites. We want to find sequences that are significantly similar to each other, without any *a priori* knowledge of what those sequences look like. A little more precisely, given a set of biological sequences, find instances of a short site that occur more often than you would expect by chance, with no *a priori* knowledge about the site.

Given a collection of k such instances (ignoring, for the moment, how to find them), this induces a profile A as described in Section 7.1. As usual, we compute a profile B from the background distribution.

From A and B, we can compute $D_2(A||B)$ as in Section 8.3, and use that as a measure of how good the collection is. The goal is to find the collection that maximizes $D_2(A||B)$. In particular, if we are looking for unknown *binding* sites, then the argument of Section 9.2 suggests that a relative entropy around $\log_2 \Gamma$ would be encouraging.

A version of the computational problem, then, is to take as inputs k sequences and an integer n, and output one length n substring from each input sequence, such that the resulting relative entropy is maximized. Let us call this the *relative entropy site selection problem*. Unfortunately, this problem is likely to be computationally intractable (Section 6.4):

Theorem 9.1 (Akutsu [1, 2]): The relative entropy site selection problem is NP-complete.

Akutsu also proved that selecting instances so as maximize the sum-of-pairs score (Section 6.2) rather than the relative entropy is NP-complete.

Lecture 10

Finding Instances of Unknown Sites

February 8, 2000
Notes: Dylan Chivian

In order to find instances of unknown sites, we would like to be able to solve the relative entropy site selection problem (Section 9.3) exactly and efficiently. Unfortunately, Theorem 9.1 shows that the relative entropy site selection problem is NP-complete, so we are unlikely to find an algorithm that will compute an optimal solution efficiently. However, if we relax the optimality constraint, it may be possible to develop algorithms that compute "good" solutions efficiently. Because of the problem abstraction required to model the biological problem mathematically, the mathematically optimal solution need not necessarily be the most biologically significant. Lower scoring solutions are potentially the "correct" answer in their biological context. Therefore, giving up on the mathematical optimality of solutions to the relative entropy site selection problem seems the right compromise.

As an example of a typical application of finding instances of unknown sites, consider the genes involved in digestion in yeast. It is likely that many of these genes have some transcription factors in common, and therefore similarities in their promoter regions. Applying the site selection problem to the 1Kb DNA sequences upstream of known digestion genes may well yield some of these transcription factor binding sites. As another example, we could use the site selection problem to find common motifs in a protein family.

As defined, the relative entropy site selection problem limits its solution to contain exactly one site per input sequence, which may not be realistic in all applications. In some applications, there may be zero or many such sites in some of the input sequences. The algorithms discussed below are described in terms of the single site assumption, but can be modified to handle the general case as well.

But in the context of this general case, this is a good point at which to consider the effects on relative entropy of increasing either the number of sites or the length of each site. Increasing the number of sites will not increase the relative entropy, which is a function only of the fraction $P(s)$ of sites containing each residue s, and not the absolute number of such sites. For instance, a perfectly conserved position has $P(s) = 1$, regardless of whether it is present in all 10 sites or all 100 sites. This aspect of relative entropy is both a strength and a weakness. The strength is that it measures the degree of conservation, but the weakness is that we would like the measure to increase with more instances of a conserved residue.

However, increasing the length n of each site *does* increase the relative entropy, as it is additive and always nonnegative (Theorem 8.8). If comparing relative entropies of different length sites is important, one may normalize by dividing by the length n of the site or, alternatively, subtracting the expected relative entropy from each position.

10.1. Greedy Algorithm

Hertz and Stormo [23] described an efficient algorithm for the relative entropy site selection problem that uses a "greedy" approach. Greedy algorithms pick the locally best choice at each step, without concern for the impact on future choices. In most applications, the greedy method will result in solutions that are far from optimal, for some input instances. However, it does work efficiently, and may produce good solutions on many of its input instances.

Hertz and Stormo's algorithm for the relative entropy site selection problem proceeds as follows. The user specifies the length n of sites. The user also specifies a maximum number d of profiles to retain at each step. Profiles with lower relative entropy scores than the top d will be discarded; this is precisely the greedy aspect of the algorithm.

INPUT: sequences $s_1, s_2, \ldots, s_k$, and n, d, and the background distribution.

ALGORITHM:

1. Create a singleton set (i.e., only one member) for each possible length n substring of each of the k input sequences.
2. For each set S retained so far, add each possible length n substring from an input sequence s_i not yet represented in S. Compute the profile and relative entropy with respect to the background for each new set. Retain the d sets with the highest relative entropy.
3. Repeat step 2 until each set has k members.

A small example from Hertz and Stormo [23] is shown in Figure 10.1. From this example it is clear that pruning the number of sets to d is crucial, in order to avoid the exponentially many possible sets. The greedy nature of this pruning biases the selection from the remaining input sequences. High scoring profiles chosen from the first few sequences may not be well represented in the remaining sequences, whereas medium scoring profiles may be well represented in most of the k sequences, and thus would have yielded superior scores.

Note that one may modify the algorithm to circumvent the assumption of a single site per sequence, by permitting multiple substrings to be chosen from the same sequence. In this case, a different stopping condition is needed.

Hertz and Stormo applied their technique to find CRP binding sites (see Section 7.1) with some success. With 18 genes containing 24 known CRP binding sites, their best solution contained 19 correct sites, plus 3 more that overlap correct sites.

10.2. Gibbs Sampler

Lawrence *et al.* [29] developed a different approach to the relative entropy site selection problem based on "Gibbs sampling". The idea behind this technique is to *start* with a complete set of k substrings (candidate sites), from which we iteratively remove one at random, and then add a new one at random with probability proportional to its score, hopefully resulting in an improved score. In the following description, we again make the assumption that we choose one site per input sequence, but this method also can be extended to permit any number of sites per sequence.

INPUT: sequences $s_1, s_2, \ldots, s_k$, n, and the background distribution.

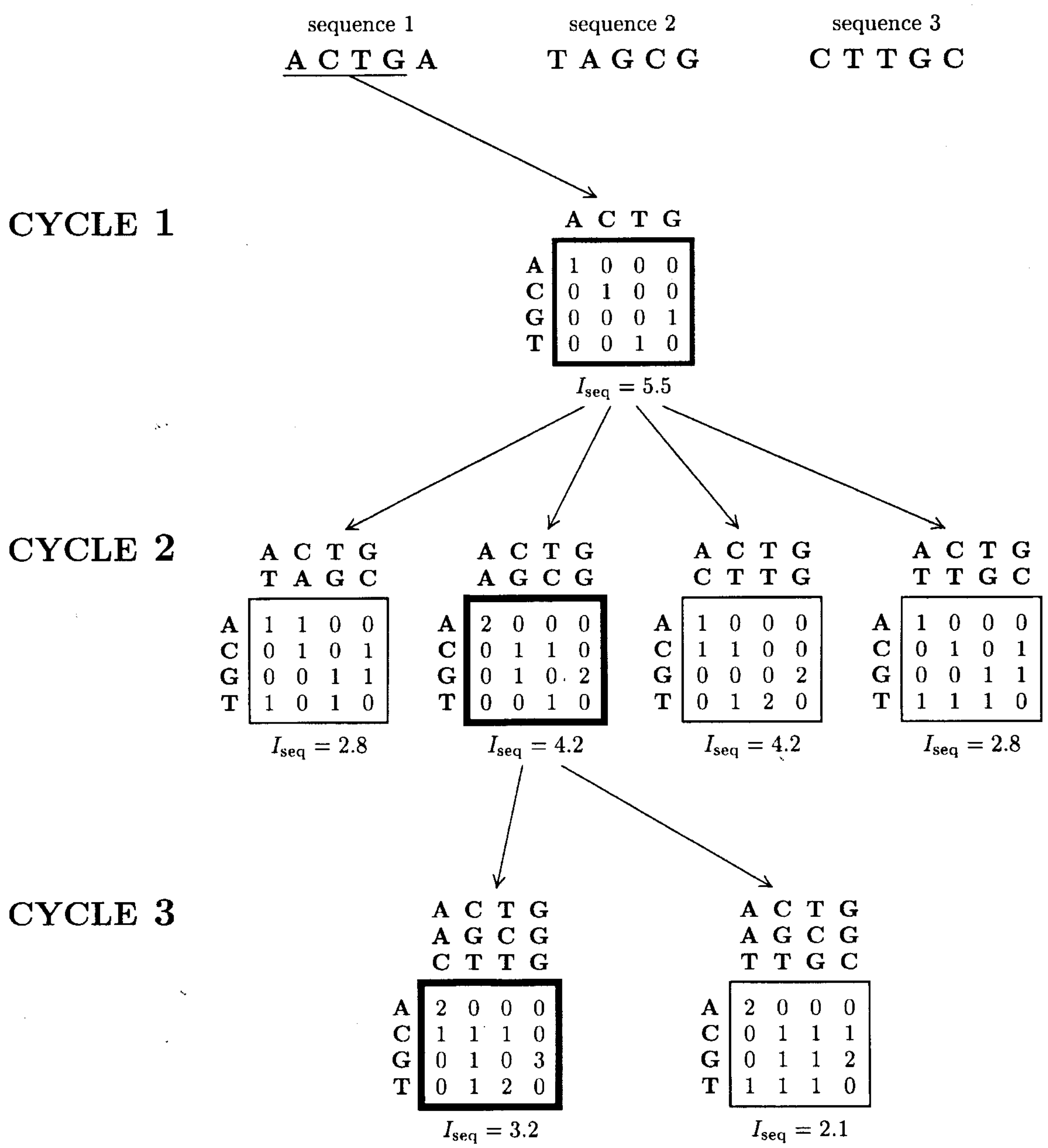

Figure 10.1: Example of Hertz and Stormo's greedy algorithm. I_{seq} denotes the relative entropy.

ALGORITHM: Initialize set T to contain substrings $t_1, t_2, \ldots, t_k$, where t_i is a substring of s_i chosen randomly and uniformly. Now perform a series of iterations, each of which consists of the following steps:

1. Choose i randomly and uniformly from $\{1, 2, \ldots, k\}$ and remove t_i from T.
2. For every j in $\{1, 2, \ldots, |s_i| - n + 1\}$:
   (a) Let t_{ij} be the length n substring of s_i that starts at position j.

   (b) Compute D_j, the relative entropy of $T \cup \{t_{ij}\}$ with respect to the background.
   (c) Let $P_j = D_j / \sum_h D_h$.

3. Randomly choose t_i to be t_{ij} with probability P_j, and add t_i to T.

We iterate until a stopping condition is met, either a fixed number of iterations or relative stability of the scores, and return the best solution set T seen in all iterations. The hope with this approach is that the random choices help to avoid some of the local optima of greedy algorithms.

Note that the Gibbs sampler may discard a substring that yields a higher scoring profile than the one that replaces it, or may restore the substring that was discarded itself. Neither of these occurrences is particularly significant, since the sampling will tend toward higher scoring profiles due to the probabilistic weighting of the substitutions by relative entropy. The Gibbs sampler does retain some degree of greediness (which is desirable), so that there may be cases where a strong signal in only a few sequences incorrectly outweighs a weaker signal in all of the sequences.

Lawrence *et al.* applied their technique to find motifs in protein families. In particular, they successfully discovered a helix-turn-helix motif, as well as motifs in lipocalins and prenyltransferases.

10.3. Other Methods

Possible extensions to the Gibbs sampler technique of Section 10.2 include the following:

1. Weight which t_i to discard in step 1 (analogously to weighting which to add in step 3).
2. Use simulated annealing (see, for example, Johnson *et al.* [25]) where, as time progresses, the probability decreases that you make a substitution that worsens the relative entropy score, yielding a more stable set T.

Another technique that has been used to solve the site selection problem is “expectation maximization” (for example, in the MEME system [4]).

Lecture 11

Correlation of Positions in Sequences

February 10, 2000
Notes: Tammy Williams

This lecture explores the validity of the assumption that the residues appearing at different positions in a sequence are independent. In previous lectures the computations assumed such positional independence. (See Section 7.2.) Here we describe a method to determine the level of dependence among residues in a sequence. By calculating the relative entropy of two models, one modeling dependence and the other modeling independence of positions, we can quantify the validity of the positional independence assumption.

Most of the material for this lecture is from Phil Green's MBT 599C lecture notes, Autumn 1996.

11.1. Nonuniform Versus Uniform Distributions

We will begin with a warmup to the method that still assumes positional independence, and proceed to the dependence question in Section 11.2. Given the genome of an organism, a simple calculation determines the frequency of each nucleotide. It is reasonable to suspect that these frequencies are more informative than the uniform nucleotide distribution, in which the probability of each nucleotide is 0.25. One can compare the frequency distribution (the nonuniform distribution in which the probability of residue r is equal to the frequency of residue r in the genome as a whole) to the uniform distribution.

Example 11.1: This example calculates the relative entropy of the frequency distribution to the uniform distribution for the archaeon *M. jannaschii*. *M. jannaschii* is a *thermophilic* prokaryote, meaning that it lives in extremely high temperature environments such as thermal springs. The frequency distribution of residues for *M. jannaschii* is given in Table 11.1.

A: 0.344
C: 0.155
G: 0.157
T: 0.343

Table 11.1: The frequency distribution of residues in *M. jannaschii*

Notice that the frequencies of residues A and T are very similar but not equal. Likewise, the residues C and G have similar frequencies. When calculating these frequencies, only one strand of DNA was used. (Had both strands been used, base pair complementarity would have ensured that these frequencies would be exactly equal rather than just similar.) Because genes and other functional regions tend to occur on both strands of DNA equally often, any bias of such a region on one strand over the other (see Section 11.4) is canceled out. This phenomenon, together with the fact that the bases occur in complementary pairs, explains why the frequencies of A's and T's are similar and the frequencies of G's and C's are similar.

Let Q be the uniform probability distribution, and let P be the frequency distribution. Notice that the frequency of residue r is equal to the probability of randomly selecting residue r from the distribution P. How much better does P model the actual genome than Q? More quantitatively, how much more information is there using P rather than Q? Recall from Section 8.3 that the relative entropy is defined as follows:

$$D_b(P||Q) = \sum_{s \in S} P(s) \log_b \frac{P(s)}{Q(s)}.$$

In Example 11.1 for *M. jannaschii*, $D_2(P||Q) = 0.103$. This implies that there are 0.103 more bits of information per position in the sequence by using distribution P over distribution Q. The value 0.103 might seem insignificant, but it means that a sequence of 100 bases has ten bits of extra information when chosen according to distribution P. Suppose a random sequence s of length 100 is selected according to the probability distribution P. Since the relative entropy is the expected log likelihood ratio for s, the sequence s is approximately $2^{10} = 1024$ times more likely to have been generated by P than by Q. The mathematics leading to this observation is flawed, since the log function and expectation do not "commute": that is, for it to be correct we would need the expected log likelihood ratio to equal the log of the expected likelihood ratio, which is not true in general. However, the intuition is helpful.

The next section explores the application of this relative entropy method to the question of dependence of nucleotides.

11.2. Dinucleotide Frequencies

How much dependence is there between *adjacent* nucleotides in a DNA sequence? Since there are four nucleotides, there are 16 possible pairs of nucleotides. To calculate the frequencies of each such pair (i, j) in a sequence, a simple algorithm computes the total number of observed occurrences of i followed immediately by j, and divides by the total number of pairs, which is the length of the sequence minus one. Let P_{ij} be the frequency of the residue i immediately followed by the residue j. In addition let P_i be the frequency of residue i in the single nucleotide distribution. The value $P'_{ij} = \frac{P_{ij}}{P_i P_j}$ gives a score representing the validity of the positional independence assumption. If $P'_{ij} = 1$, then the independence assumption is valid for residue i followed by residue j. (See Definition 7.1.) As the deviation from one increases, the independence assumption becomes less valid.

Example 11.2: Let us return to Example 11.1 involving the organism *M. jannaschii*. The P'_{ij} values are given in Table 11.2 with the residue i indexed by row and the residue j indexed by column, where residue i immediately precedes j in the sequence. Upon examination of the table, one can see that there are sizable deviations from one. For example the pairs (C,C) and (G,G) occur much more often than expected if they were independent, and (A,C), (C,G), and (G,T) occur much less often. Also, the diagonal entries show that two consecutive occurrences of the same residue occur more often than expected. Such repeats of the same residue might result from the slippage of the DNA polymerase during the replication process (see Section 1.5). The DNA polymerase inserts an extra copy of the base or misses a copy while duplicating one of the DNA strands. Even though there is a post-replication repair system to repair mistakes produced by the DNA polymerase, there is a small chance that the repeats will persist after a copy mistake. (In a similar way, dinucleotide repeats might occur during the replication process.)

	A	C	G	T
A	1.13	0.73	1.10	0.94
C	1.03	1.37	0.32	1.11
G	1.05	1.12	1.39	0.71
T	0.83	1.05	1.03	1.14

Table 11.2: The ratios of the observed dinucleotide frequency to the expected dinucleotide frequency (assuming independence) in *M. jannaschii*

Definition 11.3: The *mutual information* of a pair (X, Y) of random variables is

$$I(X;Y) = \sum_x \sum_y Pr(X = x \ \&\ Y = y) \log_2 \frac{Pr(X = x \ \&\ Y = y)}{Pr(X = x)Pr(Y = y)}.$$

If the probability distribution P is the joint distribution of X and Y, and Q is the distribution of X and Y assuming independence, then $I(X;Y) = D_2(P||Q)$. By Theorem 8.8, then, $I(X;Y) \geq 0$, with equality if and only if X and Y are independent, since in the equality case $P = Q$.

By setting the random variable X to be the first base and Y to be the second base of a pair, the value $I(X;Y)$ for *M. jannaschii* is 0.03. For a sequence of 100 bases, there are three bits of extra information when the sequence is chosen from the dinucleotide frequency distribution rather than the independence model. Thus, a random sequence s of length 100 generated by a process according to dinucleotide distribution P is eight times more likely to have been generated by P than by the independent nucleotide distribution Q.

11.3. Disymbol Frequencies

A generalization of the dinucleotide frequency is called the *disymbol frequency*, in which the two positions are not restricted to be adjacent. For example, one could study the dependence relationship between pairs of nucleotides separated by ten positions. The extension of methods presented above to this generalized setting is straightforward. Studies have revealed that the mutual information between DNA nucleotides separated by more than one base is lower than for adjacent residues. In fact, for separations of length 2, 3, and 4, the mutual information is an order of magnitude less than for adjacent residues.

11.4. Coding Sequence Biases

A similar application of relative entropy is finding biases in coding sequences. Recall that coding sequences consist of codons that are three consecutive bases: see Section 1.6.2. Do the three positions each have the same distribution as the background distribution? If such statistical features of protein-coding regions are known, they can be exploited by algorithms that locate genes.

In the bacterium *H. influenzae*, the residues A and G are more likely to appear in the first position of codons than in the genomic background. Using an analysis analogous to that used in Sections 11.1 and 11.2, there are 0.082 bits of information in the first codon position relative to the background distribution for *H. influenzae*. Since most of the *H. influenzae* genome consists of coding regions, it makes little difference if the background distribution is measured genome-wide or coding-region-wide. There are 0.175 bits of information per residue in the first position of codons for *M. jannaschii*.

The total relative entropy for the entire codon is simply the sum of the relative entropies for the three positions. (See Section 8.3.) The number of bits per codon for the organisms *H. influenzae*, *M. jannaschii*, *C. elegans*, and *H. sapiens* are 0.12, 0.21, 0.09, and 0.12, respectively. For *H. influenzae* the number of bits of information for the second position is close to zero. In humans there is more information is in the second position.

11.4.1. Codon Biases

Recall from Section 1.6.2 that there are 64 possible mRNA sequences of length three, but there are only 20 amino acids plus the stop codon. Thus, there exist *synonymous codons* that encode the same amino acid. Another statistical clue for locating genes is whether an organism uses synonymous codons equally often or has a bias toward certain codons in its genome.

For example, in *H. influenzae* the codon TTT is used about four times as often as TTC, although both TTT and TTC encode the amino acid phenylalanine. One conjecture as to why this occurs is that the tRNA for TTT is more abundant than the tRNA for TTC. Recall from Section 2.2 that the tRNA carries an amino acid to the ribosome during translation. There is selective pressure on the organism to choose the codon that is most efficiently translated, which would be affected by tRNA abundance.

A similar study investigates if organisms prefer one amino acid over another, since some amino acids such as leucine and isoleucine are chemically similar. (See Section 1.1.1.)

11.4.2. Recognizing Genes

Codon bias can be applied to the problem of recognizing genes in a DNA sequence. Define a score for codon C as follow:

$$score(C) = \log_2 \frac{C_R}{C_B},$$

where C_R is the frequency of codon C in the coding regions and C_B is the frequency of codon C in the background.

The score of a sequence $C_1, C_2, ..., C_n$ of codons is defined to be the sum of the scores of each C_i. When recognizing genes, one facet would be to identify sequences with high scores. Each reading frame must be examined since moving the frame window to the right one position or two positions results in different sequences of codons.

One drawback to this technique is that we must know the coding regions (in order to estimate C_R) before recognizing (those same) genes in the genome. There are simpler methods for finding long coding regions, and once these are known they can be used to estimate C_R and thus used to find more genes.

Lecture 12

Maximum Subsequence Problem

February 15, 2000
Notes: Mathieu Blanchette

12.1. Scoring Regions of Sequences

We have studied a variety of methods to score a DNA sequence so that regions of interest obtain a high score. For example, Section 11.4.2 suggested codon bias as a means for finding coding regions. If C is a codon, the score associated with C is $\log_2 \frac{C_R}{C_B}$, where C_R is the frequency of C in known coding regions (in the correct reading frame), and C_B is the frequency of C in noncoding regions (usually taken as the background distribution).

Notice that our goal is to identify *new* coding regions, but the method requires that we already know some coding regions in order to estimate C_R. There are a few easy ways one can identify a subset of likely coding regions. First, one could look for long *open reading frames* (*ORFs*), that is, long contiguous reading frames without STOP codons. Since 3 of the 64 codons are STOP codons (see Table 1.1), in random sequences one would expect a STOP codons every 64/3 triplets, i.e., every 64 bases, if codons are distributed uniformly. Since most genes are at least hundreds of bases long, very long ORFs are likely to be coding regions. This method will work well if we assume that the new genome contains no introns (or at least many very long exons), and if the codon distribution in long genes is similar to that in all genes, so that we can use it to estimate C_R. Another easy way to find a training set of coding sequences is by sequence similarity: compare the sequence of interest with a genome in which many genes are known, and extract the regions with high sequence similarity to known genes.

Then, if we assume that different triplets in the sequence are independent, we would like to find contiguous stretches of triplets with high total score (and thus with high log likelihood ratio). These regions would be good candidates for coding regions, to be subjected to further testing.

Another relevant question is in which reading frame to look for codons. There are 6 possible reading frames: 3 on each of the 2 strands of DNA. When looking for coding region, one would search for high scoring regions in each of these 6 reading frames.

12.2. Maximum Subsequence Problem

We can distill the following general computational problem from the preceding discussion. We are given a sequence $X_1, X_2, \ldots, X_n$ of real numbers, where X_i corresponds to the score of the ith element of the sequence. The problem is to find a contiguous subsequence $X_i, X_{i+1}, \ldots, X_j$ that maximizes $X_i + X_{i+1} + \cdots + X_j$. We will call this a *maximum subsequence*. Note that, if all X_i's are nonnegative, the problem is not interesting, since the maximum subsequence will always be $X_1, X_2, \ldots, X_n$, so the interesting case is when some of the scores are negative.

The following algorithm for finding a maximum subsequence was given by Bates and Constable [5] and Bentley [7, Column 7].

Suppose we already knew that the maximum subsequence B of $X_1, X_2, \ldots, X_k$ has score b. How can we find the maximum subsequence of $X_1, X_2, \ldots, X_k, X_{k+1}$? If X_k is included in B, then it is easy: if $X_{k+1} > 0$, we will add X_k to B, and if not, we will leave B unchanged. But what if X_k is not included in B? In that case, in addition to B we will have to keep track of the score of the *maximum suffix* F of $X_1, X_2, \ldots, X_k$: F is the suffix $X_s, X_{s+1}, \ldots, X_k$ that maximizes $f = X_s + X_{s+1} + \cdots + X_k$. Let us assume that F is also known for $X_1, X_2, \ldots, X_k$. We are now given X_{k+1}, and we want to update B and F accordingly:

if $f + X_{k+1} > b$
  then add X_{k+1} to F and replace B by F
  else if $f + X_{k+1} > 0$
    then add X_{k+1} to F
    else reset F to be empty.

The complexity of the algorithm is $O(n)$, since a constant amount of work is done for every new element X_{k+1}, and there are n such elements.

12.3. Finding *All* High Scoring Subsequences

The algorithm described in Section 12.2 works very well if we are interested in finding *one* maximum subsequence. However, we are generally looking for *all* high scoring regions, for instance, all good candidates for coding regions. We could repeatedly use the previous algorithm to find them all: find the maximum subsequence, remove it, and repeat on the two remaining parts of the sequence. We will call the problem of finding exactly these disjoint maximum subsequences the *all maximum subsequences* problem. (In practice, one would only want to retain those reported maximum subsequences with scores sufficiently high to be interesting.)

The problem with repeatedly running the previous algorithm is that it will take $O(n)$ operations per subsequence reported, and thus possibly $O(n^2)$ operations to identify all high scoring regions. Intuitively, one might hope to do better, since much of the work done to find the first maximum subsequence could be reused to find the second one, and so on. We now present an algorithm that solves the all maximum subsequences problem in time $O(n)$, the same as the time to find just one maximum subsequence. This algorithm is due to Ruzzo and Tompa [40]. We first describe the algorithm, and then discuss its performance.

Algorithm. The algorithm reads the scores from left to right, and maintains the cumulative total of the scores read so far. Additionally, it maintains a certain ordered list $I_1, I_2, \ldots, I_{k-1}$ of disjoint subsequences. For each such subsequence I_j, it records the cumulative total L_j of all scores up to but not including the leftmost score of I_j, and the total R_j up to and including the rightmost score of I_j.

The list is initially empty. Input scores are processed as follows. A nonpositive score requires no special processing when read. A positive score is incorporated into a new subsequence I_k of length one[1] that is then

[1] In practice, one could optimize this slightly by processing a consecutive series of positive scores as I_k.

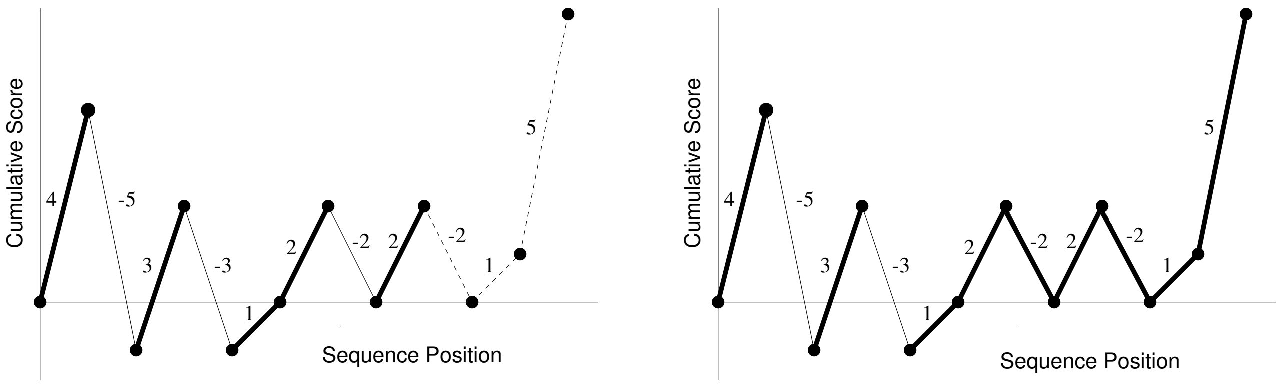

Figure 12.1: An example of the algorithm. Bold segments indicate score sequences currently in the algorithm's list. The left figure shows the state prior to adding the last three scores, and the right figure shows the state after.

integrated into the list by the following process.

1. The list is searched from right to left for the maximum value of j satisfying $L_j < L_k$.
2. If there is no such j, then add I_k to the end of the list.
3. If there is such a j, and $R_j \geq R_k$, then add I_k to the end of the list.
4. Otherwise (i.e., there is such a j, but $R_j < R_k$), extend the subsequence I_k to the left to encompass everything up to and including the leftmost score in I_j. Delete subsequences $I_j, I_{j+1}, \ldots, I_{k-1}$ from the list (none of them is maximum) and reconsider the newly extended subsequence I_k (now renumbered I_j) as in step 1.

After the end of the input is reached, all subsequences remaining on the list are maximum; output them.

As an example of the execution of the algorithm, consider the input sequence $(4, -5, 3, -3, 1, 2, -2, 2, -2, 1, 5)$. After reading the scores $(4, -5, 3, -3, 1, 2, -2, 2)$, suppose the list of disjoint subsequences is $I_1 = (4), I_2 = (3), I_3 = (1, 2), I_4 = (2)$, with $(L_1, R_1) = (0, 4)$, $(L_2, R_2) = (-1, 2)$, $(L_3, R_3) = (-1, 2)$, and $(L_4, R_4) = (0, 2)$. (See Figure 12.1.) At this point, the cumulative score is 2. If the ninth input is -2, the list of subsequences is unchanged, but the cumulative score becomes 0. If the tenth input is 1, Step 1 produces $j = 3$, because I_3 is the rightmost subsequence with $L_3 < 0$. Now Step 3 applies, since $R_3 \geq 1$. Thus $I_5 = (1)$ is added to the list with $(L_5, R_5) = (0, 1)$, and the cumulative score becomes 1. If the eleventh input is 5, Step 1 produces $j = 5$, and Step 4 applies, replacing I_5 by $(1, 5)$ with $(L_5, R_5) = (0, 6)$. The algorithm returns to Step 1 without reading further input, this time producing $j = 3$. Step 4 again applies, this time merging I_3, I_4, and I_5 into a new $I_3 = (1, 2, -2, 2, -2, 1, 5)$ with $(L_3, R_3) = (-1, 6)$. The algorithm again returns to Step 1, but this time Step 2 applies. If there are no further input scores, the complete list of maximum subsequences is then $I_1 = (4), I_2 = (3), I_3 = (1, 2, -2, 2, -2, 1, 5)$.

The fact that this algorithm correctly finds all maximum subsequences is not obvious; see Ruzzo and Tompa [40] for the details.

Analysis. There is an important optimization that may be made to the algorithm. In the case that Step 2 applies, $I_1, \ldots, I_{k-1}$ are maximum subsequences, and so may be output before reading any more of the input. Thus, Step 2 of the algorithm may be replaced by the following, which substantially reduces the memory requirements of the algorithm.

$2.'$ If there is no such j, all subsequences $I_1, I_2, \ldots, I_{k-1}$ are maximum. Output them, delete them from the list, and reinitialize the list to contain only I_k (now renumbered I_1).

The algorithm as given does not run in linear time, because several successive executions of Step 1 might re-examine a number of list items. This problem is avoided by storing with each subsequence I_k added during Step 3 a pointer to the subsequence I_j that was discovered in Step 1. The resulting linked list of subsequences will have monotonically decreasing L_j values, and can be searched in Step 1 in lieu of searching the full list. Once a list element has been bypassed by this chain, it will be examined again only if it is being deleted from the list, either in Step $2'$ or Step 4. The work done in the "reconsider" loop of Step 4 can be amortized over the list item(s) being deleted. Hence, in effect, each list item is examined a bounded number of times, and the total running time is linear.

The worst case memory complexity is also linear, although one would expect on average that the subsequence list would remain fairly short in the optimized version incorporating Step $2'$. Empirically, a few hundred stack entries suffice for processing sequences of a few million residues, for either synthetic or real genomic data.

Lecture 13

Markov Chains

February 17, 2000
Notes: Jonathan Schaefer

In Lecture 11 we discovered that correlations between sequence positions are significant, and should often be taken into account. In particular, in Section 11.4 we noted that codons displayed a significant bias, and that this could be used as a basis for finding coding regions. Lecture 12 then explored algorithms for doing exactly that.

In some sense, Lecture 12 regressed from the lesson of Lecture 11. Although it was using codon bias to score codons, it did not exploit the possible correlation between adjacent codons. Even worse, each codon was scored independently and the scores added, so that the codon score does not even depend on the position the codon occupies.

This lecture rectifies these shortcomings by taking codon correlation into account in predicting coding regions. In order to do so, we first introduce Markov chains as a model of correlation.

13.1. Introduction to Markov Chains

The end of Section 11.2 mentioned "…a random sequence …generated by a process according to dinucleotide distribution P", without giving any indication of what such a random process might look like. Such a random process is called a "Markov chain", and is more complex than a process that draws successive elements independently from a probability distribution. The definition of Markov chain will actually generalize dinucleotide dependence to the case in which the identity of the current residue depends on the previous k residues, rather than just the previous one.

Definition 13.1: Let S be a set of states (e.g., $S = \{A, C, G, T\}$). Let $(X_0, X_1, X_2, \ldots)$ be a sequence of random variables, each with sample space S. A *kth order Markov chain* satisfies

$$\Pr(X_t = x \mid X_0, X_1, \ldots, X_{t-1}) = \Pr(X_t = x \mid X_{t-k}, X_{t-k+1}, \ldots, X_{t-1})$$

for any t and any $x \in S$.

In words, in a kth order Markov chain, the distribution of X_t depends only on the k variables immediately preceding it. In a 1st order Markov chain, for example, the distribution of X_t depends only on X_{t-1}. Thus, a 1st order Markov chain models diresidue dependencies, as discussed in Section 11.2. A 0th order Markov chain is just the familiar independence model, where X_t does not depend on any other variables.

Markov chains are not restricted to modeling positional dependencies in sequences. In fact, the more usual applications are to time dependencies, as in the following illustrative example.

Example 13.2: This example is called "a random walk on the infinite 2-dimensional grid". Imagine an infinite grid of streets and intersections, where all the streets run either east-west or north-south. Suppose you are trying to find a friend who is standing at one specific intersection, but you are lost and all street signs are missing. You decide to use the following algorithm: if your friend is not standing at your current intersection, choose one of the four directions (N, E, S, or W) randomly and uniformly, and walk one block in that direction. Repeat until you find your friend.

This is an example of a 1st order Markov chain, where each intersection is a state, and X_t is the intersection where you stand after t steps. Notice that the distribution of X_t depends only on the value of X_{t-1}, and is completely independent of the path that you took to arrive at X_{t-1}.

Definition 13.3: A kth order Markov chain is said to be *stationary* if, for all t and u,

$$\Pr(X_t = x \mid X_{t-k}, X_{t-k+1}, \ldots, X_{t-1}) = \Pr(X_u = x \mid X_{u-k}, X_{u-k+1}, \ldots, X_{u-1}).$$

That is, in a stationary Markov chain, the distribution of X_t is independent of the value of t, and depends only on the previous k variables. The random walk of Example 13.2 is an example of a stationary 1st order Markov chain.

13.2. Biological Application of Markov Chains

Markov chains can be used to model biological sequences. We will assume a directional dependence and always work in one direction, for example, from $5'$ to $3'$, or N-terminal to C-terminal.

Given a sequence s, and given a Markov chain M, a basic question to answer is, "What is the probability that the sequence s was generated by the Markov chain M?" For instance, if we were modeling diresidue dependencies with a 1st order Markov chain M, we would need to be able to determine what probabilities M assigns to various sequences.

Consider, for simplicity, a stationary 1st order Markov chain M. Let $A_{r,s} = \Pr(X_t = s \mid X_{t-1} = r)$. A is called the *probability transition matrix* for M. The dimensions of the matrix A are $|S| \times |S|$, where S is the state space. For nucleotide sequences, for example, A is 4×4.

Then the probability that the sequence $s = (s_0, s_1, \ldots, s_t)$ was generated by M is

$$\Pr(s_0 s_1 \ldots s_t) = \Pr(s_0) A_{s_0,s_1} A_{s_1,s_2} \cdots A_{s_{t-1},s_t} = \Pr(s_0) \prod_{i=1}^{t} A_{s_{i-1},s_i}.$$

In this equation,

1. $\Pr(s_0)$ is estimated by the frequency of s_0 in the genome, and
2. $A_{r,s}$ is estimated by $N_{r,s}/N_r$, where $N_{r,s}$ is the number of occurrences in the genome of the diresidue (r, s), and $N_r = \sum_i N_{r,i}$.

Markov chains have some weaknesses as models of biological sequences:

1. Unidirectionality: the residue s_i is equally dependent on both s_{i-1} and s_{i+1}, yet the Markov chain only models its dependence on the k residues on one side of s_i.

2. Mononucleotide repeats are not adequately modeled. They are much more frequent in biological sequences than predicted by a Markov chain. This frequency is likely due to DNA polymerase slippage during replication, as discussed in Example 11.2.

3. Codon position biases (as discussed in Section 11.4) are not accurately modeled.

13.3. Using Markov Chains to Find Genes

We will consider two gene finding algorithms, GeneMark [8, 9] and Glimmer [13, 41]. Both are commonly used to find intron-free protein-coding regions (usually in prokaryotes), and both are based on the ideas of Markov chains. As in Section 12.1, both assume that a training set of coding regions is available, but unlike that method, the training set is used to train a Markov chain.

GeneMark [8, 9] uses a kth order Markov chain to find coding regions, where $k = 5$. This choice allows any residue to depend on all the residues in its codon and the immediately preceding codon.

The training set consists of coding sequences identified by either long open reading frames or high sequence similarity to know genes.

Three separate Markov chains are constructed from the training set, one for each of the three possible positions in the reading frame. For any one of these reading frame positions, the Markov chain is built by tabulating the frequencies of all $(k + 1)$-mers (that is, all length $k + 1$ substrings) that end in that reading frame position. These three Markov chains are then alternated to form a single nonstationary kth order Markov chain M that models the training set.

Given a candidate ORF x, we can compute the probability p that x was generated by M, as described in Section 13.2. This ORF will be selected for further consideration if p is above some predetermined threshold. The "further consideration" will deal with possible pairwise overlaps of such selected ORFs, in a way to be described in the next lecture.

Lecture 14

Using Interpolated Context Models to Find Genes

February 22, 2000
Notes: Gretta Bartels

14.1. Problems with Markov Chains for Finding Genes

The Markov chain is an effective model for finding genes, as described in Section 13.3. However, such a tool is not 100% accurate. The problem is that a kth order Markov chain requires 4^{k+1} probabilities in each of three reading frame positions. There is a tension between needing k large to produce a good gene model, and needing k small because there is insufficient data in the training set. For example, when $k = 5$ as in GeneMark, we need $3 \times 4^6 \approx$ 12,000 6-mers to build the model. Each of these 12,000 6-mers must occur often enough in the training set to support a statistically reliable sample.

Some 5-mers are too infrequent in microbial training sets, yet some 8-mers are frequent enough to be statistically reliable. Section 14.2 describes a gene finder that was designed to have the flexibility to deal with these extremes.

14.2. Glimmer

Glimmer [13, 41] is a gene prediction tool that uses a model somewhat more general than a Markov chain. In particular, Glimmer 2.0 [13] uses what the authors call the *interpolated context model* (ICM).

The *context* of a particular residue consists of the k characters immediately preceding it. A typical context size might be $k = 12$. For context $C = b_1 b_2 \ldots b_k$, the interpolated context model assigns a probability distribution for b_{k+1}, using only as many residues from C as the training data supports. Furthermore, those residues need not be consecutive in the context.

Glimmer has three phases for finding genes: training, identification, and resolving overlaps.

14.2.1. Training Phase

As in Sections 12.1 and 13.3, Glimmer uses long ORFs and sequences similar to known genes from other organisms as training data for the model. For each of the three reading frame positions, consider all $(k+1)$-mers that end in that reading frame position. Let X_i be a random variable whose distribution is given by the frequencies of the residues in position i of these $(k+1)$-mers.

In general, we will not have sufficient training data to use all k residues from this context to predict the $(k+1)$st residue b_{k+1}. Our goal is to determine which variable X_j has most correlation with X_{k+1}, and use

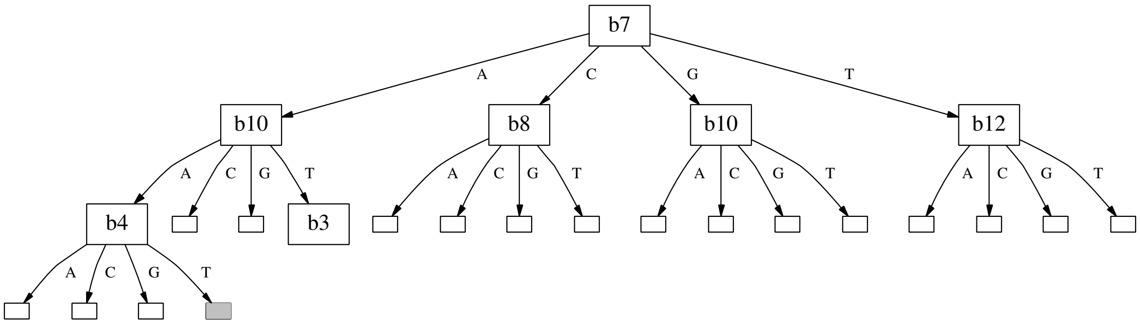

Figure 14.1: Interpolated context model tree.

it to predict b_{k+1}. The mutual information of Definition 11.3 is used to make this determination. We first find the maximum among the mutual information values

$$I(X_1; X_{k+1}), I(X_2; X_{k+1}), \ldots, I(X_k; X_{k+1}).$$

Suppose X_j maximizes this mutual information. Then the jth residue b_j will be used first to predict the value of b_{k+1}.

To determine which position has the next highest correlation, we do not simply take the second highest mutual information from the list above. The identity of the next position instead depends on the value of the first selected residue b_j. Glimmer builds a tree of influences on X_{k+1}, illustrated in Figure 14.1, as follows. The $(k+1)$-mers from this reading frame position are partitioned into four subsets according to the residue b_j. Then we repeat the mutual information calculation above for each of these subsets. In the example of Figure 14.1, X_7 was found to have the greatest mutual information with X_{k+1}, and the $(k+1)$-mers were partitioned according to the value of residue b_7. For those with b_7 = A, X_{10} was found to have the greatest mutual information with X_{k+1}, and they were further partitioned into four subsets according to the value of residue b_{10}.

A branch is terminated when the remaining subset of $(k+1)$-mers becomes too small to support further partitioning. Each such leaf of the tree is labeled with the probability distribution of X_{k+1}, given the residue values along the path from the root to that leaf. For example, in the tree shown in Figure 14.1, the leaf shaded gray would be labeled with the distribution

$$Pr(X_{k+1} = x \mid X_7 = A \;\&\; X_{10} = A \;\&\; X_4 = T).$$

Note how this tree generalizes the notion of Markov chain given in Definition 13.1.

14.2.2. Identification Phase

Once the interpolated context model has been trained, the identification phase begins. Given a candidate ORF x, compute the probability of each residue b_{k+1} of x by following the appropriate path in the tree that corresponds to b_{k+1}'s position in the reading frame. Here is a rough algorithm for the identification phase:

1. Pick the tree for the correct reading frame position.

2. Number the residues in the context of b_{k+1}, so that the previous residue is b_k, the one before that b_{k-1}, and so on.

Figure 14.2: Overlapping gene candidates. The arrow points to the $3'$ end of the sequence.

3. Trace down the tree, selecting the edges according to the particular residues in the sequence, until reaching a leaf. Read the probability $\Pr(X_{k+1} = b_{k+1} \mid \ldots)$ from that leaf.

4. Shift to the next residue in the sequence and repeat.

The product of these probabilities (times the probability of the first k residues of x, as in Section 13.2) yields the probability that x was generated by this interpolated context model. To take the length of x into account, combine these probabilities by adding their logarithms, and normalize by dividing by the length of x. Select x for further investigation if this score is above some predetermined threshold.

Note: Glimmer actually stores a probability distribution for X_{k+1} at every node of the tree, not just the leaves, and uses a combination of the distributions along a branch to predict b_{k+1}. This is where the modifier "interpolated" would enter, but we will not discuss this added complication. See the papers [13, 41] for details.

14.2.3. Resolving Overlap

In the final phase, Glimmer resolves candidate gene sequences that overlap. The main flexibility in resolving overlap is the possibility of shortening an ORF by choosing a different start codon. Suppose sequences A and B overlap, and that A has the greater score. There are four possibilities for the way they overlap, as depicted in Figure 14.2. Each arrow in that figure points to the $3'$ end of the sequence.

(a) Suppose A and B overlap as shown in Figure 14.2(a). Moving either start codon cannot eliminate the overlap in this case. If A is significantly longer than B, then reject B. Otherwise, accept both A and B with an annotation that they have a suspicious overlap.

(b) Suppose A and B overlap as shown in Figure 14.2(b). If moving B's start codon resolves the overlap in such a way that B still has great enough score and great enough length, accept both. If not, proceed as in Case (a).

(c) Suppose A and B overlap as shown in Figure 14.2(c). If the score of the overlap is a small fraction of A's score, and moving A's start codon resolves the overlap in such a way that A still has great enough length, accept both. Otherwise reject B.

(d) Suppose A and B overlap as shown in Figure 14.2(d). Move B's start codon until the overlap scores higher for B than for A. Then move A's start codon until the overlap scores higher for A than for B. Repeat until there is no more overlap, and accept both.

If there are more than two overlapping sequences, treat the overlaps in decreasing order of score. In this way, if A overlaps B and B overlaps C, and A has the highest score, then B may be rejected before its overlap with C would cause C to be rejected.

Lecture 15

Start Codon Prediction

February 24, 2000
Notes: Mingzhou Song

15.1. Experimental Results of Glimmer

The experimental results of Glimmer were presented by Delcher *et al.* [13]. They used the method described in Section 14.2 to predict genes in ten sequenced microbial genomes. The procedure was automated so as not to require human intervention. For each of the ten microbial genomes, the procedure was as follows.

In the training phase, they constructed a training set consisting of all ORFs longer than 500 bp with no overlap. The authors state that this set has more than enough data to train the interpolated context model accurately. The then trained the interpolated context model on the training set, as described in Section 14.2.1.

The identification phase and overlap resolution were then carried out as described in Sections 14.2.2 and 14.2.3. In each of the ten genomes, 99% of the annotated genes were correctly identified. The authors did not mention whether the start codons had been correctly identified in all cases.

Glimmer thus achieved a false negative rate of 1%, but also a false positive rate of 7–25% on each genome. The false negative rate is the percentage of annotated genes that were not identified by Glimmer. The false positive rate is the percentage of ORFs identified by Glimmer as genes, but not so annotated in the database. Of course, some of the annotations could be incorrect.

15.2. Start Codon Prediction

The accurate prediction of the translation start site, that is, the correct start codon, is important in order to analyze the putative protein product of a gene. Given the quality of the rest of the process, accurate start codon prediction is the most difficult remaining part of prokaryotic gene prediction. The gene-finding techniques discussed so far do little to predict the correct start codon among all the candidates.

Among the possible start codon candidates, what extra evidence can be used to identify the true translation start site? Recall from Section 2.2 that the ribosome is the structure that translates mRNA into protein and, at the initiation of that translation, is responsible for identifying the true translation start site. How does the ribosome perform this identification? Can we improve start codon prediction by mimicking the ribosome's process?

At the initiation of protein synthesis, the ribosome binds to the mRNA at a region near the $5'$ end of the mRNA called the *ribosome binding site*. This is a region of approximately 30 nucleotides of the mRNA that is protected by the ribosome during initiation. The ribosome binding site is approximately centered on the

Bacillus subtilis	5′ . . . CUGGAUCACCUCCUUUCUA 3′
Lactobacillus delbrueckii	5′ . . . CUGGAUCACCUCCUUUCUA 3′
Mycoplasma pneumoniae	5′ . . . GUGGAUCACCUCCUUUCUA 3′
Mycobacterium bovis	5′ . . . CUGGAUCACCUCCUUUCU 3′
Aquifex aeolicus	5′ . . . CUGGAUCACCUCCUUUA 3′
Synechocystis sp.	5′ . . . CUGGAUCACCUCCUUU 3′
Escherichia coli	5′ . . . UUGGAUCACCUCCUUA 3′
Haemophilus influenzae	5′ . . . UUGGAUCACCUCCUUA 3′
Helicobacter pylori	5′ . . . UUGGAUCACCUCCU 3′
Archaeoglobus fulgidus	5′ . . . CUGGAUCACCUCCU 3′
Methanobacterium thermoautotrophicum	5′ . . . CUGGAUCACCUCCU 3′
Pyrococcus horikoshii	5′ . . . CUCGAUCACCUCCU 3′
Methanococcus jannaschii	5′ . . . CUGGAUCACCUCC 3′
Mycoplasma genitalium	5′ . . . GUGGAUCACCUC 3′

Table 15.1: 3′ end of 16S rRNA for various prokaryotes

start codon (usually AUG). That is, the ribosome binding site contains not only the first few codons to be translated, but also part of the 5′ untranslated region of the mRNA (see Section 2.3).

The ribosome identifies where to bind to the mRNA at initiation not only by recognizing the start codon, but also by recognizing a short sequence in the 5′ untranslated region within the ribosome binding site. This short mRNA sequence will be called the *SD site*, for reasons that will become clear below. The mechanism by which the ribosome recognizes the SD site is relatively simple base-pairing: the SD site is complementary to a short sequence near the 3′ end of the ribosome's *16S rRNA*, one of its ribosomal RNAs.

The SD site was first postulated by Shine and Dalgarno [44] for *E. coli*. Subsequent experiments demonstrated that the SD site in *E. coli* mRNA usually matches at least 4 or 5 consecutive bases in the sequence AAGGAGG, and is usually separated from the translation start site by approximately 7 nucleotides, although this distance is variable. Numerous other researchers such as Vellanoweth and Rabinowitz [49] and Mikkonen *et al.* [34] describe very similar SD sites in the mRNA of other prokaryotes. It is not too surprising that SD sites should be so similar in various prokaryotes, since the 3′ end of the 16S rRNA of all these prokaryotes is well conserved (Mikkonen *et al.* [34]). Table 15.1 shows a number of these rRNA sequences. Note their similarity, and in particular the omnipresence of the sequence CCUCCU, complementary to the Shine-Dalgarno sequence AGGAGG.

This SD site can be used to improve start codon prediction. The simplest way to identify whether a candidate start codon is likely to be correct is by checking for approximate base pair complementarity between the 3′ end of the 16S rRNA sequence and the DNA sequence just upstream of the candidate codon. We say "approximate" complementarity because the ribosome just needs sufficient binding energy between the 16S rRNA and the mRNA, not necessarily perfect complementarity.

Several papers do use this SD site information to improve translation start site prediction. These papers are described briefly below.

Hayes and Borodovsky [22] found candidate SD sites by running a Gibbs sampler (Section 10.2) on the DNA sequences just upstream of a given genome's purported start codons. They then used the 3′ end of the genome's annotated 16S rRNA sequence to validate the SD site so found.

Frishman *et al.* [16] used a greedy version of the Gibbs sampler to find likely SD sites. In addition, they took into account the distance from the SD site to the start codon, which should be about 7 bp.

Hannenhalli *et al.* [21] used multiple features to score potential start codons. The features used were the following:

1. the binding energy between the SD site and the $3'$ end of the 16S rRNA, allowing "bulges" (that is, insertions and deletions) in the binding,
2. the identity of the start codon (AUG, UUG, or GUG),
3. coding potential downstream from the start codon and noncoding potential upstream, using GeneMark's scoring function (Section 13.3),
4. the distance from SD site to start codon, and
5. the distance from the start codon to the maximal start codon, which is as far upstream in this ORF as possible.

They took the score of any start codon to be a weighted linear combination of the scores on these five features. The coefficients of the linear combination were obtained using mixed integer programming.

15.3. Finding SD Sites

Do all prokaryotes have SD sites very similar to the Shine-Dalgarno sequences of *E. coli*? Given the collection of DNA sequences upstream from its putative genes, how can we identify a prokaryote's SD site, without reliance on the annotation of its 16S rRNA?

Tompa [48] proposed a method to discover SD sites by looking for statistically significant patterns (or *motifs*) in the sequences upstream from the putative genes. The method is reminiscent of the relative entropy site selection problem of Lecture 10 but, unlike the algorithms discussed there, this one is exhaustive, and guaranteed to find the most statistically significant motif. The statistical significance is measured by the "z-score", defined below. The sites with the highest z-scores are very unlikely to be from the background and very likely to be potential SD sites.

For each possible k-mer s, this approach takes into account both the absolute number N_s of upstream sequences containing (an approximation of) s, and the background distribution. It then calculates the unlikelihood of seeing N_s such occurrences, if the sequences had been drawn at random from the background distribution. The random process used in this calculation is a 1st order Markov chain based on the sequences' dinucleotide frequencies. (See Section 13.2.)

The measure of unlikelihood used is based on the z-score, defined as follows. Let N be the number of upstream sequences that are input, and p_s the probability that a single random upstream sequence contains at least one occurrence of (an approximation of) s. (See Tompa [48] for a description of how to compute p_s.) Then Np_s is the expected number of input sequences containing s, and $\sqrt{Np_s(1-p_s)}$ is its standard deviation. The *z-score* is defined as

$$z_s = \frac{N_s - Np_s}{\sqrt{Np_s(1-p_s)}}.$$

The measure z_s is the number of standard deviations by which the observed value N_s exceeds its expectation, and is sometimes called the "normal deviate" or "deviation in standard units". See Leung *et al.* [30] for a detailed discussion of this statistic. The measure z_s is normalized to have mean 0 and standard deviation 1, making it suitable for comparing different motifs s.

The algorithm was run on fourteen prokaryotic genomes. Those motifs with highest z-score showed a strong predominance of motifs complementary to the $3'$ end of their genome's 16S rRNA. For the bacteria, these were usually a standard Shine-Dalarno sequence consisting of 4–5 consecutive bases from AAGGAGG. For the thermophilic archaea *A. fulgidus*, *M. jannaschii*, *M. thermoautotrophicum*, and *P. horikoshii*, however, the significant SD sites uncovered were somewhat different. What is interesting about these is that their highest scoring sequences display a predominance of the pattern GGTGA or GGTG, which satisfies the requirement of complementarity to a substring near the $3'$ end of the 16S rRNA (see Table 15.1). However, that 16S substring is shifted a few nucleotides upstream compared to the bacterial sites discussed above.

Lecture 16

RNA Secondary Structure Prediction

February 29, 2000
Notes: Matthew Cary

16.1. RNA Secondary Structure

Recall from Section 1.3 that RNA is usually single-stranded in its "normal" state, and this strand folds into a functional shape by forming intramolecular base pairs among some of its bases. (See Figure 16.1 for an illustration.) The geometry of this base-pairing is known as the "secondary structure" of the RNA.

When RNA is folded, some bases are paired with other while others remain free, forming "loops" in the molecule. Speaking qualitatively, bases that are bonded tend to stabilize the RNA (i.e., have negative free energy), whereas unpaired bases form destabilizing loops (positive free energy). Through thermodynamics experiments, it has been possible to estimate the free energy of some of the common types of loops that arise.

Because the secondary structure is related to the function of the RNA, we would like to be able to predict the secondary structure. Given an RNA sequence, the *RNA Folding Problem* is to predict the secondary structure that minimizes the total free energy of the folded RNA molecule.

The prediction algorithm that will be described is by Lyngsø *et al.* [33].

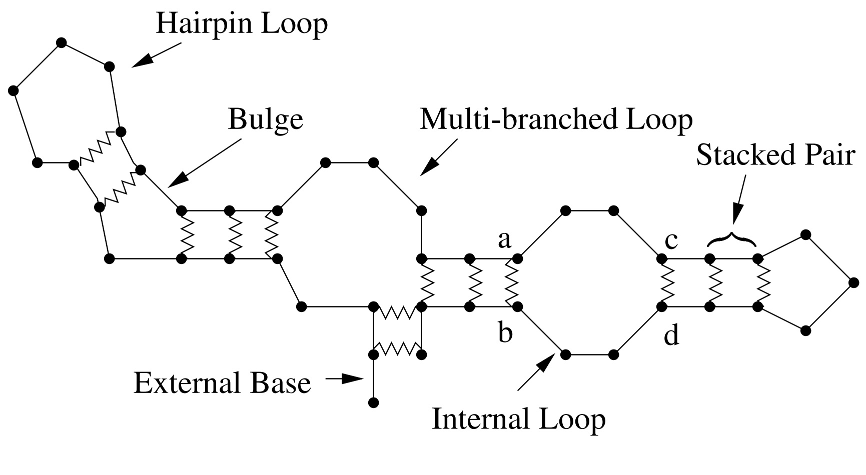

Figure 16.1: RNA Secondary Structure. The solid line indicates the backbone, and the jagged lines indicate paired bases.

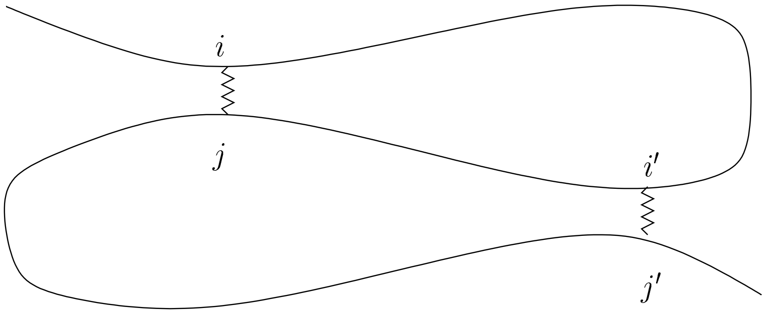

Figure 16.2: A Pseudoknot

16.2. Notation and Definitions

If $s = s_1 s_2 \ldots s_n$ is an RNA sequence and $1 \leq i < j \leq n$, then $i \cdot j$ denotes the base-pairing of s_i with s_j.

Definition 16.1: A *secondary structure* of $s = s_1 s_2 \ldots s_n$ is a set S of base pairs such that each base is paired at most once. More precisely, for all $i \cdot j \in S$ and $i' \cdot j' \in S$, $i = i'$ if and only if $j = j'$.

The configuration shown in Figure 16.2 is known as a *pseudoknot*. For the prediction algorithm that follows, we will assume that the secondary structure does not contain any pseudoknots. The ostensible justifications for this are that pseudoknots do not occur as often as the more common types of loops, and secondary structure prediction is moderately successful even if pseudoknots are prohibited. However, the real justification for this assumption is that it greatly simplifies the model and algorithm. (Certain types of pseudoknots are handled by the algorithm of Rivas and Eddy [38], but the general problem was shown NP-complete by Lyngsø and Pedersen [32].)

Definition 16.2: A *pseudoknot* in a secondary structure S is a pair of base pairs $i \cdot j \in S$ and $i' \cdot j' \in S$ with $i < i' < j < j'$.

16.3. Anatomy of Secondary Structure

Given the assumption of no pseudoknots, the secondary structure can be decomposed into a few types of simple loops, described as follows and illustrated in Figure 16.1.

Definition 16.3:

- A *hairpin loop* contains exactly one base pair.
- An *internal loop* contains exactly two base pairs.
- A *bulge* is an internal loop with one base from each of its two base pairs adjacent on the backbone.
- A *stacked pair* is a loop formed by two base pairs $i \cdot j$ and $(i + 1) \cdot (j - 1)$, thus having *both* ends adjacent on the backbone. (This is the only type of loop that stabilizes the secondary structure. All other loops are destabilizing, to varying degrees.)

- A *multibranched loop* is a loop that contains more than two base pairs.
- An *external base* is a base not contained in any loop.

Definition 16.4: Given a loop, one base pair in the loop is closest to the ends of the RNA strand. This is known as the *exterior* or *closing* pair. All other pairs are *interior*. More precisely, the exterior pair is the one that maximizes $j - i$ over all pairs $i \cdot j$ in the loop.

Note that one base pair may be the exterior pair of one loop and the interior pair of another.

16.4. Free Energy Functions

The assumption of no pseudoknots leads to the following related assumptions:

1. The free energy of a secondary structure is the sum of the free energies of its loops.
2. The free energy of a loop is independent of all other loops.

These assumptions imply that, to evaluate the free energy of a given secondary structure, all that is needed is a set of functions that provide the free energies of the allowable constituent loop types. These functions are the *free energy functions*, which we will assume are provided by experimentalists and are available for the algorithm's use. See `http://www.ibc.wustl.edu/~zuker/rna/energy/node2.html#SECTION20` for typical tables and formulas that can be used.

Definition 16.5: There are four free energy functions:

- $\mathrm{eS}(i, j)$. This function gives the free energy of a stacked pair that consists of $i \cdot j$ and $(i+1) \cdot (j-1)$. $\mathrm{eS}(i, j)$ depends on all the bases involved in the stack, namely s_i, s_j, s_{i+1}, and s_{j-1}. Because stacked complementary base pairs are stabilizing, eS values will be negative if both stacked base pairs are complementary. In addition to the usual complementary pairs A-U and C-G, the pair G-U forms a weak bond in RNA, and is sometimes called a "wobble pair". The eS values involving such pairs will also be negative.
- $\mathrm{eH}(i, j)$. This function gives the free energy of a hairpin loop closed by $i \cdot j$. This function depends on several factors, including the length of the loop, s_i and s_j, and the unpaired bases adjacent to s_i and s_j on the loop.
- $\mathrm{eL}(i, j, i', j')$. This function gives the free energy of an internal loop or bulge with exterior pair $i \cdot j$ and interior pair $i' \cdot j'$. Similar to eH, this function depends on $i' - i$, $j - j'$, the four paired bases, and the unpaired bases adjacent to the paired bases on the loop.
- $\mathrm{eM}(i, j, i_1, j_1, \ldots, i_k, j_k)$. This function gives the free energy of a multibranched loop closed by $i \cdot j$ with interior pairs $i_1 \cdot j_1, \ldots, i_k \cdot j_k$. This function is the least well understood at this time.

16.5. Dynamic Programming Arrays

The algorithm described by Lyngsø *et al.* [33] uses dynamic programming, the technique that was used to find optimal alignments (Section 4.1). Like the affine gap penalty algorithm of Section 5.3.3, this one fills in several tables simultaneously. The five tables used are described below.

$W(j)$: the free energy of the optimal structure of the first j residues, $s_1 s_2 \dots s_j$. This is the key array: if we can compute $W(n)$ (and find its associated secondary structure), we are done.

$V(i,j)$: the free energy of the optimal structure for $s_i \dots s_j$, assuming $i \cdot j$ forms a base pair in that structure.

$VBI(i,j)$: the free energy of the optimal structure for $s_i \dots s_j$, assuming $i \cdot j$ closes a bulge or internal loop.

$VM(i,j)$: the free energy of the optimal structure for $s_i \dots s_j$, assuming $i \cdot j$ closes a multibranched loop.

$WM(i,j)$: used to compute VM, in a manner to be revealed later.

Despite the similarity in their number and descriptions, it is important to understand the distinction between the free energy functions of Section 16.4 and these dynamic programming arrays. The free energy functions give the energy of a single specified loop. The arrays will generally contain free energy values for a collection of consecutive loops. For example, referring to Figure 16.1, $\mathrm{eL}(a,b,c,d)$ gives the free energy of the internal loop closed by $a \cdot b$ and $c \cdot d$, whereas $V(a,b)$ gives the total free energy of all the loops to the right of $a \cdot b$, including the stacked pairs and the hairpin.

Lecture 17

RNA Secondary Structure Prediction (continued)

March 2, 2000
Notes: Don Patterson

17.1. Recurrence Relations

The core of the dynamic programming algorithm for RNA secondary structure prediction lies in the recurrence relations used to fill the arrays introduced in Section 16.5. This section develops the recurrence relations for W, V, VBI, and VM, which are interdependent.

17.1.1. $W(j)$

$$\begin{aligned} W(0) &= 0 \\ W(j) &= \min(W(j-1), \min_{1 \le i < j}(V(i,j) + W(i-1))), \text{ for } j > 0 \end{aligned}$$

The terms in the second equation correspond to choosing the structure for bases $s_1, s_2, \ldots, s_j$ having the lesser free energy of two possible structures:

- The base s_j does not pair with any other base and is therefore an external base (see Figure 16.1). The recurrence for $W(j)$ makes the implicit assumption that the external bases do not contribute to the overall free energy of the structure. In this case the total energy is therefore $W(j-1)$.
- The base s_j pairs with some other base s_i in $s_1, s_2, \ldots, s_{j-1}$, where i is chosen to minimize the resulting free energy. That energy is the sum of the energy $V(i,j)$ of the compound structure closed by $i \cdot j$, plus the energy $W(i-1)$ of the remainder $s_1, s_2, \ldots, s_{i-1}$.

17.1.2. $V(i,j)$

$$V(i,j) = \begin{cases} +\infty, & \text{for } i \ge j \\ \min(\mathrm{eH}(i,j), \mathrm{eS}(i,j) + V(i+1,j-1), VBI(i,j), VM(i,j)), & \text{for } i < j \end{cases}$$

The terms in the second equation correspond to choosing the minimum free energy structure among the following possible solutions:

- $i \cdot j$ is the exterior pair in a hairpin loop, whose free energy is therefore given by $\mathrm{eH}(i,j)$.

- $i \cdot j$ is the exterior pair of stacked pair. In this case the free energy is the energy $\mathrm{eS}(i,j)$ of the stacked pair, plus the energy $V(i+1, j-1)$ of the compound structure closed by $(i+1) \cdot (j-1)$. We know in this case that $(i+1) \cdot (j-1)$ forms a base pair because $i \cdot j$ is the exterior pair of a stacked pair.
- $i \cdot j$ is the exterior pair of a bulge or internal loop, whose free energy is therefore given by $VBI(i,j)$.
- $i \cdot j$ is the exterior pair of a multibranched loop, whose free energy is therefore given by $VM(i,j)$.

17.1.3. $VBI(i,j)$

$$VBI(i,j) = \min_{\substack{i',j' \\ i<i'<j'<j}} (\mathrm{eL}(i,j,i',j') + V(i',j'))$$

In this case, $i \cdot j$ is the exterior pair of a bulge or interior loop, and we must search all possible interior pairs $i' \cdot j'$ for the pair that results in the minimum free energy. For each such interior pair, the resulting free energy is sum of the energy $\mathrm{eL}(i,j,i',j')$ of the bulge or internal loop, plus the energy $V(i',j')$ of the compound structure closed by $i' \cdot j'$. It is easy to see that this search for the best interior pair is computationally intensive, simply because of the number of possibilities that must be considered. We will see later how to speed up this calculation, which is the new contribution of Lyngsø *et al.* [33].

17.1.4. $VM(i,j)$

$$VM(i,j) = \min_{\substack{k,i_1,j_1,i_2,j_2,\ldots,i_k,j_k \\ i<i_1<j_1<i_2<j_2<\ldots<i_k<j_k<j \\ k\geq 2}} (\mathrm{eM}(i,j,i_1,j_1,i_2,j_2,\ldots,i_k,j_k) + \sum_{h=1}^{k} V(i_h,j_h))$$

In the same way that the recurrence for VBI requires a search for the best structure among all the possible interior pairs, the calculation for VM is even more intensive, requiring a search for k interior pairs $i_h \cdot j_h$, each of which closes its own branch out of the multibranched loop and contributes free energy $V(i_h,j_h)$. A direct implementation of the calculation shown for VM is infeasibly slow. Section 17.3 will discuss simplifying assumptions about multibranched loops that allow us to speed this up substantially.

17.2. Order of Computation

The interdependence of these recurrences requires a careful ordering of the calculations to ensure that we only rely on array entries whose values have already been determined. Specifically, the entries are computed in order from interior pairs to exterior pairs. This corresponds to filling the arrays V, VBI, and VM in order of increasing values of $j-i$. An inspection of the recurrences in Sections 17.1.2 – 17.1.4 reveals that this order will always guarantee that the needed array entries have been computed.

Within the calculations involving a given value $j-i$, we compute $VBI(i,j)$ and $VM(i,j)$ before $V(i,j)$, in order to accommodate the recurrence in Section 17.1.2. Note that the calculations for the three tables are interleaved: we calculate the entry in each table for a given pair i,j before advancing to the next pair.

Because none of these entries depend on the values of entries in W, the computation of W can be deferred until the other three tables have been completed.

17.3. Speeding Up the Multibranched Computation

As mentioned in Section 16.4, the actual free energy values of multibranched loops are not yet well understood. Given this state, the approximation we will describe is driven more by a desire to reduce the running time of the dynamic program than to produce a very accurate physical model of the loop.

For this approximation, we assume that the free energy of a multibranched loop is given by an affine linear function of the number k of branches and the size of the loop (measured as the number of unpaired bases):

$$\mathrm{eM}(i, j, i_1, j_1, \ldots, i_k, j_k) = a + bk + c((i_1 - i - 1) + (j - j_k - 1) + \sum_{h=1}^{k-1}(i_{h+1} - j_h - 1)),$$

where a, b, and c are constants. (Lyngsø *et al.* [33] suggest that it would be more accurate to approximate the free energy as a logarithmic function of the loop size.)

Assuming this linear approximation, we can devise a much more efficient dynamic programming solution for computing VM than the one given in Section 17.1.4. This solution requires an additional array WM, where $WM(i, j)$ gives the free energy of an optimal structure for $s_i, \ldots, s_j$, assuming that s_i and s_j are on a multibranched loop. WM is defined by the following recurrence relation:

$$\begin{aligned} WM(i,i) &= c \\ WM(i,j) &= \min(V(i,j) + b, \min_{i<h\leq j}(WM(i,h-1) + WM(h,j))), \text{ for } i < j \end{aligned}$$

The terms in the second equation correspond to the following possible solutions:

- $i \cdot j$ forms a base pair and therefore defines one of the k branches, whose free energy is $V(i, j)$.
- s_i and s_j are not paired with each other, so the free energy is given by the minimum partition of the sequence into two contiguous subsequences.

Calculating VM then reduces to partitioning the loop into at least two pieces with the minimum total free energy:

$$VM(i,j) = \min_{i+1<h\leq j-1}(WM(i+1,h-1) + WM(h,j-1) + a)$$

17.4. Running Time

The running time to fill in each of the complete tables (assuming the values on which it depends have already been computed and stored in their tables, and that we are using the multibranched approximation of Section 17.3) is determined as follows:

- W: $O(n^2)$. Each of n entries requires the computation of the min of $O(n)$ terms.
- V: $O(n^2)$. Each of $O(n^2)$ entries requires the computation of the min of 4 terms.

- VBI: $O(n^4)$. Each of $O(n^2)$ entries requires the computation of the min of $O(n^2)$ terms.
- WM: $O(n^3)$. Each of $O(n^2)$ entries requires the computation of the min of $O(n)$ terms.
- VM: $O(n^3)$. Each of $O(n^2)$ entries requires the computation of the min of $O(n)$ terms.

With the speedup of the multibranched loop computation described in Section 17.3, the new bottleneck has become the $O(n^4)$ time computation of the free energy of bulges and internal loops. We will see next how to eliminate this bottleneck.

Lecture 18

Speeding Up Internal Loop Computations

March 7, 2000
Notes: Kellie Plow

Recall from Section 17.4 that the running time for determining the internal loop free energy calculation is $O(n^4)$: each of the $O(n^2)$ exterior pairs $i \cdot j$ requires a search through the $O(n^2)$ interior pairs $i' \cdot j'$ for one that minimizes the resulting free energy. We will address this $O(n^4)$ running time computation of the free energy of bulges and internal loops, and show that it can be decreased to $O(n^3)$. This is the main result of Lyngsø *et al.* [33] and, combined with the remaining analysis in Section 17.4, shows that the entire RNA secondary structure prediction problem can be solved in time $O(n^3)$. $O(n^3)$ running time is not practical for long RNA sequences, but it does allow for secondary structure prediction for RNA sequences that are hundreds of bases in length, which would be prohibitive with an n^4 time algorithm.

18.1. Assumptions About Internal Loop Free Energy

To speed up the running time it is necessary to make some assumptions about the form of the internal loop free energy function $\mathrm{eL}(i, j, i', j')$ (see Definition 16.5). The authors cite thermodynamics studies that support the fact that these assumptions are realistic.

The authors first assume that eL is the sum of 3 contributions:

1. a term "$\mathrm{size}(i' - i + j - j' - 2)$" that is a function of the size of the loop, plus
2. stacking energies "$\mathrm{stacking}(i, j) + \mathrm{stacking}(i', j')$" for the unpaired bases adjacent on the loop to the two base pairs, plus
3. an asymmetry penalty "$\mathrm{asymmetry}(i' - i - 1, j - j' - 1)$", where $i' - i - 1$ is the number of unpaired bases between the two base pairs on one side of the loop, and $j - j' - 1$ the number on the other side.

The free energy function for bulges and internal loops is thus given by

$$\mathrm{eL}(i, j, i', j') = \mathrm{size}(i'-i+j-j'-2)+\mathrm{stacking}(i, j)+\mathrm{stacking}(i', j')+\mathrm{asymmetry}(i'-i-1, j-j'-1). \tag{18.1}$$

18.2. Asymmetry Penalty

The currently used asymmetry functions are of the form

$$\mathrm{asymmetry}(n_1, n_2) = \min(E_{\max}, |n_1 - n_2| f(m)), \tag{18.2}$$

where $E_{\max}$ is the maximum asymmetry penalty assessed, f is a function whose details need not concern us, $m = \min(n_1, n_2, c)$, and c is a small constant (equal to 5 and 1, respectively, in two cited thermodynamics studies).

What is important for our purposes is that this asymmetry penalty grows linearly in $|n_1 - n_2|$, provided that $n_1 \geq c$ and $n_2 \geq c$. In particular, the only assumption we will need to make about the penalty is that

$$\mathrm{asymmetry}(n_1, n_2) = \mathrm{asymmetry}(n_1 + 1, n_2 + 1) \tag{18.3}$$

for all $n_1 \geq c$ and $n_2 \geq c$. This is certainly true for the particular form given in Equation (18.2).

18.3. Comparing Interior Pairs

Recall from Section 17.1.3 the recurrence

$$VBI(i, j) = \min_{\substack{i', j' \\ i < i' < j' < j}} \left(V(i', j') + \mathrm{eL}(i, j, i', j') \right).$$

We are going to save time by not searching through all the interior pairs $i' \cdot j'$. Suppose that, for exterior pair $i \cdot j$, the interior pair $i' \cdot j'$ is better than $i'' \cdot j''$, and that both of these loops have the same size. Then, under the assumptions from Sections 18.1 and 18.2, Theorem 18.1 below demonstrates that $i' \cdot j'$ is also better for exterior pair $(i-1) \cdot (j+1)$. The intuition behind this theorem is that the asymmetry penalty for $i' \cdot j'$ is the same for the two different exterior pairs by Equation (18.3), as is the asymmetry penalty for $i'' \cdot j''$, and neither interior pair gains an advantage in loop size or stacking energies when you change from the smaller to the bigger loop.

Theorem 18.1: Let

$$j' - i' = j'' - i'' \tag{18.4}$$

(so as to compare internal loops of identical size). Let $i' - i - 1$, $j - j' - 1$, $i'' - i - 1$, and $j - j'' - 1$ each be at least c (so that Equation (18.3) applies to both loops). Suppose that

$$V(i', j') + \mathrm{eL}(i, j, i', j') \leq V(i'', j'') + \mathrm{eL}(i, j, i'', j''). \tag{18.5}$$

Then

$$V(i', j') + \mathrm{eL}(i-1, j+1, i', j') \leq V(i'', j'') + \mathrm{eL}(i-1, j+1, i'', j'').$$

Proof:

$$
\begin{aligned}
& V(i',j') + \mathrm{eL}(i-1,j+1,i',j') \\
= \quad & V(i',j') + \mathrm{size}(i'-i+j-j') + \mathrm{stacking}(i-1,j+1) + \mathrm{stacking}(i',j') \\
& + \mathrm{asymmetry}(i'-i,j-j') && \text{Equation (18.1)} \\
= \quad & V(i',j') + \mathrm{eL}(i,j,i',j') \\
& - \mathrm{size}(i'-i+j-j'-2) + \mathrm{size}(i'-i+j-j') \\
& - \mathrm{stacking}(i,j) + \mathrm{stacking}(i-1,j+1) \\
& - \mathrm{asymmetry}(i'-i-1,j-j'-1) + \mathrm{asymmetry}(i'-i,j-j') && \text{Equation (18.1)} \\
= \quad & V(i',j') + \mathrm{eL}(i,j,i',j') \\
& - \mathrm{size}(i'-i+j-j'-2) + \mathrm{size}(i'-i+j-j') \\
& - \mathrm{stacking}(i,j) + \mathrm{stacking}(i-1,j+1) && \text{Equation (18.3)} \\
\leq \quad & V(i'',j'') + \mathrm{eL}(i,j,i'',j'') \\
& - \mathrm{size}(i''-i+j-j''-2) + \mathrm{size}(i''-i+j-j'') \\
& - \mathrm{stacking}(i,j) + \mathrm{stacking}(i-1,j+1) && \text{Equations (18.4) \& (18.5)} \\
= \quad & V(i'',j'') + \mathrm{eL}(i,j,i'',j'') \\
& - \mathrm{size}(i''-i+j-j''-2) + \mathrm{size}(i''-i+j-j'') \\
& - \mathrm{stacking}(i,j) + \mathrm{stacking}(i-1,j+1) \\
& - \mathrm{asymmetry}(i''-i-1,j-j''-1) + \mathrm{asymmetry}(i''-i,j-j'') && \text{Equation (18.3)} \\
= \quad & V(i'',j'') + \mathrm{size}(i''-i+j-j'') + \mathrm{stacking}(i-1,j+1) + \mathrm{stacking}(i'',j'') \\
& + \mathrm{asymmetry}(i''-i,j-j'') && \text{Equation (18.1)} \\
= \quad & V(i'',j'') + \mathrm{eL}(i-1,j+1,i'',j'') && \text{Equation (18.1)}
\end{aligned}
$$

□

Instead of using a two-dimensional array $VBI(i,j)$, use a three-dimensional array $VBI(i,j,l)$, where l is the loop size. This array will be filled in using dynamic programming. The entry $VBI(i,j,l)$ will store not only the free energy, but also the best interior pair $i' \cdot j'$ (subject to $i'-i-1 \geq c$ and $j-j'-1 \geq c$) that gives this energy.

Now suppose that the entry $VBI(i,j,l)$ has been calculated, and we want to calculate the entry $VBI(i-1,j+1,l+2)$. By Theorem 18.1, the interior pair $i' \cdot j'$ stored in $VBI(i,j,l)$ is the best interior pair for $VBI(i-1,j+1,l+2)$, with only two possible exceptions. These possible exceptions are the loops with exterior pair $(i-1)\cdot(j+1)$, length $l+2$, and having one or the other loop side of length exactly c.

Thus, for each of $O(n^3)$ entries in VBI, it is necessary to compare 3 loop energies and store the minimum, which takes constant time. It is also necessary to compare those loops with exterior pair $(i-1)\cdot(j+1)$

and length $l + 2$ having one or the other loop side of length less than c, but there are only a constant number of these.

Bibliography

References

[1] Tatsuya Akutsu. Hardness results on gapless local multiple sequence alignment. Technical Report 98-MPS-24-2, Information Processing Society of Japan, 1998.

[2] Tatsuya Akutsu, Hiroki Arimura, and Shinichi Shimozono. On approximation algorithms for local multiple alignment. In *RECOMB00: Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, Tokyo, Japan, April 2000.

[3] S. F. Altschul. A protein alignment scoring system sensitive at all evolutionary distances. *Journal of Molecular Evolution*, 36(3):290–300, March 1993.

[4] Timothy L. Bailey and Charles Elkan. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning*, 21(1-2):51–80, October 1995.

[5] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[6] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[7] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[8] M. Borodovsky and J. McIninch. GeneMark: Parallel gene recognition for both DNA strands. *Comp. Chem.*, 17(2):123–132, 1993.

[9] M. Borodovsky, J. McIninch, E. Koonin, K. Rudd, C. Medigue, and A. Danchin. Detection of new genes in a bacterial genome using Markov models for three gene classes. *Nucleic Acids Research*, 23(17):3554–3562, 1995.

[10] C. Branden and J. Tooze. *An Introduction to Protein Structure*. Garland, 1998.

[11] Stephen A. Cook. The complexity of theorem proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[13] Arthur L. Delcher, Douglas Harmon, Simon Kasif, Owen White, and Steven L. Salzberg. Improved microbial gene identification with GLIMMER. *Nucleic Acids Research*, 27(23):4636–4641, 1999.

[14] Karl Drlica. *Understanding DNA and Gene Cloning*. John Wiley & Sons, second edition, 1992.

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

[16] Dmitrij Frishman, Andrey Mironov, Hans-Werner Mewes, and Mikhail Gelfand. Combining diverse evidence for gene recognition in completely sequenced bacterial genomes. *Nucleic Acids Research*, 26(12):2941–2947, 1998.

[17] Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[19] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993.

[20] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[21] Sridhar S. Hannenhalli, William S. Hayes, Artemis G. Hatzigeorgiou, and James W. Fickett. Bacterial start site prediction. 1999.

[22] William S. Hayes and Mark Borodovsky. Deriving ribosomal binding site (RBS) statistical models from unannotated DNA sequences and the use of the RBS model for N-terminal prediction. In *Pacific Symposium on Biocomputing*, pages 279–290, 1998.

[23] Gerald Z. Hertz and Gary D. Stormo. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7/8):563–577, July/August 1999.

[24] D. S. Hirschberg. A linear-space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, June 1975.

[25] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, Nov.–Dec. 1989.

[26] Jerzy Jurka and Mark A. Batzer. Human repetitive elements. In Robert A. Meyers, editor, *Encyclopedia of Molecular Biology and Molecular Medicine*, volume 3, pages 240–246. Weinheim, Germany, 1996.

[27] Samuel Karlin and Stephen F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Science USA*, 87(6):2264–2268, March 1990.

[28] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[29] Charles E. Lawrence, Stephen F. Altschul, Mark S. Boguski, Jun S. Liu, Andrew F. Neuwald, and John C. Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 8 October 1993.

[30] Ming-Ying Leung, Genevieve M. Marsh, and Terence P. Speed. Over- and underrepresentation of short DNA words in herpesvirus genomes. *Journal of Computational Biology*, 3(3):345–360, 1996.

[31] Benjamin Lewin. *Genes VI*. Oxford University Press, 1997.

[32] Rune B. Lyngsø and Christian N. S. Pedersen. Pseudoknots in RNA secondary structures. In *RECOMB00: Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, Tokyo, Japan, April 2000.

[33] Rune B. Lyngsø, Michael Zuker, and Christian N. S. Pedersen. Internal loops in RNA secondary structure prediction. In *RECOMB99: Proceedings of the Third Annual International Conference on Computational Molecular Biology*, pages 260–267, Lyon, France, April 1999.

[34] Merja Mikkonen, Jussi Vuoristo, and Tapani Alatossava. Ribosome binding site consensus sequence of *Lactobacillus delbrueckii* subsp. *lactis*. *FEMS Microbiology Letters*, 116:315–320, 1994.

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

[36] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.

[37] 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*, 48:443–453, 1970.

[38] Elena Rivas and Sean R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5):2053–2068, February 1999.

[39] Fred S. Roberts. *Applied Combinatorics*. Prentice-Hall, 1984.

[40] Walter L. Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241, Heidelberg, Germany, August 1999. AAAI Press.

[41] Steven L. Salzberg, Arthur L. Delcher, Simon Kasif, and Owen White. Microbial gene identification using interpolated Markov models. *Nucleic Acids Research*, 26(2):544–548, 1998.

[42] Steven L. Salzberg, David B. Searls, and Simon Kasif, editors. *Computational Methods in Molecular Biology*. Elsevier, 1998.

[43] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[44] J. Shine and L. Dalgarno. The $3'$-terminal sequence of *E. coli* 16S ribosomal RNA: Complementarity to nonsense triplets and ribosome binding sites. *Proceedings of the National Academy of Science USA*, 71:1342–1346, 1974.

[45] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[46] Gary D. Stormo and Dana S. Fields. Specificity, free energy and information content in protein-DNA interactions. *Trends in Biochemical Sciences*, 23:109–113, 1998.

[47] Gary D. Stormo and George W. Hartzell III. Identifying protein-binding sites from unaligned DNA fragments. *Proceedings of the National Academy of Science USA*, 86:1183–1187, 1989.

[48] Martin Tompa. An exact method for finding short motifs in sequences, with application to the ribosome binding site problem. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 262–271, Heidelberg, Germany, August 1999. AAAI Press.

[49] Robert Luis Vellanoweth and Jesse C. Rabinowitz. The influence of ribosome-binding-site elements on translational efficiency in *Bacillus subtilis* and *Escherichia coli in vivo*. *Molecular Microbiology*, 6(9):1105–1114, 1992.

[50] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.

[51] Michael S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, 1995.

[52] James D. Watson, Michael Gilman, Jan Witkowski, and Mark Zoller. *Recombinant DNA*. Scientific American Books (Distributed by W. H. Freeman), second edition, 1992.