Algorithms for Computational Molecular Biology

David A. Hendrix

May 11, 2018

Contents

Chapter 1

Introduction to Sequences and Probability

1.1 DNA, RNA, and Proteins

The concurrent advances in computer technology and molecular biology in the late 20th century led to the field of bioinformatics and computational biology. Fundamentally, the subject matter of these disciplines is biological sequences. More broadly, these fields seek to represent, model, analyze, and store biological information in many different forms. Although biological data can consist of a large range of types from protein structure data to images of gene expression through in situ hybridization, these notes focus on the study of biological sequences like RNA, DNA and proteins.

One important finding about the nature of biological information is the central dogma of molecular biology [1, 2]. The central dogma states that biological information flows from DNA to RNA to proteins. Further studies have shown that information can also flow from RNA to DNA, in the form of reverse transcription, and from RNA to RNA, in the form of RNA Dependent RNA Polymerase. Information is encoded in biological sequences, and numerous algorithmic approaches have been developed to study this information. Let's first lay the groundwork about sequences and strings and some of the relevant probability theory used in computational molecular biology.

1.2 Sequences and Strings

A biological sequence can be represented as an ordered array of characters from a specific alphabet. Consider the DNA sequence $x = ACGATATCAC$. The ith letter in the sequence x can be represented as $x[i]$. The substring from i to j can be represented as $x[i..j] = x[i]x[i+1]...x[j-1]x[j]$. We can represent the length of the sequence x by $|x|$. One important point is that there are different systems of indexing sequences. Computer scientists typically start with 0 (zero-based), but sometimes biological sequences are also indexed starting at 1 (1-based). In these class notes, we will start with 1 unless otherwise indicated. Putting these notations together, we have $x = x[1..|x|]$.

We can consider the terms of these sequences to be such that the elements are members of an alphabet $\mathcal{A}$, in other words $x[i] \in \mathcal{A}$. For DNA, we have the alphabet of nucleotides $\mathcal{D} = \{A, C, G, T\}$, and for RNA, we have the alphabet $\mathcal{R} = \{A, C, G, U\}$. Proteins have a 20 character alphabet $\mathcal{P} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$. In each of these cases, depending on the context and application, other non-canonical characters corresponding to non-canonical nucleotides and amino acids, such as 5-methylcytosine, which is a frequently occurring modified DNA nucleotide.

One useful construct is the indicator function $\mathbb{1}_A(x)$, which can be defined on a value x and a set A such that:

$$\mathbb{1}_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases} \tag{1.1}$$

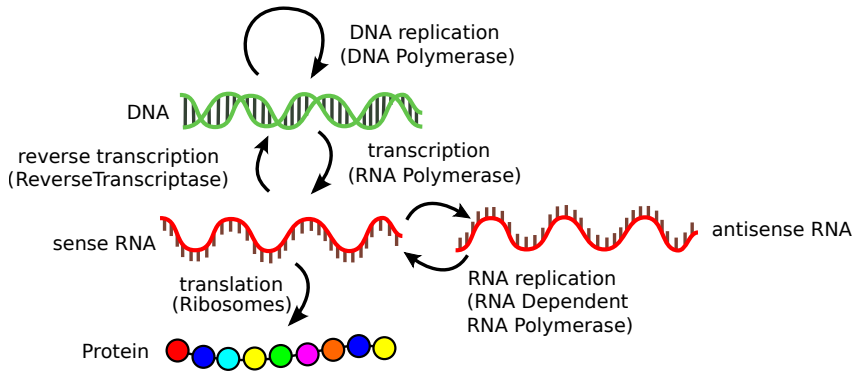


Figure 1.1: Central Dogma of Molecular Biology

Symbol	Meaning	Mnemonic
R	A, G	puRine
Y	C, T	pYrimidine
W	A, T	Weak (weaker basepairs, fewer hydrogen bonds)
S	G, C	Strong (stronger basepairs, more hydrogen bonds)
K	G or T	Keto (both G and T have a keto group)
M	A or C	aMine (both A and C have a amine group)
B	C, G, T	not A (B comes after A)
D	A, G, T	not C (D comes after C)
H	A, C, T	not G (H comes after G)
V	A, C, G	not T or U (V comes after T and U)
N	A, C, G, T	aNy base

Table 1.1: IUPAC codes for nucleotides. In this table, everywhere that T applies, U applies as well.

For example, we can compute the GC content of a sequence x by define the set $S = \{G, C\}$, the set containing G and C:

$$GC(x) = \sum_{i=1}^{|x|} \mathbb{1}_S(x) \tag{1.2}$$

The choice of the character S to refer to the set containing G or C was not arbitrary. The International Union of Pure and Applied Chemistry sets codes for different nucleotide combinations, and G or C is denoted by S, for "strong" since these nucleotides basepair more strongly due to more hydrogen bonds. These codes can be thought of a as a "set of characters", in that a given nucleotide can be an element of the set, much like $G, C \in S$. Table 1.1 demonstrates these codes and a mnemonic for remembering them.

RNA and DNA can form base pairs, such that A pairs with U or T, and G pairs with C. These bases that pair are said to be "complementary". For a given sequence x, we are often interested in the sequence that it pairs with: its reverse complement. Let x' denote the reverse of the sequence x. Then we have the relationship

$$x'[i] = x[|x| - i + 1]$$

We can define x^* as the complementary sequence, so we have $A^* = T$ (for DNA) and $G^* = C$. We have the properties that $(x^*)^* = x$ and $(x')' = x$. Where w is the concatenation of multiple sequences such that $w = xyz$, then we have the relationship $w'^* = x'^* y'^* z'^*$

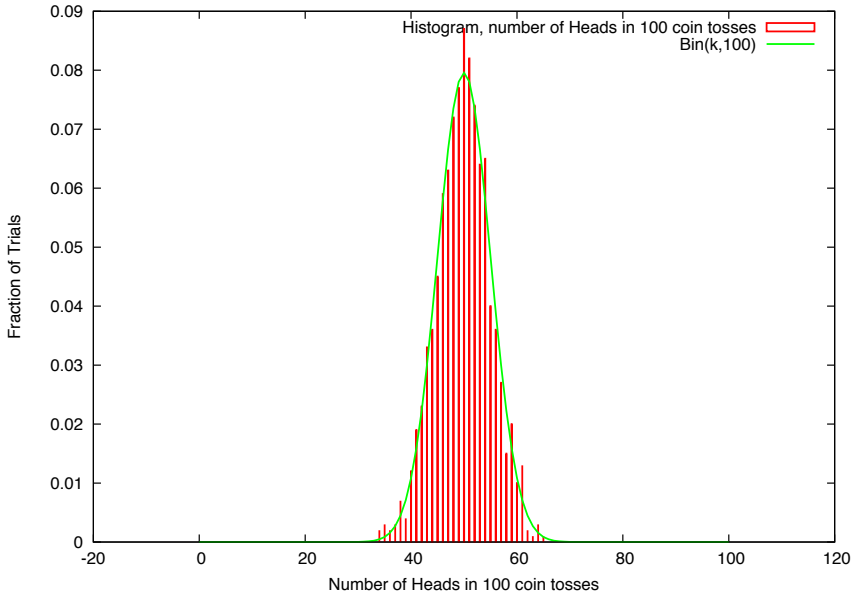


Figure 1.2: A histogram of the number of heads observed in 100 tosses of a fair coin, repeated 1000 times.

1.3 Probability

1.3.1 Bernouli Distribution

One of the simplest probability distributions we should consider is the Bernouli Distribution. It is essentially a single event, with a probability of success denoted p, and a probability of failure of $q = 1 - p$.

Example 1: A coin toss. Heads or tails? Typically for a fair coin, this will be such that $p = q = 0.5$.

Example 2: Selecting a single nucleotide from the genome at random. Is it purine (A or G) or is it pyrimidine (C or T)? For the human genome, the GC content is about 0.417, but it varies from chromosome to chromosome.

1.3.2 Binomial Distribution

Central to the Binomial distribution is the binomial coefficient $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ The number of ways to have k successes out of n trials is $\binom{n}{k}$. The probability of observing k successes out of n trials is $P(k|p,n) = \binom{n}{k}p^k q^{n-k}$. We can see that these probabilities are normalized (sum to one) by the equation:

$$1 = (p+q)^n = \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k}$$

Example 1: Consider tossing a fair coin 100 times, and repeating this for 1000 trials. A histogram for k, number of heads in n tosses is shown in Figure 1.2. The expected value of k is $E[k] = np$, and the variance is $var(k) = npq$.

1.3.3 Categorical Distribution

The generalization of the binomial distribution to more than two possible outcomes is called the "categorical distirubution". This applies nicely to biological sequences where we would like to describe the probability of

a particular character. For example, a DNA sequence would be described by the probabilities p_A, p_C, p_G, p_T. We have the condition that the sum over all possible outcomes would equal 1, so in this example:

$$\sum_{b=A}^{T} p_b = 1 \tag{1.3}$$

1.3.4 Multinomial Distribution

The multinomial distribution deals which situations when there are more than two possible outcomes (for example, DNA nucleotides). The multinomial coefficient, $M(\vec{n})$, in this example describes the number of ways to have a sequence of length n, with the number of occurrences of A,C,G,and T to be n_A, n_C, n_G, and n_T respectively.

$$M(\vec{n}) = \frac{n!}{n_A! n_C! n_G! n_T!}$$

The probability of a particular set of observed counts $\vec{n} = (n_A, n_C, n_G, n_T)$ depends on the frequencies $\vec{p} = (p_A, p_C, p_G, p_T)$ by the expression:

$$P(\vec{n}|\vec{p}) = \frac{n!}{n_A! n_C! n_G! n_T!} \prod_{i=A}^{T} p_i^{n_i}$$

1.3.5 Conditional Probability

In probability, we often want to talk about conditional probability, which gives us the probability of an event given that we know that another event has occurred.

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Bayes' Theorem tells us that:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

We can also expand a probability of an event into the conditional probabilities of each other condition, times the probability of these other events.

$$P(x) = \sum_{i=1}^{N} P(x|A_i)P(A_i)$$

This expression $P(x)$ is often called the evidence. We can express the probability of A given x, which is called the posterior probability by

$$P(A|x) = \frac{P(x|A)P(A)}{P(x)}$$

Also in this equation, $P(x|A)$ is called the likelihood, and $P(A)$ is called the prior.

Chapter 2

Sequence Comparison Algorithms

Fundamental to computational molecular biology is the comparison of sequences. For example, one might want to know the number of mutations between two sequences, or to find a common subsequence between them. Here we describe some basic algorithms for comparing two biological sequences without actually performing an alignment.

The following definitions are useful for describing comparisons of sequences.

- **Similarity:** the degree of resemblance between two sequences.
- **Identity:** the state of possessing the same subsequence. One often quantifies the percent identity between two sequences.
- **Homology:** the state of sharing a common evolutionary origin.
- **Orthologous:** homologous sequences that arose from a common ancestral gene during speciation.
- **Paralogous:** homologous sequences that arose from a common ancestral gene from gene duplication.

2.1 Dynamic Programming Algorithms for Sequence Comparisons

Dynamic Programming: The optimal solution to the problem is computed in terms of the optimal solution to a sub-problem. We build the full solution from the solutions to portions of the larger task at hand.

2.2 Edit Distance

The concept of an edit distance between two sequences x and y was introduced by Levenshtein as the minimum number of insertions, deletions, and substitutions needed to transform x into y.

Denote such an edit distance as $d(x, y)$. This is a true distance metric in a topological sense, meaning:

- identity: $d(x, y) = 0$ if and only if $x = y$
- symmetry: $d(x, y) = d(y, x)$ for all x and y.
- triangle inequality: $d(x, y) \leq d(x, z) + d(z, y)$

Going back to the previous example, the edit distance between $x =$ `CGCACGAAGACAGG` and $y =$ `CTGACTAGTCAGAG` corresponds to the total length of the gaps and substituted nucleotides. We can see in Table 3.1, the total number of gaps and mismatches is 6, so in this case, the edit distance would be $d(x, y) = 6$.

An algorithm for computing the edit distance has been invented by multiple people, but is often credited to **Wagner and Fischer**. It begins by defining a $(|x| + 1) \times (|y| + 1)$ matrix D such that the terms of the matrix $D_{i,j}$ are the distance between the prefix subsequences $x[1..i]$ and $y[1..j]$. In other words, $D_{i,j} = d(x[1..i], y[1..j])$. To fill up the terms of D involves the recurrence relation:

	-	C	T	G	A	C	T	A	G	T	C	A	G	A	G
-	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
G	2	1	1	1	2	3	4	5	6	7	8	9	10	11	12
C	3	2	2	2	2	2	3	4	5	6	7	8	9	10	11
A	4	3	3	3	2	3	3	3	4	5	6	7	8	9	10
C	5	4	4	4	3	2	3	4	4	5	5	6	7	8	9
G	6	5	5	4	4	3	3	4	4	5	6	6	6	7	8
A	7	6	6	5	4	4	4	3	4	5	6	6	7	6	7
A	8	7	7	6	5	5	5	4	4	5	6	6	7	7	7
G	9	8	8	7	6	6	6	5	4	5	6	7	6	7	7
A	10	9	9	8	7	7	7	6	5	5	6	6	7	6	7
C	11	10	10	9	8	7	8	7	6	6	5	6	7	7	7
A	12	11	11	10	9	8	8	8	7	7	6	5	6	7	8
G	13	12	12	11	10	9	9	9	8	8	7	6	5	6	7
G	14	13	13	12	11	10	10	10	9	9	8	7	6	6	6

Table 2.1: The Distance matrix $D_{i,j}$ computed from the Wagner-Fischer Algorithm for two sequences.

$$D_{i,j} = min \begin{cases} D_{i-1,j} + 1 & \text{skip a position of } x \\ D_{i,j-1} + 1 & \text{skip a position of } y \\ D_{i-1,j-1} & \text{if } x[i] = y[j] \\ D_{i-1,j-1} + 1 & \text{if } x[i] \neq y[j] \end{cases} \tag{2.1}$$

In addition to the recurrence relation, we need to define boundary conditions for the matrix. Namely, the terms $D_{i,0} = i$ and $D_{0,j} = j$, because these terms correspond to introducing a gap of length i and j respectively, at the beginning of the alignment.

The Wagner-Fischer Algorithm for computing the Levenshtein Distance between two sequences x and y.

```
function EDIT DISTANCE(x, y)                                   ▷ Compute d(x, y)
    for i ← 0 to |x| do
        D_{i,0} ← i
    for j ← 0 to |y| do
        D_{0,j} ← j
    for i ← 1 to |x| do
        for j ← 1 to |y| do
            if x[i] = y[j] then
                D_{i,j} ← D_{i-1,j-1}                          ▷ Match
            else
                if D_{i-1,j-1} < D_{i-1,j} and D_{i-1,j-1} < D_{i,j-1} then
                    D_{i,j} ← D_{i-1,j-1} + 1                  ▷ Subs.
                else if D_{i-1,j} < D_{i,j-1} then
                    D_{i,j} ← D_{i-1,j} + 1                    ▷ Insertion
                else
                    D_{i,j} ← D_{i,j-1} + 1                    ▷ Deletion
    return D_{|x|,|y|}
```

After filling up the terms of the matrix with D this algorithm, we get the following $(|x| + 1) \times (|y| + 1)$ matrix listed in Table 2.1.

The time and data storage of the Wagner-Fischer is $\mathcal{O}(|x||y|)$.

2.3 Longest Common Subsequence Algorithm

Longest common subsequence: Given two sequences x and y, a common subsequence of length ℓ can be defined as a set of indices

$$1 \leq i_1 < ... < i_\ell \leq |x|$$

and

$$1 \leq j_1 < ... < j_\ell \leq |y|$$

such that $x[i_k] = y[j_k]$ for $1 \leq k \leq \ell$. Note that the indices i_k and j_k do not need to all be adjacent.

For example, $x =$ `CGCACGAAGACAGG` and $y =$ `CTGACTAGTCAGAG` have the Longest common subsequence of `CGACAGCAGG`

Let $\ell_{LCS}(x, y)$ be the length of the longest common subsequence (LCS) of x and y.

For the purposes of framing a dynamic programming algorithm to find the LCS, we need to define a matrix $L_{i,j}$ that we will be filling such that the terms of the matrix are the LCS for comparing $x[1..i]$ and $y[1..j]$:

$$L_{i,j} = \ell_{LCS}(x[1..i], y[1..j])$$

We can define a set of boundary conditions for the scoring matrix $L_{i,j}$, namely that the score is 0 at the boundaries so that $L_{i,0} = L_{0,j} = 0$ for $0 \leq i \leq |x|$ and $0 \leq j \leq |y|$. Define the recurrence relation:

$$L_{i,j} = max \begin{cases} L_{i-1,j} & \text{skip a position of } x \\ L_{i,j-1} & \text{skip a position of } y \\ L_{i-1,j-1} + 1 & \text{if } x[i] = y[j] \end{cases} \tag{2.2}$$

Compute the length of the longest common substring between two sequences x and y.

```
function LCS(x, y)                                   ▷ Compute ℓ_LCS(x, y)
    for i ← 0 to |x| do
        L_{i,0} ← 0
    for j ← 0 to |y| do
        L_{0,j} ← 0
    for i ← 1 to |x| do
        for j ← 1 to |y| do
            if x[i] = y[j] then
                L_{i,j} ← L_{i-1,j-1} + 1
            else if L_{i-1,j} > L_{i,j-1} then
                L_{i,j} ← L_{i-1,j}
            else
                L_{i,j} ← L_{i,j-1}
    return L_{|x|,|y|}
```

Chapter 3

Sequence Alignment

3.1 Introduction to Sequence Alignments

Sequence Alignment: The computation of an optimal arrangement of two or more sequences to minimize the differences between them.

A sequence alignment represents a comparison between two sequences by introducing gaps, represented by a dash "-", to indicate a place where an insertion or deletion is needed to make the similar characters line up. For example, an alignment of the sequences $x =$ `CGCACGAAGACAGG` and $y =$ `CTGACTAGTCAGAG` could be represented in Figure 3.1.

where the vertical bars "`|`" represent matches. Before we discuss algorithms for computing an optimal alignment between two sequences, let's consider the task of computing the number of edits required to change one sequence to another.

The difference between the alignment algorithms discussed below and LCS is that there is an alignment score and usually a gap penalty. Before we talk about alignment algorithms, let's spend some time talking about scoring methods.

3.2 Dynamic Programming Alignment Algorithms

3.2.1 Alignment Scoring Systems

Let us assume that we have an alignment for the sequences x and y. The matching amino acids will be necessarily of the same length. Let's call that length n and here we will only consider those positions for the following calculation.

$x[1]x[2]\dots x[n]$
$y[1]y[2]\dots y[n]$

Let's first compute the probability of the alignment be due to random chance. In such a situation, the probability is just the product of the probabilities p_a of each amino acid. These could be based on the frequency in a database of proteins [3].

```
C-GCACGAAGACAG-G
| | || | | ||| |
CTG-ACTA-GTCAGAG
```

Table 3.1: An alignment between two sequences. Insertions and deletions (gaps) are represented by a dash "-" and matches are represented by a pipe "`|`"

$$P(x, y|R) = \prod_{i=1}^{n} p_{x[i]} \prod_{i=1}^{n} p_{y[i]}$$

Similarly, we have the probabilities q_{ab} that the two amino acids are due to being part of homologous sequences, i.e. that pair of amino acids are evolutionarily related and the difference is due to an evolutionary change. These can be determined from a curated database of known protein alignments. Let's call this the "ancestral" model, to contrast with our random model, with probability that the two are homologous given by:

$$P(x, y|A) = \prod_{i} q_{x[i],y[i]}$$

A log likelihood score for comparing the probabilities due to the evolutionary model and due to a random chance observation:

$$log\left(\frac{P(x, y|A)}{P(x, y|R)}\right) = log\left(\frac{\prod_i q_{x[i],y[i]}}{\prod_{i=1}^{n} p_{x[i]} \prod_{i=1}^{n} p_{y[i]}}\right) = \sum_{i=1}^{n} log\left(\frac{q_{x[i],y[i]}}{p_{x[i]} p_{y[i]}}\right)$$

Therefore, we can define the terms in the final sum to be a similarity matrix, or scoring matrix $S_{a,b} = log\left(\frac{q_{a,b}}{p_a p_b}\right)$, and these terms will be used to score our alignments.

In addition to the scoring matrix, we need to also set up a gap cost $c(d)$ for penalizing gaps in the alignment of length d. Here are two gap penalty systems.

The linear gap penalty describes the cost of a gap $c_L(d)$ as just a linear multiple of the length of the gap d,

$$c_L(d) = d \times G,$$

using a parameter $G < 0$ that provides the cost per unit length of the gap.

The "affine" gap penalty has an additive constant in addition to the linear term. The cost of a gap $c_A(d)$ associated with an affine gap penalty is given by

$$c_A(d) = G + (d - 1) \times E,$$

which includes a gap open parameter G and a gap extension parameter E.

Let's compare Needleman-Wunsch and Smith-Waterman on the nucleotide sequences introduced earlier: $x =$ `CGCACGAAGACAGG` and $y =$ `CTGACTAGTCAGAG`, using a linear gap penalty $G = -1$, and the following scoring matrix:

$$S_{a,b} = \begin{cases} 1, & \text{if } a = b \\ -1, & \text{if } a \neq b \end{cases} \tag{3.1}$$

3.2.2 Needleman-Wunsch Alignment

For our global alignment, we will proceed very similar to computing the LCS, and will fill a matrix F such that the terms $F_{i,j}$ correspond to the score of aligning the subsequences $x[1..i]$ and $y[1..j]$

We can define a set of boundary conditions for the scoring matrix $F_{i,j}$, namely that the score is 0 at the boundaries so that $F_{i,0} = i \times G$ and $F_{0,j} = j \times G$ for $1 \leq i \leq |x|$ and $1 \leq j \leq |y|$. Define the recurrence relation:

$$F_{i,j} = max \begin{cases} F_{i-1,j} + G & \text{skip a position of } x \\ F_{i,j-1} + G & \text{skip a position of } y \\ F_{i-1,j-1} + S_{x[i],y[j]} & \text{match/mismatch} \end{cases} \tag{3.2}$$

If we incorporate the boundary conditions and recursion relation, we can define some pseudocode for the Needleman-Wunsch Alignment algorithm:

	-	C	T	G	A	C	T	A	G	T	C	A	G	A	G
-	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14
C	-1	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12
G	-2	0	0	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
C	-3	-1	-1	0	0	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	-4	-2	-2	-1	1	0	0	1	0	-1	-2	-3	-4	-5	-6
C	-5	-3	-3	-2	0	2	1	0	0	-1	0	-1	-2	-3	-4
G	-6	-4	-4	-2	-1	1	1	0	1	0	-1	-1	0	-1	-2
A	-7	-5	-5	-3	-1	0	0	2	1	0	-1	0	-1	1	0
A	-8	-6	-6	-4	-2	-1	-1	1	1	0	-1	0	-1	0	0
G	-9	-7	-7	-5	-3	-2	-2	0	2	1	0	-1	1	0	1
A	-10	-8	-8	-6	-4	-3	-3	-1	1	1	0	1	0	2	1
C	-11	-9	-9	-7	-5	-3	-4	-2	0	0	2	1	0	1	1
A	-12	-10	-10	-8	-6	-4	-4	-3	-1	-1	1	3	2	1	0
G	-13	-11	-11	-9	-7	-5	-5	-4	-2	-2	0	2	4	3	2
G	-14	-12	-12	-10	-8	-6	-6	-5	-3	-3	-1	1	3	3	4

Table 3.2: A score matrix for Needleman-Wunsch Alignment.

	-	C	T	G	A	C	T	A	G	T	C	A	G	A	G
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	↖	←	←	←	↖	←	←	←	←	↖	←	←	←	←
G	0	↑	↖	↖	←	←	←	←	↖	←	←	←	↖	←	↖
C	0	↖	↖	↑	↖	↖	←	←	←	←	↖	←	←	←	←
A	0	↑	↖	↑	↖	←	↖	↖	←	←	←	↖	←	↖	←
C	0	↖	↖	↑	↑	↖	←	←	↖	↖	↖	←	←	←	←
G	0	↑	↖	↖	↑	↑	↖	↖	↖	←	←	↖	↖	←	↖
A	0	↑	↖	↑	↖	↑	↖	↖	←	↖	↖	↖	←	↖	←
A	0	↑	↖	↑	↖	↑	↖	↖	↖	↖	↖	↖	↖	↖	↖
G	0	↑	↖	↖	↑	↑	↖	↑	↖	←	←	←	↖	←	↖
A	0	↑	↖	↑	↖	↑	↖	↖	↑	↖	↖	↖	←	↖	←
C	0	↖	↖	↑	↑	↖	↖	↑	↑	↖	↖	←	↖	↑	↖
A	0	↑	↖	↑	↖	↑	↖	↖	↑	↖	↑	↖	←	↖	↖
G	0	↑	↖	↖	↑	↑	↖	↑	↖	↖	↑	↑	↖	←	↖
G	0	↑	↖	↖	↑	↑	↖	↑	↖	↖	↑	↑	↖	↖	↖

Table 3.3: A traceback matrix for Needleman-Wunsch.

Algorithm for Needleman-Wunsch Alignment.

function NEEDLEMANWUNSCH(x, y) ▷ Optimal global alignment between x and y.
  for $i \leftarrow 0$ to $|x|$ **do**
    $F_{i,0} \leftarrow i \times G$
  for $j \leftarrow 0$ to $|y|$ **do**
    $F_{0,j} \leftarrow j \times G$
  for $i \leftarrow 0$ to $|x|$ **do**
    for $j \leftarrow 0$ to $|y|$ **do**
      Match $\leftarrow F_{i-1,j-1} + S_{x[i],y[j]}$
      Insert $\leftarrow F_{i-1,j} + G$
      Delete $\leftarrow F_{i,j-1} + G$
      $F_{i,j} = \max$(Match, Insert, Delete)
  return $F_{|x|,|y|}$

Here is the resulting alignment from performing Needleman-Wunsch alignment:

```
C-GCACGAAGACAG-G
| | ||  || ||| |
CTG-AC-TAGTCAGAG
```

3.2.3 Smith-Waterman Alignment

In most applications we are only interested in aligning a small portion of the sequence to produce a local alignment. Furthermore, we don't necessarily want to force the first and last residues to be aligned. Smith-Waterman is an alignment algorithm that has these properties.

We can define a set of boundary conditions for the scoring matrix $F_{i,j}$, namely that the score is 0 at the boundaries so that $F_{i,0} = F_{0,j} = 0$ for $1 \leq i \leq |x|$ and $1 \leq j \leq |y|$. Define the recurrence relation:

$$F_{i,j} = max \begin{cases} F_{i-1,j} + G & \text{skip a position of } x \\ F_{i,j-1} + G & \text{skip a position of } y \\ F_{i-1,j-1} + S_{x[i],y[j]} & \text{match/mismatch} \\ 0 & \text{zero-out negative scores} \end{cases} \tag{3.3}$$

In addition to the different boundary conditions, a key difference between Needleman-Wunsch (global alignment) and Smith-Waterman (local alignment) is that whereas with the global alignment we start tracing back from the lower right term of the matrix, for the local alignment we start at the maximum value. This value corresponds to the last matched character of the optimal alignment.

Putting all these pieces together, we can produce the following pseudocode for the Smith-Waterman Alignment:

Algorithm for Smith-Waterman Alignment.

```
function SmithWaterman(x, y)                ▷ Optimal local alignment between x and y.
    for i ← 0 to |x| do
        F_{i,0} ← 0
    for j ← 0 to |y| do
        F_{0,j} ← 0
    for i ← 0 to |x| do
        for j ← 0 to |y| do
            Match ← F_{i-1,j-1} + S_{x[i],y[j]}
            Insert ← F_{i-1,j} + G
            Delete ← F_{i,j-1} + G
            F_{i,j} = max(Match, Insert, Delete,0)
    return max{F_{i,j}}
```

	-	C	T	G	A	C	T	A	G	T	C	A	G	A	G
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
G	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1
C	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
A	0	0	0	0	1	0	0	1	0	0	0	2	1	1	0
C	0	1	0	0	0	2	1	0	0	0	1	1	1	0	0
G	0	0	0	1	0	1	1	0	1	0	0	0	2	1	1
A	0	0	0	0	2	1	0	2	1	0	0	1	1	3	2
A	0	0	0	0	1	1	0	1	1	0	0	1	0	2	2
G	0	0	0	1	0	0	0	0	2	1	0	0	2	1	3
A	0	0	0	0	2	1	0	1	1	1	0	1	1	3	2
C	0	1	0	0	1	3	2	1	0	0	2	1	0	2	2
A	0	0	0	0	1	2	2	3	2	1	1	3	2	1	1
G	0	0	0	1	0	1	1	2	4	3	2	2	4	3	2
G	0	0	0	1	0	0	0	1	3	3	2	1	3	3	4

Table 3.4: A matrix for Smith-Waterman, with an optimal path labeled in blue.

	-	C	T	G	A	C	T	A	G	T	C	A	G	A	G
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	↖	←	·	·	↖	←	·	·	·	↖	←	·	·	·
G	0	↑	↖	↖	←	↑	↖	·	↖	←	↑	↖	↖	←	↖
C	0	↖	←	↑	↖	↖	←	·	↑	↖	↖	←	↑	↖	↑
A	0	↑	↖	·	↖	←	↖	↖	←	·	↑	↖	←	↖	←
C	0	↖	←	·	↑	↖	←	←	↖	·	↖	↑	↖	↖	↖
G	0	↑	↖	↖	←	↑	↖	↖	↖	←	↑	↖	↖	←	↖
A	0	·	·	↑	↖	←	↖	↖	←	↖	·	↖	↑	↖	←
A	0	·	·	·	↖	↖	↖	↖	↖	↖	·	↖	↖	↖	↖
G	0	·	·	↖	←	↖	↖	↑	↖	←	←	↑	↖	←	↖
A	0	·	·	↑	↖	←	←	↖	↑	↖	↖	↖	↑	↖	←
C	0	↖	←	·	↑	↖	←	←	↖	↖	↖	←	↖	↑	↖
A	0	↑	↖	·	↖	↑	↖	↖	←	←	↑	↖	←	↖	↖
G	0	·	·	↖	←	↑	↖	↑	↖	←	←	↑	↖	←	↖
G	0	·	·	↖	↖	↑	↖	↑	↖	↖	↖	↖	↖	↖	↖

Table 3.5: Here is the traceback matrix for the Smith-Waterman alignment. Note there are two paths with the same maximum score.

Here are the resulting alignments from Smith-Waterman:

```
GAC-AG   AGACAG-G
||| ||   || ||| |
GACTAG   AGTCAGAG
```

Table 3.6: These two local alignments have the same score of 4. These two alignments correspond to the blue and red paths depicted in Figures 3.4 and 3.5

The Smith-Waterman alignment has two possible paths with the same score. Therefore, both could be output from our algorithm depending on how it is implemented. Therefore, the alignments produced from Needleman-Wunsch and Smith-Waterman are very different.

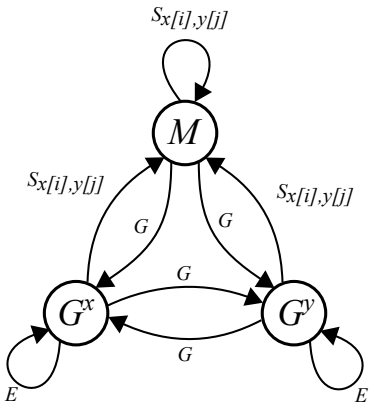


Figure 3.1: Alignment with affine gap penalty can be viewed as three different states: the match state M, the gap in x state G^x, and the gap in y state G^y. Performing the alignment can be viewed as transitions between these states.

3.2.4 Affine Gap Penalties

As stated earlier, there are two widely used gap penalty systems. The linear gap penalty just considers the penalty as a linear multiple of the length of the gap d, and is given by the equation $c_L(d) = d \times G$, where it is assumed that $G < 0$. An "affine" gap penalty is described by two parameters G and E that describe the cost of opening a gap and extending the gap. The equation describing an affine gap penalty $c_A(d) = G + (d-1) \times E$, which includes a gap open parameter $G < 0$ and a gap extension parameter $E < 0$ is difficult to compute using our standard approach for dynamic programming alignments. We can compute it easily by defining an additional matrix to keep track of whether we are in a gap, or outside a gap. To do this, let's create two matrices $G^x_{i,j}$ to hold the value of the best alignment up to positions i and j within a gap in x, and $G^y_{i,j}$ to hold the value of the best alignment up to positions i and j within a gap in y. The matrix $M_{i,j}$ will hold the best alignment value such that the positions i and j are in a match.

$$M_{i,j} = max \begin{cases} M_{i-1,j-1} + S_{x[i],y[j]} & \text{add to the matched region} \\ G^x_{i-1,j-1} + S_{x[i],y[j]} & \text{close the gap in } x \\ G^y_{i-1,j-1} + S_{x[i],y[j]} & \text{close the gap in } y \end{cases} \tag{3.4}$$

$$G^x_{i,j} = max \begin{cases} M_{i-1,j} + G & \text{start a new gap in } x \\ G^x_{i-1,j} + E & \text{extend the gap in } x \\ G^y_{i-1,j} + G & \text{go from gap in } x \text{ to gap in } y \end{cases} \tag{3.5}$$

$$G^y_{i,j} = max \begin{cases} M_{i,j-1} + G & \text{start a new gap in } y \\ G^x_{i,j-1} + G & \text{go from gap in } y \text{ to gap in } x \\ G^y_{i,j-1} + E & \text{extend the gap in } y \end{cases} \tag{3.6}$$

3.2.5 Banded Alignment

In many cases, we want to consider aligning two sequences that are very similar, and we expect all the terms of the optimal alignment to be such that $|i - j| \leq k$ for some k. This approach is called "Banded Alignment". Through this procedeure, we can improve the alignment efficiency of Needleman-Wunsch and Smith-Waterman by banding the comparisons that are made with a distance k from the main diagonal.

This algorithm can run in $\mathcal{O}(kn)$ since it is searching a width $2k + 1$.

We first need to define a function that tests if we are within a distance k from the main diagonal of the alignment matrix.

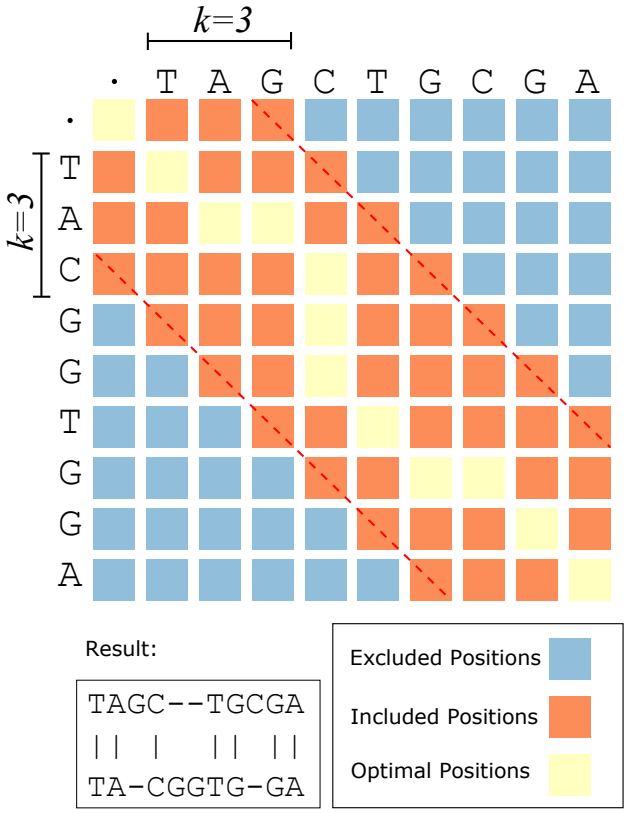


Figure 3.2: A banded alignment for the sequences TAGCTGCGTA and TACGGTGGTA using banded Smith-Waterman using $k = 3$. The blue squares are excluded from the alignment, and the orange squares represent the band that is included. Yellow squares indicate the optimal alignment.

```
function InsideBand(i, j, k)
    if |i − j| ≤ k then
        return "true"
    else
        return "false"
```

The algorithm, which calls this function is then defined as follows:

Algorithm for Banded Alignment. This is banded Needleman-Wunsch, but Smith-Waterman can be done by making the boundary conditions 0, and adding 0 as an additional term in the max function.

```
function BandedAlignment(x, y, k)        ▷ Best score of an alignment within k positions from diagonal
    for i ← 1 to k do
        F_{i,0} ← i × G
    for j ← 1 to k do
        F_{0,j} ← j × G
    for i ← 1 to |x| do
        for d ← −k to k do
            j ← i + d
            if 1 ≤ j ≤ |y| then
                F_{i,j} ← F_{i−1,j−1} + S_{x[i],y[j]}
                if InsideBand(i-1,j,k) then
                    F_{i,j} ← max(F_{i,j}, F_{i−1,j} + G)
                if InsideBand(i,j-1,k) then
                    F_{i,j} ← max(F_{i,j}, F_{i,j−1} + G)
    return F_{|x|,|y|}
```

3.3 Multiple Sequence Alignment

A multiple sequence alignment is an arrangement of the characters in a set if biological sequences that, along with gaps "-" optimize matches according to a scoring system.

```
E-RLYMQVEY
EGRIYMQV-Y
EGRLYLQVEY
EG-MY-QVEY
```

Let's begin by aligning a set of N sequences $\{s_1, s_2, ..., s_N\}$. We must first define the scoring system that will be used to choose the best alignment. One possible scoring system is the "Sum of Pairs" Score, S_{SP}, defined to be the sum of the pairwise scores using our standard scoring matrices for pairwise alignments. For example, the sixth position of the alignment above would have the following score:

$$\begin{aligned} S_{SP}(\{M, M, L, -\}) = S_{M,M} + S_{M,L} \\ + S_{M,-} + S_{M,L} + S_{M,-} + S_{L,-} \end{aligned} \tag{3.7}$$

In general, for a column at position i of the alignment α, we could define a column for that alignment as $\alpha(i)$. The score for this column of the alignment is the sum of the comparisons of pairs of sequences s_x and s_y at that position of the alignment:

$$S_{SP}(\alpha(i)) = \sum_{x<y} S(\alpha_{x,y}(i));$$

where $\alpha(i)$ contains the information for the alignment, and $\alpha_{x,y(i)}$ is the pairwise alignment for s_x and s_y that results from projecting the multiple sequence alignment.

The arrangement of any two sequences in a multiple alignment doesn't necessarily correspond to the optimal pairwise alignment when just the two sequences are considered. In other words, if a multiple sequence alignment is a multi-dimensional object, then projections to two dimensions aren't always optimal for a pair of sequences in the multiple alignment. For example, the multiple alignment here: lexicographical

```
E-RL-QVEY
EGR-IQV-Y
EGRLIQVEY
EG-LIQVEY
```

The first two sequences would be projected as:

```
E-RL-QVEY
EGR-IQV-Y
```

Whereas, since `L` and `I` are similar amino acids the optimal alignment might best be represented as

```
E-RLQVEY
EGRIQV-Y
```

This second alignment would have a positive score from the substitution matrix for $S_{I,L}$ as opposed to two gap penalties. Along with the concept of a multiple sequence alignment comes that of the phylogenetic tree. A phylogenetic tree is a (often binary) tree that represents the evolutionary history of the characters, sequences or species that the tree depicts. The leaf nodes represent the observed sequences, and the internal nodes represent the ancestral sequences.

3.4 Multiple Sequence Alignment

3.4.1 Generalized Dynamic Programming Approach

Let's first consider the dynamic programming approach to multiple sequence alignment. This would involve creating a N-dimensional matrix a, with $L+1$ terms in each dimension, meaning $(L+1)^N$ total entries in the "matrix". Clearly, storing the matrix takes $\mathcal{O}(L^N)$.

Let's use vector notation to refer to the N-tuples so that the vector of positions $\vec{i} = (i_1, ...i_N)$ corresponds to a particular position in our N-dimensional alignment matrix $a[\vec{i}] = a[i_1, ..., i_N]$. Similar to the alignment of two sequences, we will fill up the terms of this matrix $a[i_1, ..., i_N]$, each of which holds the score of the sub-alignment for $s_1[1..i_1], ..., s_N[1..i_N]$. The matrix a would then be initialized such that $a[0, ..., 0] \leftarrow 0$.

Consider how many calculations are required to compute each entry of the matrix. Because each such term depends on the column of the alignment under consideration, there must be $2^N - 1$ entries it depends on. This is because each of the N rows of a column can contain a gap or a character (2 choices), but they can't all contain gaps (hence the "-1").

Note that the case of $N = 2$ is just the Smith-Waterman or Needleman-Wunsch algorithm, in which each position depends on $2^2 - 1 = 3$ positions. Therefore, when computing $F_{i,j}$ for those algorithms, it depends on 3 other cells of the matrix $F_{i-1,j-1}$, $F_{i-1,j}$, and $F_{i,j-1}$.

In addition, it takes $\binom{N}{2} = N(N-1)/2 \sim N^2$ pairwise scores to combine into the S_{SP}. Therefore, the whole computation can take as much as $\mathcal{O}(N^2 2^N L^N)$ steps to compute the optimal alignment.

We can compute the terms of the matrix with the following recursion relation:

$$a[\vec{i}] \leftarrow \max_{\vec{b}\neq\vec{0}} \left\{ a[\vec{i}-\vec{b}] + S_{SP}(\vec{\gamma}(\vec{s},\vec{i},\vec{b})) \right\}$$

where $\vec{b}$ is a non-zero binary vector. For simplicity, let's define a set $\mathcal{B} = \{0,1\}^N \setminus \vec{0}$, and we can then just consider $\vec{b} \in \mathcal{B}$. Therefore $\vec{b}$ ranges over all non-zero binary vectors of N elements. The function $\vec{\gamma}$ corresponds to a column of the alignment such that $(\gamma_1, .., \gamma_N) = \vec{\gamma}(\vec{s},\vec{i},\vec{b})$ and

$$\gamma_j = \begin{cases} s_j[i_j], & \text{if } b_j = 1 \\ -, & \text{if } b_j = 0 \end{cases} \tag{3.8}$$

For each position $\vec{i}$ of our matrix, we will need to consider a number of other cells in the maximum calculation. We can say that the position $\vec{i}$ "depends on" another position $\vec{j}$ if it is used in this maximum calculation. From the previous slide, it appears that the positions we need to consider are $\vec{j} = \vec{i} - \vec{b}$ for some $\vec{b} \neq \vec{0}$. Therefore, it depends on $2^N - 1$ other terms. A useful concept here that is different from previous alignment algorithms is the concept of a pool $\mathcal{P}$ which contains the set of positions under consideration. Initially, this will just contain $\vec{0}$.

When a position $\vec{i}$ is considered, the value $a[\vec{i}]$ is computed, and $\vec{i}$ is added to the pool. When a new position $\vec{i}$ that depends on $\vec{j}$ is added to the pool, its value is initialized by:

$$a[\vec{i}] \leftarrow a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s},\vec{i},\vec{i}-\vec{j}))$$

If it is already in the pool, then it can be updated by

$$a[\vec{i}] \leftarrow \max_{\vec{j}\in\mathcal{P}}(a[\vec{i}], a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s},\vec{i},\vec{i}-\vec{j}))$$

---

function MULTIPLE SEQUENCE ALIGNMENT($s_1, s_2, .., s_N$)
  $\mathcal{P} \leftarrow \{\vec{0}\}$
  while $\mathcal{P} \neq \emptyset$ **do**
    $\vec{j} \leftarrow$ lexicographically lowest in $\mathcal{P}$
    $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\vec{j}\}$
    for all $\vec{i} = \vec{j} + \vec{b}$ for $\vec{b} \in \mathcal{B}$ **do**
      if $\vec{i} \notin \mathcal{P}$ **then**
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{\vec{i}\}$
        $a[\vec{i}] \leftarrow a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s},\vec{j},\vec{i}-\vec{j}))$
      else
        $a[\vec{i}] \leftarrow \max(a[\vec{i}], a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s},\vec{j},\vec{i}-\vec{j}))$
  return $a[|s_1|, |s_2|, ..., |s_N|]$

3.4.2 Reducing the search space: Carrillo and Lipman Algorithm (MSA)

We can reduce our search space by excluding certain positions of the matrix that don't sufficiently contribute to the optimal score, or that don't help achieve some score threshold. The following procedure was been proposed by Carrillo and Lipman in 1988, and implemented in the `MSA` software package [**?**].

As mentioned above, the projection $\alpha_{x,y}$ of the optimal multiple sequence alignment α into two particular dimensions corresponding to sequences s_x and s_y, is not always the optimal sequence alignment for just those two sequences. More precisely, if we define $S^*(s_x, s_y)$ to be optimal pairwise alignment of s_x and s_y, then

$$S(\alpha_{x,y}) \leq S^*(s_x, s_y) \tag{3.9}$$

Let's suppose that we want to restrict our search space for the multiple sequence alignment algorithm described above to only consider alignments α such that $S(\alpha) \geq \theta$. Then, given the definition of the sum of pairs score, we should require

$$S_{SP}(\alpha) = \sum_{x<y} S(\alpha_{xy}) \geq \theta$$

Then this implies that the score for a particular pair of sequences must be such that

$$S(\alpha_{xy}) \geq \theta - \sum_{\substack{(x,y)\neq(j,k) \\ j<k}} S(\alpha_{jk}),$$

but since Equation 3.9 provides a bound on how good each such projection α_{jk} must be, we then can conclude

$$S(\alpha_{xy}) \geq \theta_{xy},$$

where

$$\theta_{xy} = \theta - \sum_{\substack{(x,y)\neq(j,k) \\ j<k}} S^*(s_j, s_k).$$

In conclusion, we can only consider alignments α that are such that each projection α_{xy} has a score that is greater than or equal to θ_{xy}. We can compute this by performing pairwise alignments for each pair of two sequences in our set, and compute a threshold for each pair. Using this threshold, we can also define the range of positions $\vec{i}$ that are relevant to computing the optimal alignment. That is, for any pair of sequences, let $F^{x,y}$ be the scoring matrix computed for the pairwise alignment of x and y (for example, the Needleman-Wunsch scoring matrix). Then, for the pairs of positions i_x and i_y that are part of $\vec{i} = (i_1, ...i_N)$, we can require that

$$F^{x,y}_{i_x,i_y} \geq \theta_{x,y}$$

We can combine this information into the dynamic programming algorithm for multiple sequence alignment described above, with the following

---

function MULTIPLE SEQUENCE ALIGNMENT($s_1, .., s_N, \theta$)
  for all x and y, $1 \leq x < y \leq N$ **do**
    Compute $F^{x,y}$ the total score array for s_x and s_y
  for all x and y, $1 \leq x < y \leq N$ **do**
    $\theta_{x,y} \leftarrow \theta - \sum_{(j,k)\neq(x,y)} S^*(s_j, s_k)$
  $\mathcal{P} \leftarrow \{\vec{0}\}$
  while $\mathcal{P} \neq \emptyset$ **do**
    $\vec{j} \leftarrow$ lexicographically smallest cell in $\mathcal{P}$
    $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\vec{j}\}$
    for all $\vec{i}$ dependent on $\vec{j}$ **do**
      if $F^{x,y}_{i_x,i_y} \geq \theta_{x,y}, \forall(x,y) : 1 \leq x < y \leq N$ **then**
        if $\vec{i} \notin \mathcal{P}$ **then**
          $\mathcal{P} \leftarrow \mathcal{P} \cup \{\vec{i}\}$
          $a[\vec{i}] \leftarrow a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s}, \vec{j}, \vec{i} - \vec{j}))$
        else
          $a[\vec{i}] \leftarrow \max(a[\vec{i}], a[\vec{j}] + S_{SP}(\vec{\gamma}(\vec{s}, \vec{j}, \vec{i} - \vec{j}))$
  return $a[|s_1|, |s_2|, , ..., |s_N|]$

We'll revisit multiple sequence alignment later in the discussion of phylogenetic trees. Before we do that, let's examine other approaches to sequence alignments that can be used in rapidly searching large databases or genomes.

Chapter 4

Sequence Search Algorithms

Extending beyond the concept of aligning sequences, we often want to search for the optimum alignment in the entire genome, or in a database of sequences. Such tasks require specialized algorithms and software to quickly search through a large sequence or database. In some cases, like BLAST, indexing of the sequences are used for rapid retrieval. In other cases, such as Burrows-Wheeler Transform-based alignment and suffix trees, advanced data structures and data compression algorithms are used to speed up the searches.

4.1 BLAST

Probably one of the most widely used software tools in bioinformatics is the Basic Local Alignment Search Tool (BLAST). BLAST implements a number of heuristics that make searching a large database fast and efficient. BLAST, along with FASTA, was among the first software tools for aligning sequences to large databases.

BLAST essentially approximates a Smith-Waterman alignment with the database. A BLAST database is an indexing of K-mers, or "words" from a defined set of sequences. BLAST identifies segment pairs, or two matching words, or two strings of consecutive matching words, from a pair of sequences. One of these sequences is a query sequence and the other is a sequence from our database. Using the statistics outlined in subsection 4.1.2, we can score a particular segment pair as being unlikely due to chance. The most statistically significant segment pairs are called high-scoring segment pairs, or HSPs.

To summarize, in BLAST we have the following terminology:

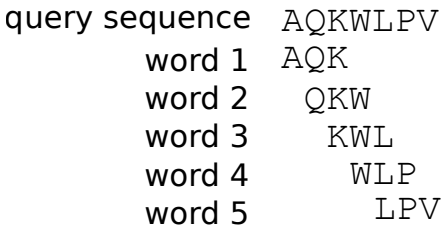
- **database:** A collection of sequences to be searched.
- **segment:** A substring of a sequence.
- **segment pair:** is a pair of segments from two sequences.
- **HSP (high-scoring segment pair):** A pair of segments that have a statistically significant score.

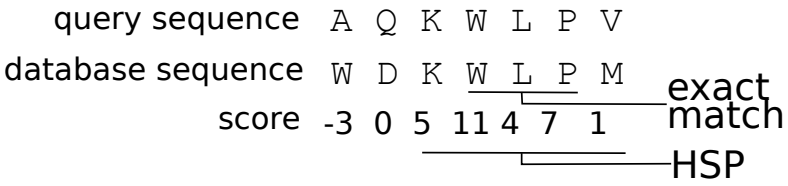
The algorithm of BLAST proceeds as follows. First, low complexity sequences are removed from the query sequences, such as poly(A) and repeated dinucleotides. Then, each of the K-mer words in the query are enumerated in a list. For proteins, one often uses the length $K = 3$ by default. For nucleotides, the word length is often tuned as high as $K = 11$. Then for each word, a score is computed for the 20^K possible K-mer matches. Then as the K-mer in list is scored, the list of possible matches is reduced to only include word matches to a threshold T.

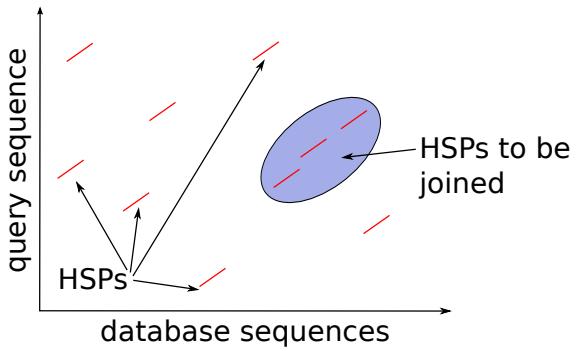
The word matches are then extended to form segments. As each segment is scored, HSPs are identified as meeting a defined statistical threshold. Then exact matches are combined with other exact matches that are with a distance d to connect to a larger alignment.

For a set of HSPs that mean a statistical significance requirement, Smith-Waterman is performed on the individual sequences and this is the final alignment that is reported.

To summarize, the steps of BLAST are as follows:


query sequence AQKWLPV
word 1 AQK
word 2 QKW
word 3 KWL
word 4 WLP
word 5 LPV


query sequence A Q K W L P V
database sequence W D K W L P M
score -3 0 5 11 4 7 1
exact match
HSP


query sequence
database sequences
HSPs
HSPs to be joined

1. Remove low-complexity regions or sequence repeats in the query sequence.
2. Make K-mer word list of the query sequence (Proteins often $K = 3$)
3. List the possible 20^3 matching words and score with a scoring matrix
4. Reduce the list of word matches with threshold T
5. Extend the exact matches to HSPs
6. List all HSPs and evaluate significance
7. Combine two or more HSPs into a longer alignment
8. Report the gapped Smith-Waterman local alignments of the query and each of the matched database sequences.

4.1.1 Statistics of Alignment Scores: Bayesian Model

Bayes Theorem:

$$P(M|x,y) = \frac{P(x,y|M)P(M)}{P(x,y)}$$

Expand the denominator into the two models, the mutation model and random model.

$$P(M|x,y) = \frac{P(x,y|M)P(M)}{P(x,y|M)P(M) + P(x,y|R)P(R)}$$

$$P(M|x,y) = \frac{e^{S+\mu}}{1+e^{S+\mu}}$$

where

$$S = \log\left(\frac{P(x,y|M)}{P(x,y|R)}\right)$$

is basically a log-likelihood ratio, and

$$\mu = \log\left(\frac{P(M)}{P(R)}\right)$$

is a prior log-odds ratio. This equation for $P(M|x,y)$ is a logistic function, and has a "soft-threshold" of μ.

4.1.2 Statistics of Alignment Scores: Extreme Value Distribution

The maximum value in a set of random variables $X_1, ..., X_N$ is also a random variable. This variable is described by the Extreme Value Distribution, also known as the Gumbel Distribution.

$$F(x|\mu,\sigma,\xi) = \exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{-1/\xi}\right\}$$

For $1+\xi(x-\mu)/\sigma > 0$, and with a location parameter μ, a location parameter σ, and a shape parameter ξ. In the limit that $\xi \to 0$, the distribution reduces to:

$$F(x|\mu,\sigma) = \exp\left\{-\exp\left(-\frac{x-\mu}{\sigma}\right)\right\}$$

It can be shown that for N independent variables, and redefining some of our parameters, the CDF for the EVD is:

$$F(x|\mu, \lambda, K, N) = \exp\{-KN\exp(-\lambda(x-\mu))\}$$

Karlin and Altschul showed in 1990 that for or local ungapped alignments of a query sequence of length n, and a database of length m, the CDF for a score S is

$$F(S|\lambda, K, n, m) = \exp\{-Kmn\exp(-\lambda S)\}$$

If you assume that the expected value of the score S obeys a Poisson Distribution, the expected value is given by

$$E(S) = Kmne^{-\lambda S}$$

We can simplify the CDF equation as:

$$F(S|\lambda, K, n, m) = e^{-E(S)}$$

The p-value of the alignment is given by $1 - F(S)$ so that:

$$P(x > S) = 1 - e^{-E(S)}$$

4.2 Burrows-Wheeler Transform

High-throughput sequence data produces hundreds of millions of short (50-200nt) "reads" have a wide variety of applications. These large datasets have necessitated the need for the development of fast alignment algorithms to map these reads to genomic locations. Many of these read mapping programs are based on the Burrows-Wheeler Transform, such as BWA, and *bowtie*. The Burrows-Wheeler transfer is used in data compression, but it has proven to be remarkably useful in read mapping approaches. So it is important to make the distinction that the Burrows-Wheeler Transform isn't a search algorithm, but it and associated data structures can be used for rapidly searching for a substring W within a genome or other large sequence.

4.2.1 Lexigraphical sorting

For our alphabet $\mathcal{A}$, let's introduce a concept of "lexicographical sorting". The meaning here is similar to alphabetic order, but is expanded to include non-alphabetic characters, such as symbols. In most applications to computational biology, the alphabet is the DNA alphabet and the characters are the DNA bases. We can imagine that there is a precise ordering $b_1 < b_2 < ... < b_n$, for all $b_i \in \mathcal{A}$. For convenience, let's use a notation such that it is understood that for a character $b \in \mathcal{A}$, the character $b + 1$ is the character that is next in the lexicographical sorting, so $b = b_i < b_{i+1} = b + 1$. For a collection of sequences, we can define a similar ordering by first comparing the first character, then breaking ties with the second character, and so on.

4.2.2 The Search Text

Consider a sequence x of length $|x| = n$ from an alphabet $\mathcal{A}$. We can think of x as a genome to be searched, or more generally, any text that we would like to search. We can construct another string $x\$$ with length $|x\$| = n+1$ within the alphabet $\mathcal{A} \cup \{\$\}$ as the concatenation of x with \$, and where the symbol \$ is designed to be lexicographically (or alphabetically) less than all other characters in $\mathcal{A}$. Such a sequence is said to be $\$-$terminated.

4.2.3 Suffix Array

From our previous notation, a suffix of $x\$$ starting at position i is given by $x\$[i..n+1]$. A suffix array S of the sequence $x\$$ is defined such that $S[i]$ is the start of the ith "lexicographically lowest" suffix (including the one that consists of just the terminal character \$). As an example, let's consider the word $x\$ = ACACGC\$$. The ordered list of suffixes and their corresponding suffix array values are shown in Figure 4.1.

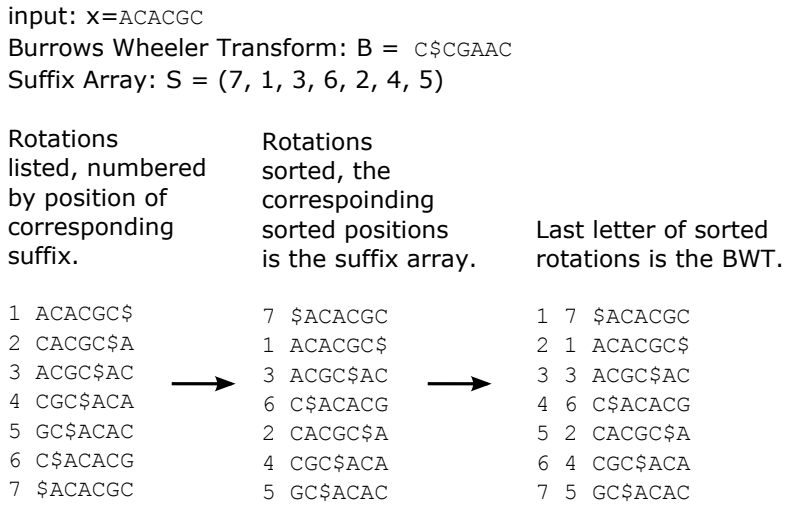


Figure 4.1: To compute the BWT and suffix array from a sequence x, one can first list all rotations x, then sort them lexicographically. The concatenation of the last letter of the sorted rotations is the BWT. The suffix array corresponds to the list of positions $S[k]$ that are the start of the kth smallest suffix.

The Burrows-Wheeler Transform (BWT) B of x is defined such that $B[i] = \$$ when $S[i] = 1$ and $B[i] = x\$[S[i]-1]$ otherwise. Therefore, it would appear that B[i] is the character preceding the ith smallest suffix, except for the suffix starting at $S[i] = 1$, in which case $B[i] = \$$ [4]. Clearly, $|B| = |x\$| = |S| = n+1$. Although it may seems like a strange rearrangement of the characters of the sequence x, this sequence B has a lot of useful characteristics, including that you can use it to search the original sequence x for any search word W in $\mathcal{O}(|W|)$ time once a few data structures are created, including the suffix array S.

To compute the BWT, we can create a matrix consisting of the different rotations of the input text $x\$$. Rotations in sequence analysis are created by remove the first character, and appending it to the end of a sequence. This process can be repeated to create each row of the matrix. Then sort the rows of the matrix lexicographically and hereafter refer to it as M. The last column of the matrix is the BWT B. We can define the first column of the matrix F. Clearly, F is the lexicographical sorting of the input string $x\$$. Clearly, $F[i] = x\$[S[i]]$

Consider the sequence $x = ACACGC$. After creating the string $x\$ = ACACGC\$$, we get the resulting suffix array $S = (7, 4, 1, 6, 3, 5, 2)$, and the BWT $B = C\$CGAAC$ (See figure 4.1).

Each column of the matrix M is a different permutation of the BWT, B. The first column F is the result of lexicographically sorting B. We may be interested in the mapping of a particular position i with corresponding character $b = B[i]$, to the position j that this character occupies in F, such that $F[j] = b$. Let's call this mapping $\phi(i) = j$. One of the nice properties of the BWT is that such a mapping can be readily computed after creating two data structures.

Let's define the function $C(b)$, such that it returns the number of occurrences of any of the characters lexicographically less than b in the text $x\$$, so the number of occurrences of any character in the set $\{\$, ..., b-1\}$. It is clear that because F is sorted, the first occurrence of the character b in F is $C(b)+1$, and the last occurrence of b in F is $C(b+1)$. Therefore, the function $C(B[i])$ takes us most of the way to $\phi(i)$. The remaining number of positions we should consider to get all the way to $\phi(i)$ corresponds to the number of occurrences of b in the suffix $B[1..i]$. To understand this, let's consider the rows $M[i]$ of the matrix M. The matrix will begin with $\$$ at the beginning of $M[1]$. What about the rows that begin with b? They will be all together compactly in our matrix because the rows are sorted:

$$M = \begin{pmatrix} M[1] \\ \dots \\ \dots \\ \dots \\ M[C(b)+1] \\ M[C(b)+O(B[i],i)] \\ M[C(b)+O(B[i],i+1)] \\ M[C(b+1)] \\ \dots \\ M[i-1] \\ M[i] \\ M[i+1] \\ M[i+2] \\ \dots \\ \dots \end{pmatrix} = \begin{pmatrix} \$ & \dots & \dots \\ \dots & \dots & \dots \\ X & \dots & b \\ \dots & \dots & \dots \\ bX & \dots & \dots \\ bW & \dots & \dots \\ bW & \dots & \dots \\ bY & \dots & \dots \\ \dots & \dots & \dots \\ W & \dots & a \\ W & \dots & b \\ W & \dots & b \\ W & \dots & c \\ Y & \dots & b \\ \dots & \dots & \dots \end{pmatrix} \tag{4.1}$$

They will begin with the row at position $C(b)+1$ and end with the row at $C(b+1)$. Since the rows are sorted and have the same first character, then the suffixes $M[i][2..n+1]$ resulting from removing the first character (b) must also be sorted. Therefore, the ordering of the rows that start with b is the same as the ordering of occurrences of b in B. Therefore, the rank R_b of the rows $M[i]$ that start with b must be equal to the number of occurences of b in $B[1..i]$. Let's then define the number of occurrences of b in $B[1..i]$ as $O(b,i)$. Thus, we have the equation:

$$\phi(i) = C(B[i]) + O(B[i], i) \tag{4.2}$$

It should be clear that no other column of M has this property. It may also be not as straightforward to define such a mapping for other columns in the matrix.

4.2.4 Burrows-Wheeler for Exact Search

As stated above, for any character b in $x\$$, the first occurrence of b in F is $C(b)+1$, and the last occurrence is $C(b+1)$. We can call these bounds a "Suffix Array interval", or SA-interval for the character b, because the interval $[C(b)+1, C(b+1)]$ defines a range of the indices i for the suffix array S corresponding to these occurrences of b. More generally, let's define the interval $[\underline{R}(W), \overline{R}(W)]$ as the SA-interval of the word W. The equation 4.2 suggests a relationship between the SA-interval for the word W and for the concatenation of the character b with the word W.

$$\underline{R}(bW) = C(b) + O(b, \underline{R}(W) - 1) + 1 \tag{4.3}$$

$$\overline{R}(bW) = C(b) + O(b, \overline{R}(W)) \tag{4.4}$$

Note that for an empty string, $\overline{R}(W) = n+1$ and $\underline{R}(W) = 1$, and that $\underline{R}(W) \leq \overline{R}(W)$ if and only if W is a substring of x. This equation was first discovered by Ferragina and Mandini [5].This result means that one can determine if W is a substring of x and count the number of occurrences of W in x in $\mathcal{O}(|W|)$ by computing $\underline{R}(W)$ and $\overline{R}(W)$ one character at a time, and iterate the equation above.

To understand these equations, let's first consider the equation for $\underline{R}(bW)$. As stated above, the first occurrence of b in F is at position $C(b)+1$. Among the occurrences of a, some of them are followed by W, and some of them aren't (assuming that the string bW is actually in $x\$$). The insight from the derivation of Equation 4.2 tells us that the ordering of these rows must be the same as the ordering of the occurrences of b in B. Therefore, the first row $M[i]$ of M that begins with W ends with b, must correspond to the row $M[j]$ that begins with aW through the mapping $\phi(i) = j$ (see Equation 4.1).

Consider the rows of M that begin with b. The number of rows that begin with b that precede the first row that begins with bW must be $O(b, \underline{R}(W) - 1)$ because this gives the number of occurrences of b in B that precede something lexicographically less than W in $x\$$. In fact, it is the number of occurrences of b that precede suffixes strictly less than W in $x\$$. Therefore, the first occurrence of aW, if it exists, must be exactly one more than $C(b) + O(b, \underline{R}(W) - 1)$, as indicated by Equation 4.3.

Putting it all together, we can formulate an algorithm for exact search using the data structures associated with exact search, called the Ferragina and Manzini Algorithm [5].

Ferragina and Manzini Algorithm: An algorithm for exact search of a sequence W with the BWT.

function FERRAGINA-MANZINI ALGORITHM($W, C(b), O(b,i)$)
  $i \leftarrow |W|$
  $b \leftarrow W[i]$
  $k \leftarrow C(b) + 1$
  $\ell \leftarrow C(b+1)$
  while $k \leq \ell$ and $2 \leq i$ **do**
    $b \leftarrow W[i-1]$
    $k \leftarrow C[b] + O(b, k-1) + 1$
    $\ell \leftarrow C[b] + O(b, \ell)$
    $i \leftarrow i - 1$
  return $[k, \ell]$

4.2.5 Burrows-Wheeler for Inexact Search

We can pre-compute $C(b)$ and $O(b,i)$ for all $b \in \mathcal{A}$ from the reference sequence x once and for all, and also compute $O'(b,i)$ for the reverse (not complement) of the reference sequence x'. Once these are computed, we can compute compute SA Intervals of substrings that match W in x with no more than d mismatches with the *InexactSearch* algorithm presented below:

function INEXACTSEARCH(W, d)
  CalculateD(W)
return *InexRecur*($W, |W|, d, 1, |x|$)

This function is expressed in terms of two more functions that we will now define. Let's first define a value $D(i)$ as the lower bound on the number of differences d needed to find $W[1..i]$ in x. Pre-computing $D(i)$ will save us some time later on by avoiding searching for mismatched matches that aren't there. Let's first define an algorithm *CalculateD* for computing this function $D(i)$.

function CALCULATED(W) ▷ Compute $D(i)$
  $k \leftarrow 2$ ▷ initialize lower bound at beginning of x'
  $\ell \leftarrow |x|$ ▷ initialize upper bound at end of x'.
  $d \leftarrow 0$ ▷ initialize $d = D(i)$ to 1.
  for $i \leftarrow 1$ to $|W|$ **do**
    $k \leftarrow C(W[i]) + O'(W[i], k-1) + 1$
    $\ell \leftarrow C(W[i]) + O'(W[i], \ell)$
    if $k > \ell$ **then** ▷ If not a substring, re-initialize for larger d.
      $k \leftarrow 2$
      $\ell \leftarrow |x|$
      $d \leftarrow d + 1$
    $D(i) \leftarrow d$

Note that with *CalculateD*(W) we are doing the search in the reverse order than what we have done with exact search above. The idea is that we are finding the number of mismatches needed to find $W[1..i]$. For the actual search, we will proceed in the other direction, but this distance threshold $D(i)$ tells us if it is worth to continue decreasing i. If you can't find $W[1..i]$, then the search is reset, and k and ℓ are put back to their initial values, d is incremented, and the search continues beginning with $W[i+1]$.

It should be noted that *CalculateD*(W) is not optimal in the sense that it will give an upper bound $D(i)$ for the number of mismatches needed to find $W[1..i]$, but if it can't be found, it will continue to search for

$W[i+1..|W|]$. For example, it could find $W[1..i]$ and $W[i+1..|W|]$ in the search text x, but not check how many characters are between the instances of these two substrings.

Next, the function $InexRecur(W, i, d, k, l)$ recursively calculates the SA intervals of substrings that match $W[1..i]$ with no more than d differences on the condition that suffix $W[i+1..|W|]$ matches interval $[k, l]$. The idea is the search begins with an number d of allowed mismatches. In this sense, d serves as an "allowance". When no exact match is found, and an insertion, deletion, or mismatch is required to proceed, the value of d is decreased by 1. When the allowance d has run out, the search finishes empty handed.

```
function InexRecur(W, i, d, k, ℓ)
    if d < D(i) then return ∅
    if i < 1 then return {[k, ℓ]}
    I ← ∅                                              ▷ Initialize to the empty set.
    I ← I ∪ InexRecur(W, i − 1, d − 1, k, ℓ)           ▷ gap in x
    for each b ∈ {A, C, G, T} do
        k ← C(b) + O(b, k − 1) + 1
        ℓ ← C(b) + O(b, ℓ)
        if k ≤ ℓ then
            I ← I ∪ InexRecur(W, i, d − 1, k, ℓ)       ▷ gap in W
            if b = W[i] then
                I ← I ∪ InexRecur(W, i − 1, d, k, ℓ)   ▷ match
            else
                I ← I ∪ InexRecur(W, i − 1, d − 1, k, ℓ)   ▷ mismatch
    return I
```

Because this is not an exact search, it necessarily requires that the result is the union of a number of SA-intervals, thus the result does not typically represent one contiguous range of positions. The approach is recursive, and explores different "paths" that correspond to various mismatches and gaps, and the resulting intervals I are passed back up the recursion, and a union is made with other SA-intervals.

4.2.6 Prefix Trie

A "prefix trie" is constructed such that the edges are labeled with letters from the alphabet $\mathcal{A}$ and the string concatenation of the edge letters on the path from a leaf to the root gives a unique prefix of x. String concatenation of the edge symbols from a node to the root gives a unique substring of x. The nodes of the prefix trie are labeled with the SA-Intervals of the unique sequence created when you concatenate the edge symbols along the path to the root. With a prefix trie, one can test whether a query W is an exact substring of x by equivalently finding a node that represents W, and this can be done in $\mathcal{O}(|W|)$. See Figure 4.2.

Input x$ = ACACGC$
Burrows Wheeler Transform: B = C$CGAAC
Suffix Array: S = (7, 1, 3, 6, 2, 4, 5)

k	S(k)	
1	7	$ACACGC
2	1	ACACGC$
3	3	ACGC$AC
4	6	C$ACACG
5	2	CACGC$A
6	4	CGC$ACA
7	5	GC$ACAC

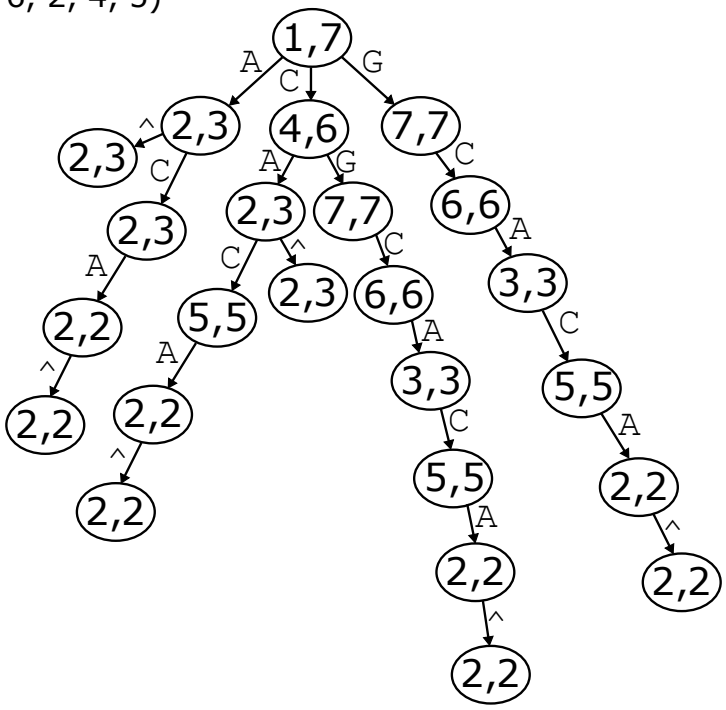


Figure 4.2: A prefix trie for the text $x\$ = ACACGC\$$.

The value of the prefix trie is it helps to visualize the process in inexact search with the BWT. The process of the $InexactSearch(W, d)$ algorithm above can be viewed as traversing the prefix trie, and exploring down the different branches. When the distance d has decreased to 0, then the branch is exhausted and result propagates up.

4.3 Suffix trees

A suffix tree of a string x is such that there is a path from the root to a leaf node for each suffix of the sequence x. The leaf nodes contain a number, which is the start position of the corresponding suffix. Therefore, there are $n = |x|$ leaf nodes.

Similar to the prefix trie, we add a termination symbol \$ that doesn't exist in the input string. This is so that repeating occurrences of a substring can be distinguished.

Consider searching the query string $q =$ `TA` in the string $x =$ `TAGTAC`. Once the suffix tree is created in order $\mathcal{O}(|x|)$ time, it then takes $\mathcal{O}(|q|)$ time to search for q.

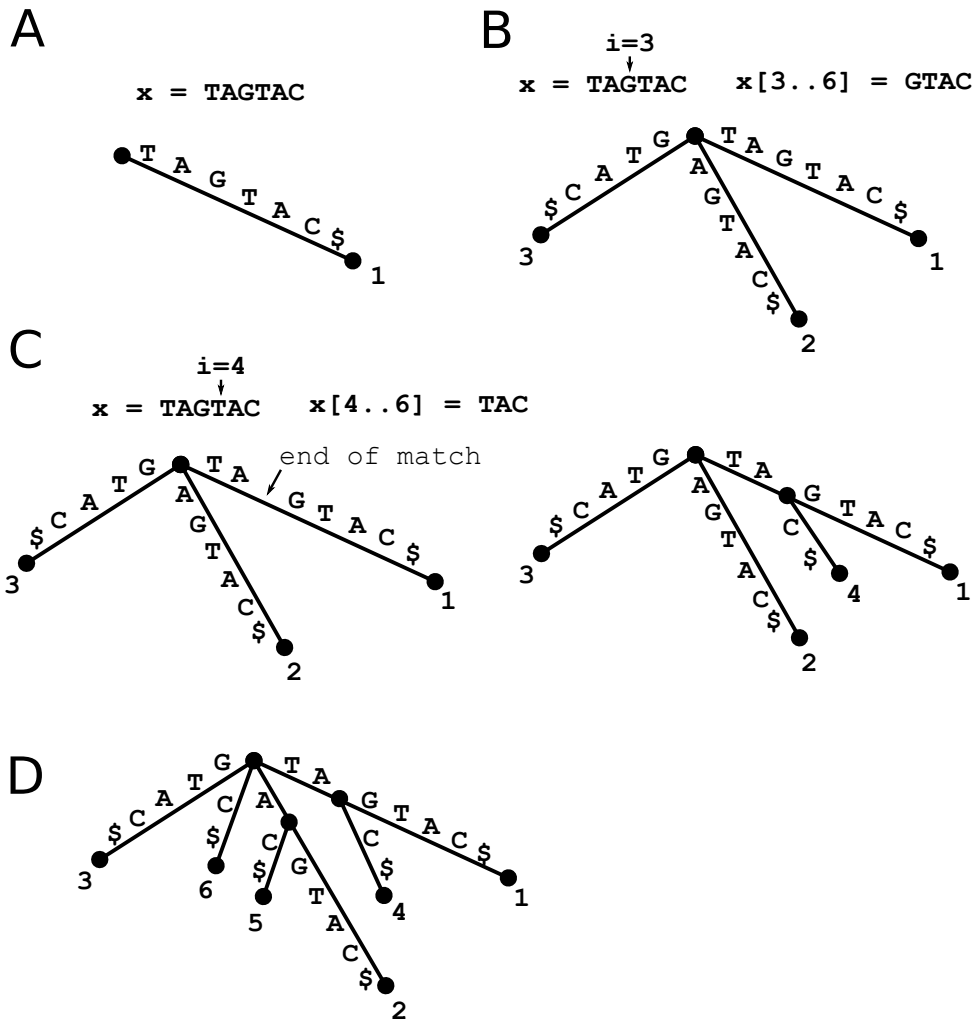
In this example, the query $q =$ `TA` occurs at positions 1 and 4 of the string $x =$ `TAGTAC`. Likewise, the branch starting with `TA` has leaf nodes labeled with 1 and 4.

One can build a suffix tree by an iterative approach. First, an edge is added for the suffix $x[1..|x|]$ starting at position 1, which is the entire string.

Then one adds the suffix $x[i..|x|]$ for $i = 2, ..., |x|$ until the suffix tree is complete.

Let's expand upon this approach.

Define the single-edged tree consisting of the single edge from the root to a leaf labeled 1 as $\mathcal{T}_1$.



We can construct the tree $\mathcal{T}_{i+1}$ from the tree $\mathcal{T}_i$.

We identify a connection point on the tree $\mathcal{T}_i$ by finding the longest path from the root whose edge label matches a prefix of $x[i+1..|x|]$. One continues down the branch until the point at which no matching characters are found. Since each path out of a node starts with a different character, there can be only one path that matches.

When the end of the match of $x[i+1..|x|]$ is reached, it is either in the middle of an edge, or at a node. If it is at a node, then a new edge out of this node is created, such that the edge is labeled with the remaining characters of $x[i+1..|x|]$ that do not match.

If the end of the match is in the middle of an edge, a new node is created at this point, and a new edge is created such that the remaining characters of $x[i+1..|x|]$ that don't match label this new edge coming out of the new node. Finally, the leaf node at the end of the new edge is labeled with $i+1$.

Chapter 5

Phylogenetics Algorithms

5.1 Phylogenetic Trees

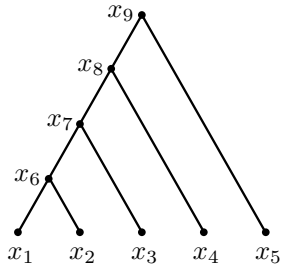


Figure 5.1: An example of a binary phylogenetic tree.

Having the phylogenetic tree can not only tell you about the relationship between taxa under consideration, but can also inform how one computes a multiple sequence alignment. For example, you may be more tolerant of allowing gaps and mismatches for more distantly related sequences. Let's explore some algorithms and concepts for building phylogenetic trees, and later touch on how they can be used to guide multiple sequence alignment.

5.1.1 Binary Trees

Consider a tree with N leaves such that each internal node has two child nodes, denoted left and right. Such a tree is called a **binary tree**.

Typically, we think of the nodes of the tree representing sequences or individual characters. The leaf nodes correspond to observed characters from a set of species or genes, and the internal nodes correspond to ancestral species or genes. Importantly, we can think of the length of the branches as corresponding to

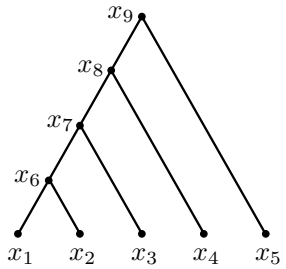


Figure 5.2: A binary tree.

the edit distance between sequences. The distance to an internal node would then correspond to the edit distance to the hypothetical common ancestor.

If there are N leaves and the tree has a root, then there must be $2N-1$ nodes including both the internal nodes and the leaves. For example, the tree in Figure 5.2 has five leaf nodes, and has nine total nodes, consistent with the equation because $2N - 1 = 2 \times 5 - 1 = 9$.

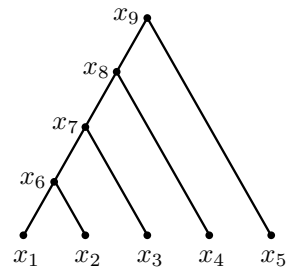
This equation is generally true because because each internal node has exactly two child nodes for a binary tree. The simplest binary tree just has two leaf nodes and one internal node (the root). If you imagine adding internal nodes to this tree, for each new internal node there is a "bifurcation", where one edge splits into two, then you would need $N - 1$ such splitting events, to end up with n leaves. A tree with three leaf nodes must have two internal nodes, the root node and an additional internal node, corresponding to the bifurcation event that produced the third leaf node. We can continue this process for N leaf nodes, giving us $N - 1$ internal nodes. Therefore, there are $2N - 1$ total nodes (both internal and leaf nodes) for a rooted, binary tree with N leaf nodes.

5.1.2 Tree traversal

Another important concept is that of tree traversal. When dealing with trees, it is helpful to define an ordering with which to visit nodes. The more common approach for the algorithms we will examine in this course are post-order traversal. The idea is that are recursive function is defined such that the children are visited first, with the left child followed by the right child, and then the root (or internal node) under examination is visited. In doing so, we always visit the leaf nodes first, and then the internal nodes. The order of the internal nodes, however, depends on the structure of the tree. This scheme is depicted in Figure 5.3.

---

function POSTORDER(x)
  postOrder($left(x)$)
  postOrder($right(x)$)
  visit(x)



post-order traversal:
$x_1 \to x_2 \to x_6 \to x_3 \to x_7$
$\to x_4 \to x_8 \to x_5 \to x_9$
pre-order traversal:
$x_9 \to x_8 \to x_7 \to x_6 \to x_1$
$\to x_2 \to x_3 \to x_4 \to x_5$

Figure 5.3: An example of pre- and post-order traversal. In post-order traversal, children are visited first. In pre-order traversal, they are visited after the parent nodes.

In contrast, pre-order traversal visits the node under examination first, followed by the children. For each of these orderings, the name implies the order, and whether the node itself is visited before (pre) or after (post) its children.

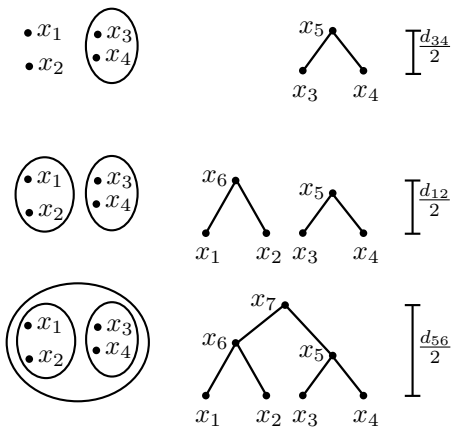


Figure 5.4: A depiction of the approach taken from UPGMA.

---

function PREORDER(x)
  visit(x)
  preOrder($left(x)$)
  preOrder($right(x)$)

5.2 Algorithms for building Phylogenetic Trees

5.2.1 UPGMA: Unweighted pair group method using averages

Unweighted pair group method using averages (UPGMA) is probably the most straightforward algorithm for building a phylogenetic trees from a set of sequences. Perhaps in its simplicity, it lacks the complexity to deal with certain exceptions that we will discuss below. The idea is that we will build nested "clusters" of characters that corresponding to nodes in our phylogenetic tree. The procedure begins by assigning each sequence to its own single-element cluster corresponding to leaf nodes. Next, the most similar clusters are merged, and doing so will be associated with creating an internal node above the nodes in the tree.

Beginning with our singleton clusters, we define pairwise distances between each of these clusters as the edit distance between the sequences. Pairs of clusters can be grouped, by grouping the closest two clusters first. When two clusters are compared, one can use the average distance between each sequence:

$$d_{ij} = \frac{1}{|C_i||C_j|} \sum_{p \in C_i, q \in C_j} d_{pq}$$

As two clusters are grouped, you can define the resulting cluster as the union of the two combined clusters. You can compute the new distance between C_k and all the other clusters as the average distance from each other cluster and C_i and C_j.

$$d_{kl} = \frac{d_{il}|C_i| + d_{jl}|C_j|}{|C_i| + |C_j|}$$

UPGMA Algorithm

function UPGMA(d_{ij} for all pairs of sequences i and j) ▷ Compute T
  for $i \leftarrow 1, ..., N$ **do**
    Assign each sequence to its own single element set C_i
    Assign each sequence to a leaf node at 0
  while More than two clusters **do**
    Find the minimum d_{ij}
    Create new cluster $C_k = C_i \cup C_j$
    Define new average distances as $d_{kl} = \frac{d_{il}|C_i|+d_{jl}|C_j|}{|C_i|+|C_j|}$
    Define internal node k at height $d_{ij}/2$
    Remove clusters C_i and C_j
  Set root at height $d_{ij}/2$ for last two clusters C_i and C_j

UPGMA and the Molecular Clock

One of the assumptions of UPGMA is that sequences evolve at a constant rate. This usually isn't the case for biological sequences, but is a good approximation under certain constraints, such as the rate of mutation for the 3rd position of codons.

One can test if this "molecular clock" assumption is true with the **ultrametric condition**. Under this condition, for any three sequences x_1, x_2, x_3, then the pairwise distances $d(x_1, x_2) = d_{12}, d(x_1, x_3) = d_{13}, d(x_2, x_3) = d_{23}$ must be either all equal (think equilateral triangle) or two must be equal and the other one smaller (think acute isosceles triangle).

This condition is equivalent to saying that from any node, if we add the lengths in any path to a leaf, we should get the same value.

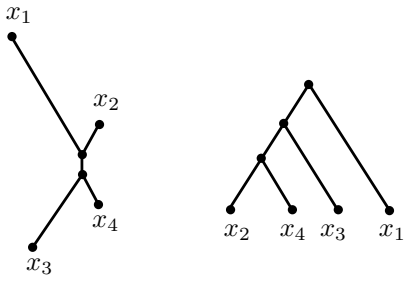


Figure 5.5: An example of the situation where UPGMA fails. In this case, this tree is additive, but not ultrametric. The greedy approach of UPGMA would try and treat x_2 and x_4 as children of the same internal node because they are closer.

Additivity: an assumption that the distance between any two leaves is the sum of the lengths of the paths connecting them.

This assumption is made by UPGMA. It is possible to have trees that are additive, but fail the ultrametric condition. Neighbor-joining addresses these situations.

5.2.2 The Neighbor Joining Algorithm

It turns out, for a tree with distances d_{ij} that are additive, one can construct new distances D_{ij} where being minimal implies that i and j are neighboring nodes (sister phyla). D_{ij} is give by:

$$D_{ij} = d_{ij} - (r_i + r_j)$$

where r_i is the average distance to each other leaf node, given by

$$r_i = \frac{1}{|L| - 2} \sum_{k \in L} d_{ik}$$

The Neighbor-Joining approach involves building a tree T, while simultaneously removing nodes from a set of "active nodes" L, which initially is the set of leaf nodes. For example, when a new internal node h is added, the distance between h and other leaves k is given by:

$$d_{hk} = \frac{1}{2}(d_{ik} + d_{jk} - d_{ij})$$

and the distances on the branches of the tree are $d_{ik} = \frac{1}{2}(d_{ij} + r_i - r_j)$, and $d_{jk} = d_{ij} - d_{ik}$. These equation can be understood as the proper equations to satisfy additivity (See Figure 5.6). Effectively, we can grow the tree, starting with the leaves and joining neighbors by adding internal nodes. As we remove leaves from L, they remain in T, but we connect the leaves in T with internal nodes.

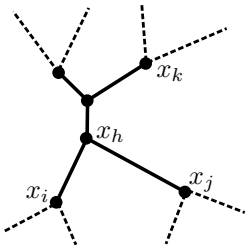


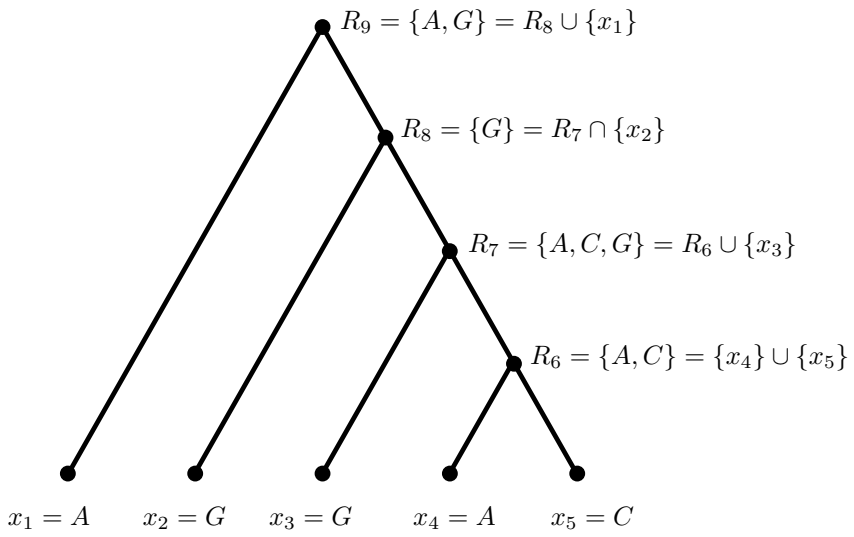
Figure 5.6: For any three nodes i, j, and k, consider a node h that is connected to all of them, possibly through other internal nodes. We can use additivity to compute the distance between each point. In this example $d_{ik} = d_{ih} + d_{hk}$, $d_{jk} = d_{jh} + d_{hk}$, and $d_{ij} = d_{ih} + d_{hj}$. We can combine these equations and solve for $d_{hk} = \frac{1}{2}(d_{ik} + d_{jk} - d_{ij})$. This result is used in the Neighbor Joining Algorithm.

Neighbor-Joining Algorithm

function NEIGHBOR-JOINING ALGORITHM(L,d_{ij} for all pairs of sequences i and j) ▷ Compute T
  Initialize the tree T to just be the set of leaf nodes L
  Compute r_i for all $i \in L$ using $r_i = \frac{1}{|L|-2} \sum_{j \in L} d_{ij}$
  Define $D_{ij} = d_{ij} - (r_i + r_j)$ for all i and j.
  while $|L| > 2$ **do**
    Select i and j from L with minimal D_{ij}
    Define new node h, setting $d_{hk} = \frac{1}{2}(d_{ik} + d_{jk} - d_{ij})$ for all other nodes $k \in L$
    Add h to T with edge lengths $d_{ih} = \frac{1}{2}(d_{ij} + r_i - r_j)$, $d_{jh} = d_{ij} - d_{ih}$
    Remove i and j from L, add h to L
    Recompute r_i using the updated L with $r_i = \frac{1}{|L|-2} \sum_{j \in L} d_{ij}$
    Update $D_{ij} = d_{ij} - (r_i + r_j)$ for all i and j.
  Connect remaining nodes k and m with distance d_{km}

Choosing a root

The UPGMA method computes a root position as part of the algorithm. For neighbor-joining, the root must be added the tree is constructed. One method is to add a sequence to the initial set that is an **outgroup**. This sequence is chosen to be evolutionarily most distant from all the other sequences. Hence the branch coming out of the unrooted tree corresponding to the outgroup is taken as a proxy for the root.



5.3 Evaluating Phylogenetic Trees

5.3.1 Bootstrapping Phylogenetic Trees

Trees can be assessed with the bootstrap as follows: Given a dataset D consisting of a multiple sequence alignment, where the individual sequences correspond to leaf nodes of our tree $\mathcal{T}$, we can randomly sample columns from the alignment **with replacement**, and a tree is built with this artificial data and evaluated with some metric. Then this process is then repeated some number of times (thousands).

For example, you could assess the percentage of samples that exhibit a particular feature, like a branching event of interest.

5.3.2 Parsimony

Parsimony is the principle of choosing the tree that explains the observed sequences with the minimal number of substitutions.

This approach computes a cost for trees, rather than building them from distances. Therefore, it is necessary to search all tree topologies (usually not practical) or sample them through some heuristic sampling procedure. We want to develop a formalism for evaluating trees based on their topologies and branch lengths in terms of the sequences that are represented by the leaf nodes, and inferred ancestral sequences at the internal nodes.

In the simplest case, we would like to count the number of substitutions C associated with all the branches of the tree. Let's consider a single position m of the sequences. We define a sequence x_k for each node, and we want to evaluate position m by comparing $x_k[m]$ for each node k. Let us also define R_k as the set of possible characters that could be at the position of interest in the internal node k. Since this corresponds to an unobserved ancestral character, we would consider those that are consistent with the observed sequences.

For considering ancestral sequences (internal nodes) k that is the parent node of i and j, we want it to be consistent with observed characters for the nodes i and j.

If R_i corresponds to the set of possible characters observed at position m of node i, and R_j is the set of possible characters for position m of node j, then the most parsimonious assumption is that $R_k = R_i \cap R_j$, which implies that the ancestral sequence consists of what is in common to the observed sequences. That being said, sometimes this intersection is null, which corresponds to one or more mutations along the branches leading from k to i and j. In which case, we can define $R_k = R_i \cup R_j$ and add to our substitution count C, because such a situation must include a substitution along the considered beaches.

function PARSIMONY ALGORITHM($\mathcal{T}$, with leaf nodes $L = \{x_1, ..., x_n\}$)
  Initialize $C = 0$
  for all k in post-order traversal **do**
    if $k \in L$ **then**
      Set $R_k = \{x_k[m]\}$
    if $k \notin L$ **then**
      Let i,j be the child nodes of k
      if $R_i \cap R_j \neq \emptyset$ **then**
        $R_k = R_i \cap R_j$
      else
        $R_k = R_i \cup R_j$, and increment C
  return C

5.3.3 Weighted Parsimony

A variant of Parsimony is "Weighted Parsimony". In this case, rather than a counting the number of substitutions C, we define a cost function $C_k(a)$ for the cost of assigning character a to node k.

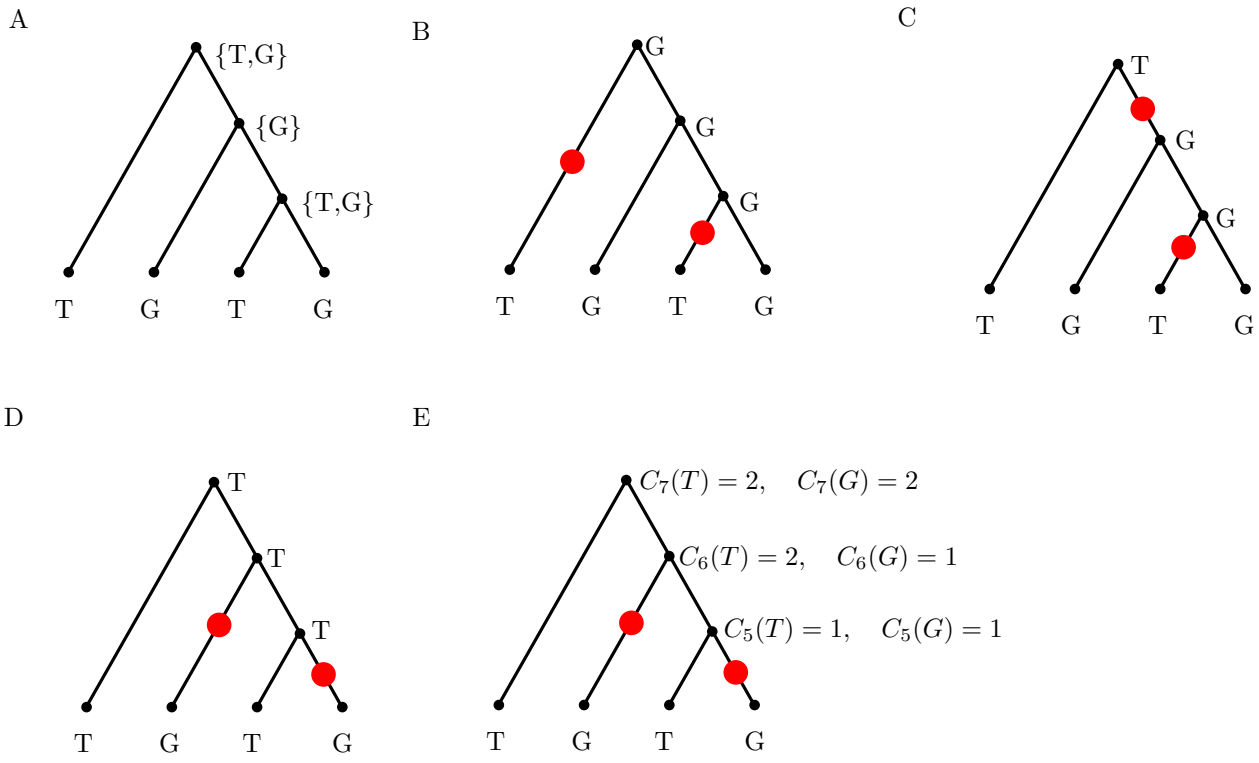


Figure 5.7: Weighted Parsimony. A. The solution from Maximum Parsimony provides the possible characters at each internal node in this example. B. One possible evolutionary history is to have all G for the internal nodes, requiring two mutations indicated by large red circles. C. A second solution is allowable with Maximum Parsimony, which has Ts for two of the internal nodes. D. The solution containing all Ts for the internal nodes is excluded using Maximum Parsimony, because the middle internal node can only be G. E. Weighted Parsimony provides a solution that indicates that the cost of the trees, as indicated by the cost at the root node is the same for both possibilities, allowing T at the middle internal node.

To do this, we need to define a cost matrix $C_{a,b}$, that gives the cost of substituting a for b. This could encode the chemical similarity of amino acids, or nucleotide substitution rates, for example.

---

function WEIGHTED PARSIMONY ALGORITHM($\mathcal{T}$, with leaf nodes $L = \{x_1, ..., x_n\}$)
  for all k in post-order traversal **do**
    if $k \in L$ **then**
      Set $C_k(a) = 0$ for $a = x_k[m]$, $C_k(a) = \infty$ otherwise
    if $k \notin L$ **then**
      Let i and j be the daughter nodes of k
      for all a **do**
        $C_k(a) = \min_b(C_i(b) + C_{a,b}) + \min_b(C_j(b) + C_{a,b})$
  return $\min_a C_{2n-1}(a)$

We can define pointers going from a parent node k to the left and right child nodes:

$$l_k(a) = \arg\min_b(C_i(b) + C_{a,b})$$

$$r_k(a) = \arg\min_b(C_j(b) + C_{a,b})$$

These pointers will allow us to define a set of paths to the leaf nodes, once the optimal ancestral character a is determined by minimizing $C_{2n-1}(a)$ over all a.

5.3.4 Maximum Likelihood

Alternatively, you could compute the probability of the tree given the data $P(\mathcal{T}|D)$.

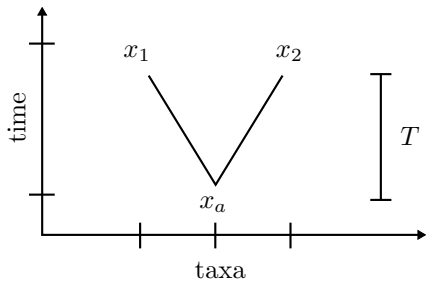


Figure 5.8: A simple tree with only two leaf nodes. Such as situation illustrates the core calculation used in Maximum Likelihood.

Consider a simple case where we have two leaf nodes x_1 and x_2, depicted in Figure 5.8. Let's consider position m of these sequences. The branch lengths leading to x_1 and x_2 are t_1 and t_2 respectively. The likelihood we should consider is given by:

$$P(x_1[m], x_2[m], a|T, t_1, t_2) = p_a P(x_1[m]|a, t_1)P(x_2[m]|a, t_2)$$

where a is an ancestral character. Since we don't know the ancestral character, we should consider summing over all possible values of a:

$$P(x_1[m], x_2[m]|T, t_1, t_2) = \sum_a p_a P(x_1[m]|a, t_1)P(x_2[m]|a, t_2)$$

For the full alignment, you can take the product over the individual columns of the alignment:

$$P(x_1, x_2|T, t_1, t_2) = \prod_{m=1}^{N} P(x_1[m], x_2[m]|T, t_1, t_2)$$

The terms that we need to expand upon are the individual probabilities $P(b|a,t)$, which describe the likelihood of going from $a \to b$ in the time t. There are numerous models for this that vary the rates of going from certain classes of nucleotides/amino acids to others. In general, we can describe this by a matrix of probabilities:

$$P = e^{Rt}$$

Where R is a matrix of rates.
For example, the rate matrix for the Jukes-Cantor model is as follows:

$$R = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} -3\alpha & \alpha & \alpha & \alpha \\ \alpha & -3\alpha & \alpha & \alpha \\ \alpha & \alpha & -3\alpha & \alpha \\ \alpha & \alpha & \alpha & -3\alpha \end{pmatrix} \end{array}$$

It is useful to consider $\mathcal{T}_k$, the subtree of all terms of $\mathcal{T}$ below the node k. Let $P(\mathcal{T}_k|a)$ be the probability of all the of the subtree below k, given that the character at k is a.

Then for a parent node k, with child nodes i and j, then we have the recursion relation:

$$P(\mathcal{T}_k|a) = \sum_{b,c} P(b|a,t_i)P(\mathcal{T}_i|b)P(c|a,t_j)P(\mathcal{T}_j|c)$$

which sums over all possible child node character states b and b at i and j, respectively. Using this equation, we should be able to recursively compute the probability of a tree using the Felsenstein Algorithm.

Felsenstein's Algorithm

function FELSENSTEIN'S ALGORITHM($\mathcal{T}$, with leaf nodes $L = \{x_1, ..., x_n\}$)
  for all k in post-order traversal **do**
    if $k \in L$ **then**
      Set $P(\mathcal{T}_k|a) = 1$, if $a = x_k[m]$, and $P(\mathcal{T}_k|a) = 0$ otherwise
    if $k \notin L$ **then**
      $P(\mathcal{T}_k|a) = \sum_{b,c} P(b|a,t_i)P(\mathcal{T}_i|b)P(c|a,t_j)P(\mathcal{T}_j|c)$
  return Likelihood $P(\mathcal{T}|D) = \sum_a P(\mathcal{T}_{2n-1}|a)p_a$

5.4 Tree-based Alignment Algorithms

5.4.1 Progressive Alignment

Progressive Alignment is a method for multiple sequence alignment that builds the multiple alignment by successively adding pairwise alignments.
Variants of this method can vary, for example by:

1. The order with which pairwise alignments are incorporated into the multiple alignment
2. What each progression includes: some grow by including sequences, others by aligning alignments to each other
3. Whether tree structure is used to build up the alignment
4. Scoring systems

Feng-Doolittle Progressive Alignment Algorithm

First thing is that Feng and Doolittle do is define a distance used to build a tree from the scores. For each pair of sequences, a distance is computed as:

$$D = -\log S_{eff} = -\log\left(\frac{S_{obs} - S_{rand}}{S_{max} - Srand}\right)$$

S_{eff}: the effective score that results from the normalization.

S_{obs}: the observed pairwise alignment score.

S_{rand}: the expected score attained when aligning two sequences of the same length and residue composition.

S_{max}: the maximum score attained when each sequence is aligned to itself.

Feng-Doolittle Progressive Alignment Algorithm

1. Construct a matrix of all the $N(N-1)/2$ distances between each pair of sequences
2. Construct a guide tree from the distance matrix using the Fitch-Margoliash clustering algorithm
3. Align the child nodes of each node as it is added to the tree

Each alignment of child nodes could involve aligning two sequences, aligning a sequence and an alignment, or aligning two alignments. For profile aligning profiles, relative entropy could be used in the comparison.

ClustalW

ClustalW: An implementation of progressive profile Alignment

1. Construct a distance matrix of all $N(N-1)/2$ pairs of sequences. Distances are computed from scores by the Kimura 83 model.
2. Construct a guide tree by the neighbor-joining clustering algorithm
3. Progressively align at nodes in order of decreasing similarity using sequence-sequence, sequence-profile, and profile-profile alignment

- The substitution matrix is chosen on the basis of the similarity expected and the evolutionary distance between sequences (For example PAM1 for close sequences and PAM250 for distance sequences)
- The gap penalties are position-specific. For example, the gap penalties are reduced if the position is within 5 or more consecutive hydrophobic residues.
- Both gap-open and gap-extend parameters are increased in regions that don't have a lot of gaps, forcing the gapped regions to cluster in nearby positions.
- If the score of an alignment is low during the progressive alignment portion, the guide tree may be adjusted to defer the low scoring alignment until more profile information has been computed.

Barton-Sternberg multiple alignment

1. Find the pair of sequences with the highest similarity, and use this alignment to begin the multiple alignment.
2. Find the sequence that is most similar to the profile of the alignment of the first two, an align it with profile-sequence alignment. Repeat until all sequences are aligned.
3. Remove sequence x_1 and realign it to the profile of the other aligned sequences $x_2, ..., x_N$. Repeat this process for each other sequence.

4. Repeat the re-alignment process either a fixed number of times, or until the alignment score converges.

One of the problems with progressive alignments is that it might heavily depend on the more similar sequences. However, in practice, there are some positions that are strongly conserved, and other cases that are less conserved. The more strongly conserved positions probably need to be scored more stringently and the less conserved positions should be scored less stringently.

A Profile alignments split the alignments up by position, recognizing that some parts of an alignment may require different parameters than others. For example, splitting an alignment from 1 to N into alignments from 1 to n and from $n+1$ to N.

In iterative alignment, sequences can be removed and realigned to a profile of the other sequences.

5.5 Measures of Multiple Sequence Alignment quality

Some other measure of the quality of a multiple sequence alignment are Information Content, and Relative Entropy.

The entropy of a random variable X that is described by a probability distribution $P(x_i)$ for a set of K discrete events is given by:

$$H(X) = -\sum_{i=1}^{K} P(x_i) \log P(x_i)$$

Information content can be described as the reduction in entropy:

$$I(X) = -\Delta H = H_{before} - H_{after}$$

Relative entropy describes the divergence or difference between two distributions. For two distributions P and Q, the relative entropy is given by:

$$H(P||Q) = \sum_{i=1}^{K} P(x_i) \log \left(\frac{P(x_i)}{Q(x_i)} \right)$$

Also called Kullback-Leibler divergence.

Chapter 6

Motif Finding

A biological sequence motif is a pattern that has many instances in a set of sequences under consideration. For a set of input sequences $S = \{s_1, s_2, ..., s_N\}$, we could define a motif as a set of K-mers in these input sequences that fit some set of conditions. One one extreme, we could require all such instances of our motif to match the sequence `TGACGTCA` exactly, and at the other extreme, we might have a probabilistic model that describes the probability of each possible nucleotide at each possible position of our motif. In other cases, we might consider a motif that consists of all K-mers that are within 2 mismatches from the sequence `TGACGTCA`. Other approaches define a consensus sequences using degenerate characters as defined in Table 1.1

The common pattern is found without performing a sequence alignment as described in previous chapters. More general approaches are needed to find a short pattern that could repeat any number of times.

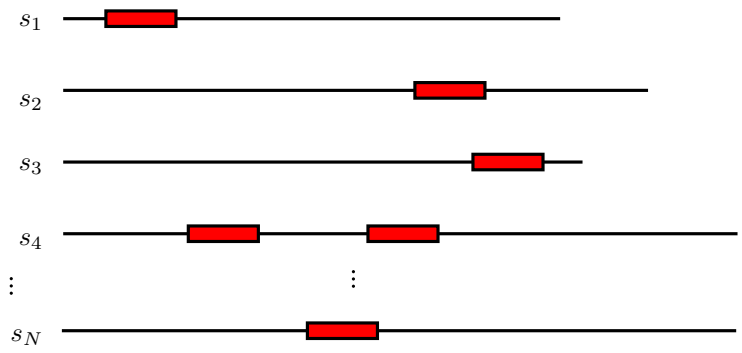


Figure 6.1: An illustration of motif finding Given a set of sequences $S = \{s_1, s_2, ..., s_N\}$, we seek to find a set similar sequences, which are the motif instances indicated by red rectangles, without prior knowledge of what the motif looks like.

6.1 Combinatorial Motif Finding

Given a set of sequences $S = \{s_1, s_2, ..., s_N\}$, construct a graph $G(S, K, d)$, where the vertices are substrings of length K of the input sequences denoted $s_{ij} = s_i[j..j + K - 1]$, i.e. the substring of length K within the input sequence s_i starting at position j.

Connect a vertex s_{ij} to another vertex s_{pq} when $i \neq p$ and if $d(s_{ij}, s_{pq}) \leq d$. For simplicity, let $d(x, y) \equiv d_H(x, y)$ be the Hamming distance, which allows only substitutions, but insertions or deletions could in theory be added in a generalization of this approach.

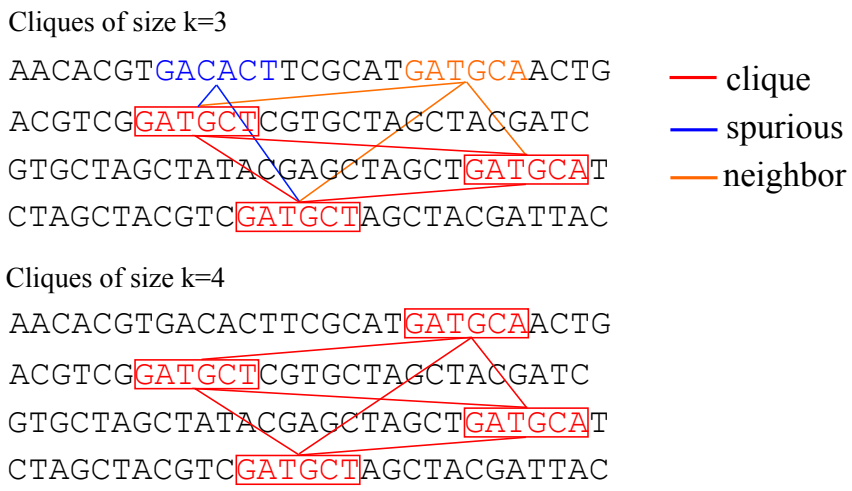


Figure 6.2: As the size of the cliques k increases, spurious edges are identified and removed.

Let's define a (K, d)-signal as a motif in the data such that the occurrences of the motif are of length K and are within d mismatches of "central" sequence. Since every two sequences x and y that are instances of the signal are such that $d(x, y) \leq 2d$, one could say $(x, y) \in G(S, K, 2d)$

For N sequences, one can make $\binom{N}{2}$ sequence comparisons and enumerate all possible pairs of subsequences that differ by at most $2d$ mismatches.

Simulations suggest that for the case of $(15, 4)$-signals, only a small portion of these pairs are from the true signal. (Pevzner and Sze 2000). Such a motif finding problem in sequences of length 600nt is often called "Pevzner's Challenge".

A clique in a graph is a subgraph such that any two vertices in it are connected by an edge. One could say that a (K, d)-signal in S corresponds to clique of size N in the $G(S, K, 2d)$ graph. Therefore, this type of motif finding is equivalent to finding large cliques in $G(S, K, 2d)$

Winnowing is an approach to finding (K, d)-signals by focusing on spurious similarities rather than the signal itself. The WINNOWER algorithm constructs a graph of vertices corresponding to substrings, and edges corresponding to similar substrings. Spurious edges are removed.

6.1.1 The WINNOWER Algorithm

Let's define a vertex w as a **neighbor** of a clique $C = \{v_1, v_2, ..., v_{|C|}\}$ if $C \cup \{w\}$ is also a clique in the graph.

An clique is **extendable** if it has at least one neighbor in every part of the multipartite graph G.

An edge is called **spurious** if it does not belong to any extendable clique of size k.

One approach to building large cliques is as k grows, delete all spurious edges to ensure that only extendable cliques remain in the final graph.

WINNOWER uses a more stringent method and operates on the observation that every edge in a maximal N-clique belongs to at least $\binom{N-2}{k-2}$ extendable cliques of size k.

Consider the example in Figure 6.3 A, where there are $N = 7$ input sequences, and the K-mers presented as blue boxes depict the instances of the (K, d)-signal, and therefore, all these K-mers would constitute an N-clique depicted with red edges. The arrow indicates a particular edge under examination. Removing that edge would create vertices that are not part of the resulting $N - 2$-clique, depicted as gray in Figure 6.3 B. Among the $N-2$ vertices that still form a clique, depicted as purple, a set of any $k-2$ of them would be such that returning the removed edge would result in a k-clique. For example, for $k = 4$, we could select $\binom{5}{2=10}$ pairs of purple vertices, and any one of those pairs of vertices would become a k-clique upon the return of

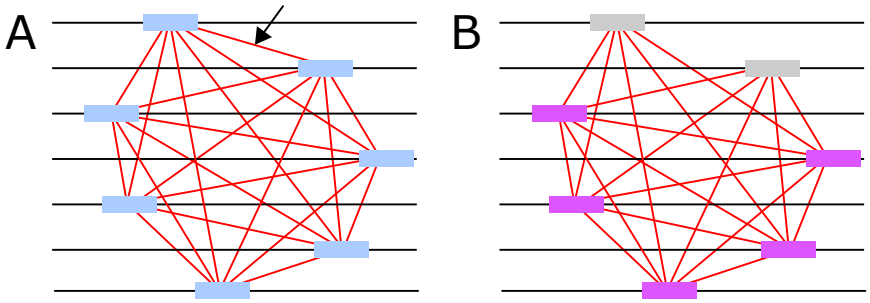


Figure 6.3: **A.** The arrow indicates a particular edge under examination of an N-clique. **B.** Removing that edge would create vertices that are not part of the resulting $N-2$-clique, depicted as gray. Among the $N-2$ vertices that still form a clique, depicted as purple, a set of any $k-2$ of them would be such that returning the removed edge would result in a k-clique.

the removed edge. Therefore, the removed edge is part of $\binom{N-2}{k-2}$ cliques of size k, all of which are extendable. The same applies to any other examined edge, because we are dealing with a completely connected graph.

Based on this observation, WINNOWER defines **inconsistent** edges as those that belong to less than $\binom{N-2}{k-2}$ extendable cliques of size k.

We can define a data structure x_{ijpq} to be 0 if the j-th letter in sequence s_i is the same as the q-th letter of s_p.

If the number of mismatches between s_{ij} and s_{pq} is defined as d_{ijpq}, then it must be the case that:

$$d_{i,j+1,p,q+1} = d_{ijpq} - x_{ijpq} + x_{i,j+K,p,q+K}$$

This relationship enables one to construct the graph $G(S,K,d)$ in $\mathcal{O}(L^2)$ time, where $L = \sum_i |s_i|$ is the total length of all the sequences.

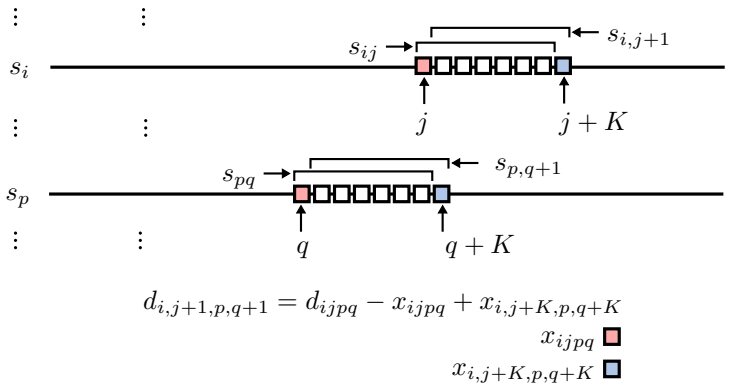


Figure 6.4: We can efficiently compute the graph $G(S,K,d)$ using the fact that K-mers overlap by $K-1$ positions.

Given a set of sequences $S = \{s_1, s_2, ..., s_N\}$, construct a graph $G(S,K,d)$, where the vertices are substrings of length K of the input sequences denoted $s_{ij} = s_i[j..j+K-1]$, i.e. the substring of length K within the input sequence s_i starting at position j.

Let's consider the challenge of finding a (K,d)-signal, a pattern defined by a string of length K such that the instances of the pattern are with d mismatches from from this string.

Let's define the distance between a pattern P and a particular sequence s_i from our input sequences S to be

$$d(P, s_i) = min_{1 \leq j \leq n-K+1} d(P, s_{ij})$$

So note that this is a minimum and picks out the best matching K-mer to the pattern P. In addition, let's define a distance between the pattern and the set of sequences S to be the sum given by:

$$d(P, S) = \sum_{i=1}^{N} d(P, s_i)$$

A pattern-centric method (Staden 1989, Tompa 1999) to find a median string P that minimizes $d(P, S)$ would involve enumerating all 4^K K-mers, but such a method is computationally infeasible as K increases.

Another approach (Li, Ma, and Wang 1999) computes all possible combinations of r substrings of length K. This method also becomes difficult as r increases.

Let's explore the effectiveness of the case where $r = 1$, which would be the most computationally practical value of r, or searching for

$$P^* = \min_{P \in S} d(P, S)$$

Let P be a (K, d)-signal such that the best matches are $P_1, ..., P_N$, each occurring within to the corresponding input sequences $s_1, ..., s_N \in S$.

According to Pevzner and Sze, P^* is often unrelated to the (K, d)-signal and often a random pattern W is such that $d(W, S)$ ls less than $min_{1 \leq i \leq N} d(P_i, S)$.

If the best matches to P_i are $P_1, ..., P_N$ for each of the input sequences $s_1, ..., s_N$, then we expect that $d(P_i, S) \approx 2d(N - 1)$.

However, it is also highly likely that a random pattern W will also be such that $d(W, S) \approx 2d(N - 1)$. So in a motif search, it is likely that a random pattern W would be selected instead of P_i.

To resolve this, Pevzner and Sze presented SP-STAR that selects patterns as follows.

6.1.2 SP-STAR

Let W be a K-mer within our dataset S and $W_1, ..., W_N$ be the best matches to W. Then SP-STAR uses a sum-of-pairs distance function defined as

$$D(W, S) = \sum_{1 \leq i < j \leq N} d(W_i, W_j)$$

The SP-STAR distance function $D(W, S)$ can delineate random sequences from the pattern P because $d(W_i, W_j)$ can be as high as $4d$ because each instance W_i is on average a distance of $2d$ from W.

For the instances of the pattern P, however, each instance varies by as much as $2d$ from each other, by virtue of being an instance of a (K, d)-signal.

The SP-STAR algorithm first finds a pattern W that minimizes $D(W, S)$.

Then SP-STAR the "majority string" W' such that at each position i, W' has the most frequently occurring letter at position i in all the instances $W_1, ..., W_N$. The best sequence W' in each of the input sequences s_i define a new set of sequences $W'_1, ..., W'_N$ with which to compute $D(W, S)$.

This procedure is repeated until the score does not improve any more.

6.2 Probabilistic Motif Finding

6.2.1 Background Models

In order to find motifs, we need to first have a model of what a motif does not look like, that is, a model of the "background". This is essentially the same idea as the random model discussed in protein scoring matrices.

For DNA motifs, the zeroth order model would be to just use the single nucleotide frequencies, $\{p_A, p_C, p_G, p_T\}$. Under such a system, the probability of a sequence x is just the product of the single nucleotide frequencies.

$$P(x) = \prod_{i=1}^{|x|} p_{x[i]}$$

We can also have models that take into account higher-order sequence features such as dinucleotides, trinucleotides and beyond. For example consider a model where the probability of a nucleotide depends on the preceding nucleotide in the sequence. We would describe such a model by the conditional probabilities:

$$P(x[i]|x[i-1])$$

and the probability of a sequence x of length $|x| = n$ is written as:

$$P(x) = P(x[n]|x[n-1])P(x[n-1]|x[n-2])...P(x[2]|x[1])$$

Such a model is the first order Markov chain. The Markov chain of order m is such that the probability of a nucleotide depends on the previous m nucleotides. Such a probability is written $P(x[i]|x[i-1], x[i-2], x[i-3], ..., x[i-m])$.

6.2.2 Information Content

It is helpful to consider quantities that will help define how "predictable" a probabilistic model is. For example, we already talked about the complexity of the a sequence. Now consider a set of short sequences that have been aligned, and we want to compute the number of possible columns that have that arrangement of characters defined by the nucleotide counts n_A, n_C, n_G, n_T. The multinomial coefficient gives us this information. Taking the log would give :

$$\log\left(M(\vec{n})\right) = \log\left(\frac{N!}{\prod_b n_b!}\right)$$

using the Stirling's approximation, we can make the approximation $\log(n!) \approx n\log(n) - n$, thus resulting in

$$\log\left(M(\vec{n})\right) = N \times \left(-\sum_b p_b \log(p_b)\right)$$

where $p_b = n_b/N$. The quantity $H = -\sum_b p_b \log(p_b)$ is known as the Shannon Entropy, and describes the uncertainty of this column. When one of the characters is present in 90% of the instances, the entropy is low, indicating that there is more certainty in the motif. Alternatively, when $p_b = \frac{1}{4}$ for all $b \in \{A, C, G, T\}$, then entropy is at a maximum:

$$H_{bg} = 4 \times -\frac{1}{4}\log_2\left(\frac{1}{4}\right) = 2\text{bits}$$

One can interpret the units of "bits" as the number of questions one must ask to determine the outcome of a particular base. For example, consider the questions "Is it a purine?" if yes, then as "is it A?", otherwise ask "is it C?". Either way, the unknown base will be determined.

For a sequence-specific motif, we require position-dependent probabilities $f_{b,i}$ describing the probability of nucleotide b at position i.

$$f = \begin{pmatrix} f_{1A} & f_{2A} & f_{3A} & \cdots & f_{\ell A} \\ f_{1C} & f_{2C} & f_{3C} & \cdots & f_{\ell C} \\ f_{1G} & f_{2G} & f_{3G} & \cdots & f_{\ell G} \\ f_{1T} & f_{2T} & f_{3T} & \cdots & f_{\ell T} \end{pmatrix}$$

The Information Content at each position i is defined as the negative change in entropy associated with going from the initial state *initial* to the final state $final$:

$$I(i) = -\Delta H(i) = H_{initial}(i) - H_{final}(i)$$

In the case of motif finding, we are going from the background model as our initial state, to the motif model M as the final state.

$$I(i) = -\Delta H(i) = H_{bg}(i) - H_M(i)$$

where

$$H_M(i) = -\sum_{b=A}^{T} f_{b,i} \log_2 f_{b,i}$$

Therefore, an increase in information is understood as the decrease in entropy associated with going from the background model to the motif model. One useful construct for dealing with motifs is the indicator matrix $x_{b,i}$:

$$x_{b,i} = \begin{cases} 1, & \text{if } x[i] = b \\ 0, & \text{if } x[i] \neq b \end{cases}$$

In terms of the indicator function defined in equation 1.1, we have the relationship

$$x_{b,i} = \mathbb{1}_b(x[i])$$

For example, let's say we have a set of motif instances $M = \{x_1, x_2, ..., x_{|M|}\}$ for our motif. We can construct our count matrix as a sum over the indicator matrices x_{jbi} associated with all of the instances x_j of the motif in the set M.

$$C_{b,i} = \sum_{j=1}^{|M|} x_{jbi}$$

And we can then use this count matrix to construct an estimate of the motif's probability matrix

$$f_{b,i} = \frac{C_{b,i}}{\sum_{b=A}^{T} C_{b,i}}$$

A very important concept for motif finding is the weight matrix:

$$W_{b,i} = \log\left(\frac{f_{b,i}}{p_b}\right)$$

This matrix is used to score a sequence x to determine if it is an instance of the motif or not. Note that the terms of this matrix form a log-likelihood, much like the protein scoring matrices. A significantly high log-likelihood score, given by

$$S(x) = \sum_{i=1}^{\ell} \sum_{b=A}^{T} x_{b,i} W_{b,i} = \sum_{i=1}^{\ell} \sum_{b=A}^{T} x_{b,i} \log\left(\frac{f_{b,i}}{p_b}\right)$$

means that x is an instance of the motif. So a specific sequence such as

$$x = \texttt{CGTAAGGT}$$

has a score given by

$$S(x) = W_{1C} + W_{2G} + W_{3T} + W_{4A} + W_{5A} + W_{6G} + W_{7G} + W_{8T}$$

As before with alignments, we are going to want to talk about likelihoods and posterior probabilities. The likelihood for the motif model is given by

$$P(x|M) = \prod_{i=1}^{\ell} \prod_{b=A}^{T} f_{b,i}^{x_{bi}}$$

And for our background model

$$P(x|B) = \prod_{i=1}^{\ell} \prod_{b=A}^{T} p_{b}^{x_{bi}}$$

Each of these states has an associated prior $P(M)$ and $P(B) = 1 - P(M)$
Now we are in a position to construct the posterior probability:

$$\begin{aligned} P(M|x) &= \frac{P(x|M)P(M)}{P(x|M)P(M) + P(x|B)P(B)} \\ &= \frac{\prod_{i,b} f_{b,i}^{x_{bi}} P(M)}{\prod_{i,b} f_{b,i}^{x_{bi}} P(M) + \prod_{i,b} p_{b}^{x_{bi}} (1 - P(M)))} \end{aligned} \tag{6.1}$$

We are interested in the posterior probability because it tells us how likely it is for the given sequence x to be an instance of the motif. In contrast, the likelihood $P(x|M)$ can be thought of as the probability of a given sequence given that you already know it's an instance of the motif. Thinking of a generative model, we can imagine the motif "emitting" sequences, each generated with a probability $P(x|M)$.

6.2.3 Gibb's Sampling

The motif finding problem is: given a set of sequences $S = \{s_1, ..., s_N\}$, discover a motif of length K defined by a set of positions $j_1, ..., j_N$ that define instances of the motif by the subsequences $s_1[j_1..j_1 + K - 1], ..., s_N[j_N..j_N + K - 1]$. In this description of the problem, we assume that each sequence exactly one instance of the motif. Although this is an assumption made in the most basic motif finding approaches, it is relaxed in the general case where we can have any number (including zero) of instances in each of the sequences. In the simplest versions of Gibb's Sampling, this assumption holds true.

The Gibb's Sampling approach can be described as follows:

1) First choose a set of initial positions at random $j_i = rand(1..|s_i| - K + 1)$ for $i = 1, ..., N$

2) Build a weight matrix $W_{b,i}$ and associated score function $S(x)$ from the subsequence defined by these positions

3) Update the positions such that the score is maximized:

$$j_i = \arg\max_j S(s_i[j..j + K - 1])$$

4) Repeat steps 2 and 3 until a convergence condition is met.

6.2.4 Expectation Maximization

One of the challenges of motif finding is identifying the optimal parameters for our motif model. In what preceded, we can call the motif with the frequency matrix f and the prior probability P_M.

Let us define the parameters for our model as $\theta = (f, P_M)$.

In what follows we will present an algorithm that chooses the parameters θ that maximizes the expected value of the likelihood of the data x given the model $P(x|\theta)$. We will keep in mind the motif finding problem, but what follows is more generally applicable.

In general, we would like to identify an updated parameter set $\theta^{(t+1)}$ that improves our likelihood over the current parameters $\theta^{(t)}$. This is equivalent to maximizing the log-likelihood since the logarithm is a monotonic function.

$$\log\left(P(x|\theta^{(t+1)})\right) - \log\left(P(x|\theta^{(t)})\right) > 0$$

The key idea of Expectation Maximization (EM) is the introduce the concept of the missing data z. In the case of the motif finding problem, this could represent the state of being or not being an instance of the motif.

The likelihood, for a given value of θ can be written as the sum over all possible values of z as:

$$P(x|\theta) = \sum_z P(x, z|\theta)$$

However such a sum is often too intensive to compute, so one computes z conditioned on the data x and current set of parameters θ

Let's expand the likelihood in terms of z using Bayes' Theorem.

$$P(x|\theta) = \frac{P(x, z|\theta)}{P(z|x, \theta)} \tag{6.2}$$

Plugging this into the log likelihood gives us

$$\log\left(P(x|\theta)\right) = \log\left(P(x, z|\theta)\right) - \log\left(P(z|x, \theta)\right)$$

Now let's express this in terms of our desired parameters $\theta^{(t+1)}$

$$\log\left(P(x|\theta^{(t+1)})\right) = \log\left(P(x, z|\theta^{(t+1)})\right) - \log\left(P(z|x, \theta^{(t+1)})\right)$$

Multiplying both sides of the previous equation by $P(z|x, \theta^{(t)})$, which is the probability of z conditioned on our data x and current set of parameters $\theta^{(t)}$ and summing over all possible values of z gives:

$$\begin{aligned}\sum_z P(z|x, \theta^{(t)}) \log\left(P(x|\theta^{(t+1)})\right) = &\sum_z P(z|x, \theta^{(t)}) \log\left(P(x, z|\theta^{(t+1)})\right) \\ &- \sum_z P(z|x, \theta^{(t)}) \log\left(P(z|x, \theta^{(t+1)})\right)\end{aligned} \tag{6.3}$$

The left hand side of the equation is just the log likelihood $\log\left(P(x|\theta^{(t+1)})\right)$ because it is independent of z and the sum would sum to 1.

Therefore, our likelihood expression can then be written as:

$$\begin{aligned}\log\left(P(x|\theta^{(t+1)})\right) = &\sum_z P(z|x, \theta^{(t)}) \log\left(P(x, z|\theta^{(t+1)})\right) \\ &- \sum_z P(z|x, \theta^{(t)}) \log\left(P(z|x, \theta^{(t+1)})\right)\end{aligned} \tag{6.4}$$

To improve our parameters, we require that:

$$\log\left(P(x|\theta^{(t+1)})\right) - \log\left(P(x|\theta^{(t)})\right) > 0$$

but we have shown that this difference of log likelihoods is equal to

$$Q(\theta^{(t+1)}|\theta^{(t)}) - Q(\theta^{(t)}|\theta^{(t)}) + \sum_z P(z|x,\theta^{(t)}) \log\left(\frac{P(z|x,\theta^{(t)})}{P(z|x,\theta^{(t+1)})}\right) > 0$$

where

$$Q(\theta^{(t+1)}|\theta^{(t)}) = \sum_z P(z|x,\theta^{(t)}) \log\Big(P(x,z|\theta^{(t+1)})\Big)$$

And the last term of the inequality is the relative entropy, which is always greater than or equal to zero

$$H(\theta^{(t)}|\theta^{(t+1)}) = \sum_z P(z|x,\theta^{(t)}) \log\left(\frac{P(z|x,\theta^{(t)})}{P(z|x,\theta^{(t+1)})}\right) \geq 0$$

So we now have the equation

$$\log\Big(P(x|\theta^{(t+1)})\Big) - \log\Big(P(x|\theta^{(t)})\Big) = Q(\theta^{(t+1)}|\theta^{(t)}) - Q(\theta^{(t)}|\theta^{(t)}) + H(\theta^{(t)}|\theta^{(t+1)})$$

Which means that

$$\log\Big(P(x|\theta^{(t+1)})\Big) - \log\Big(P(x|\theta^{(t)})\Big) \geq Q(\theta^{(t+1)}|\theta^{(t)}) - Q(\theta^{(t)}|\theta^{(t)})$$

Therefore, optimizing the expression on the right will optimize our updated likelihood
Our choice of parameters is then

$$\theta^{(t+1)} = \arg\max_\theta Q(\theta|\theta^{(t)})$$

One version of the Expectation Maximization algorithm can then be expressed as:

E-step: Calculate the function:

$$Q(\theta|\theta^{(t)}) = \sum_z P(z|x,\theta^{(t)}) \log\left(P(x,z|\theta)\right)$$

M-step: Calculate the value of θ that maximizes Q:

$$\theta^{(t+1)} = \arg\max_\theta Q(\theta|\theta^{(t)})$$

As mentioned above, we saw that our parameters for the motif model are $\theta = (f, P_M)$, and the likelihood of a given subsequence x as being an instance of the model is given by:

$$P(x|\theta) = \prod_{i=1}^{\ell} \prod_{b=A}^{T} f_{b,i}^{x_{b,i}}$$

6.2.5 MEME: Motif finding with the EM Algorithm

As before, the motif finding problem is the task of given a set of sequences $S = \{s_1, ..., s_N\}$ to discover a motif of length K. In what follows we will present an algorithm for finding motifs without restriction on the number of occurrences per sequence.

We can assume that our sequences are from a finite alphabet A, which could in practice be the nucleotides or the amino acids, or a more general alphabet.

Let's construct a new dataset $x = (x_1, ..., x_n)$ of all possible overlapping subsequences of length K. In what follows we will present an unsupervised algorithm to identify which K-mer from the set x is an instance of a motif.

6.2.6 Finding a Single Motif with EM

For finding a single motif, we will need to compute a $K \times |\mathcal{A}|$ matrix such that its terms $f_{b,i}$ correspond to the probability of the letter $b \in A$ occurring at position i of an instance of motif. In addition, we will have a mixing parameter, or prior probabilities for being an instance of the motif. This variable will be given by $\lambda = (\lambda_0, \lambda_1)$, such that our motif has a mixing parameter λ_1, and the background has a parameter λ_0.

For convenience here, let's define the variable $\theta = \{\theta_0, \theta_1\}$ to correspond to this the probability matrices. However, for the motif, we'll have $\theta_1 = \{f_{b,i}\}$, and for the background model, we'll have a background model $\theta_0 = \{p_b\}$. For the missing data z, we can think of the data as a matrix of association for each sample x_i being an instance of motif or not. In other words,

$$z = (z_1, ..., z_n)$$

so that for each sample x_i in x, the value of z_i gives the degree of membership into the motif. The values for z_i are either0 and 1, corresponding to not being, and being an instance of the motif. Therefore, we have

$$z_i = \begin{cases} 1, & \text{if } x_i \text{ is an instance of the motif} \\ 0, & \text{if } otherwise \end{cases}$$

The likelihood of the data given the model parameters can be written as

$$P(x_i|z_i, \theta, \lambda) = P(x_i|\theta_0)^{1-z_i} P(x_i|\theta_1)^{z_i}$$

Where the exponent picks out the proper j for a given i, and similarly for the probability of z_i given the parameters:

$$P(z_i|\theta, \lambda) = \lambda_0^{1-z_i} \lambda_1^{z_i}$$

And the joint density can be written as the product of these two if we assume independence:

$$P(x_i, z_i|\theta, \lambda) = (P(x_i|\theta_0)\lambda_0)^{1-z_i} (P(x_i|\theta_1)\lambda_1)^{z_i}$$

We can then write the probability of all the data x and missing data z as the product over all samples i, again by assuming independence, giving us:

$$P(x, z|\theta, \lambda) = \prod_{i=1}^{n} (P(x_i|\theta_0)\lambda_0)^{1-z_i} (P(x_i|\theta_1)\lambda_1)^{z_i}$$

It is useful to work with the log-likelihood in this case, which is

$$\log P(x, z|\theta, \lambda) = \sum_{i=1}^{n} (1 - z_i) \log (P(x_i|\theta_0)\lambda_0) + z_i \log (P(x_i|\theta_1)\lambda_1)$$

As previously discussed, the expectation maximization algorithm will construct a quantity $Q(\theta^{(t+1)}|\theta^{(t)})$, which is the expected value of the log-likelihood in terms of the updated parameters $\theta^{(t+1)}$ weighted by probabilities in terms of the current parameters $\theta^{(t)}$.

$$\begin{aligned} Q(\theta^{(t+1)}|\theta^{(t)}) &= \sum_z P(z|x, \theta^{(t)}, \lambda^{(t)}) \log \left(P(x, z|\theta^{(t+1)}, \lambda^{(t+1)}) \right) \\ &= \sum_{i=1}^{n} P(z_i = 1|x_i, \theta_1^{(t)}, \lambda_1^{(t)}) \log \left(P(x_i|\theta_1^{(t+1)}) \lambda_1^{(t+1)} \right) \\ &\quad + P(z_i = 0|x_i, \theta_0^{(t)}, \lambda_0^{(t)}) \log \left(P(x_i|\theta_0) \lambda_0^{(t+1)} \right) \end{aligned} \tag{6.5}$$

The key term that we need to compute in the $Q(\theta^{(t+1)}|\theta^{(t)})$ terms is the probabilities of the z_i in terms of the current parameters $\theta^{(t)}$. However, it is clear that these are essentially the posterior probabilities, when we use Bayes' rule:

$$P(z_i = 1|x_i, \theta^{(t)}, \lambda^{(t)}) = \frac{P(x_i|\theta_1^{(t)})\lambda_1^{(t)}}{P(x_i|\theta_1^{(t)})\lambda_1^{(t)} + P(x_i|\theta_0)(\lambda_0^{(t)})}$$

Let's define this term as $z_i^{(t)}$ since it is the expected value of z_i with the current parameters $\theta_j^{(t)}$. That is,

$$\begin{aligned} z_i^{(t)} =& 1 \times P(z_i = 1|x_i, \theta_1^{(t)}, \lambda_1^{(t)}) + 0 \times P(z_i = 0|x_i, \theta_0^{(t)}, \lambda_0^{(t)}) \\ =& P(z_i = 1|x_i, \theta^{(t)}, \lambda^{(t)}) \end{aligned} \tag{6.6}$$

Plugging these values back into our equation for $Q(\theta^{(t+1)}|\theta^{(t)})$ gives us:

$$Q(\theta^{(t+1)}|\theta^{(t)}) = \sum_{i=1}^{n} z_i^{(t)} \log\left(P(x_i|\theta_1^{(t+1)})\lambda_1^{(t+1)}\right) + (1 - z_i^{(t)}) \log\left(P(x_i|\theta_0)\lambda_0^{(t+1)}\right)$$

To compute our updated parameters, we choose those that maximize $Q(\theta^{(t+1)}|\theta^{(t)})$. First let compute the updated values for $\lambda^{(t+1)}$ by using a Lagrangian with the constraint equation that $\sum_j \lambda_j - 1 = 0$

$$\mathcal{L} = Q(\theta^{(t+1)}|\theta^{(t)}) + \alpha\left(\sum_j \lambda_j - 1\right)$$

Then differentiating and setting equal to zero gives us

$$\frac{\partial \mathcal{L}}{\partial \lambda_j^{(t+1)}} = 0$$

implies that

$$\lambda_j^{(t+1)} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)}}{n}$$

Now for computing optimal values of $\theta^{(t+1)}$. To do this we need to write the expression for $P(x_i|\theta_1)$. This was given in the previous lecture as:

$$P(x_i|\theta_1) = \prod_{k=1}^{K} \prod_{b \in A} f_{kb}^{x_{ikb}}$$

Where x_{ikb} is the indicator matrix defined for the $K-$mer x_i such that

$$x_{ikb} = \begin{cases} 1, & \text{if } x_i[k] = b \\ 0, & \text{if } x_i[k] \neq b \end{cases}$$

Similarly, we can express $P(x_i|R)$ as a product of nucleotide frequencies:

$$P(x_i|\theta_0) = \prod_{k=1}^{K} \prod_{b \in A} p_b^{x_{ikb}}$$

So we can therefore compute the optimal updated parameters for each f_{jkb} term of the updated parameters $\theta^{(t+1)}$. Our Lagrangian will be defined analogously as before, but with the constraint equation $\sum_b f_{jkb} - 1 = 0$ for all $j = 1, ..., g$ and $k = 1, ..., K$. In doing so,

$$\frac{\partial \mathcal{L}}{\partial f_{kb}^{(t+1)}} = 0$$

Implies that

$$f_{kb}^{(t+1)} = \frac{\sum_i z_i^{(t)} x_{ikb}}{\sum_i \sum_{b \in A} z_i^{(t)} x_{ikb}}$$

Which gives the probability matrix terms as the counts contributed from each $K-$mer x_i weighted by its degree of membership z_{ij} into the motif j.

In summary, we can simplify our procedure by iterating the following steps for a set number of iterations, or until convergence:

1. Compute $z_i^{(t)}$ in terms of $\lambda^{(t)}$ and $f_{kb}^{(t)}$

$$z_i^{(t)} = \frac{P(x_i|\theta_1^{(t)})\lambda_1^{(t)}}{P(x_i|\theta_1^{(t)})\lambda_1^{(t)} + P(x_i|\theta_0)\lambda_0^{(t)}}$$

2. Compute the updates for $\lambda^{(t+1)}$ and $f_{kb}^{(t+1)}$

$$\lambda^{(t+1)} = \sum_{i=1}^{n} \frac{z_i^{(t)}}{n}$$

and

$$f_{kb}^{(t+1)} = \frac{\sum_i z_i^{(t)} x_{ikb}}{\sum_i \sum_{b \in A} z_i^{(t)} x_{ikb}}$$

Continue this process until convergence of the motif, or after some number of iterations.

6.2.7 Finding Multiple Motifs with EM

For the multiple motif case, we will be finding g different motifs in the data. We will consider g motifs/groups of the subsequences, so our parameters will be $\theta = (\theta_1, ..., \theta_g)$. For each of the g motifs, we will estimate a $K \times |\mathcal{A}|$ matrix such that its terms f_{jib} correspond to the probability of the letter $b \in A$ occurring at position i of an instance of motif j.

In addition, we will have a set of mixing parameters, or prior probabilities for being an instance of one of the g motifs $\lambda = (\lambda_1, ..., \lambda_g)$ where $\sum_j \lambda_j = 1$.

For convenience, let us define the variable $\theta_j = (\vec{f_{j1}}, ..., \vec{f_{jK}}, \lambda_j)$ for each motif such that $\vec{f_{ji}}$ is the frequency vector of probabilities for each position i of the motif.

For the missing data z, we can think of the data as a matrix of association for each sample x_i being an instance of motif j. In that sense,

$$z = (z_1, ..., z_n)$$

for each of the samples in x, and then

$$z_i = (z_{i1}, ..., z_{ig})$$

for the membership into one of the motifs/groups g. Therefore, we have

$$z_{ij} = \begin{cases} 1, & \text{if } x_i \text{ is in group } j \\ 0, & \text{if } otherwise \end{cases}$$

The likelihood of the data given the model parameters can be written as

$$P(x_i|z_i, \theta, \lambda) = \prod_{j=1}^{g} p(x_i|\theta_j)^{z_{ij}}$$

Where the exponent picks out the proper j for a given i, and similarly for the probability of z_i given the parameters:

$$P(z_i|\theta,\lambda) = \prod_{j=1}^{g} \lambda_j^{z_{ij}}$$

And the joint density can be written as the product of these two if we assume independence:

$$P(x_i, z_i|\theta,\lambda) = \prod_{j=1}^{g} [P(x_i|\theta_j)\lambda_j]^{z_{ij}}$$

We can then write the probability of all the data x and missing data z as the product over all samples i, again by assuming independence, giving us:

$$P(x, z|\theta,\lambda) = \prod_{i=1}^{n}\prod_{j=1}^{g} [P(x_i|\theta_j)\lambda_j]^{z_{ij}}$$

It is useful to work with the log-likelihood in this case, which is

$$\log P(x, z|\theta,\lambda) = \sum_{i=1}^{n}\sum_{j=1}^{g} z_{ij} \log\left(P(x_i|\theta_j)\lambda_j\right)$$

As discussed in Lecture 11, we showed that the expectation maximization algorithm will construct a quantity $Q(\theta^{(t+1)}|\theta^{(t)})$, which is the expected value of the log-likelihood in terms of the updated parameters $\theta^{(t+1)}$ weighted by probabilities in terms of the current parameters $\theta^{(t)}$.

$$\begin{aligned} Q(\theta^{(t+1)}|\theta^{(t)}) &= \sum_{z} P(z|x,\theta^{(t)},\lambda^{(t)}) \log\left(P(x,z|\theta^{(t+1)},\lambda^{(t+1)})\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{g} P(z_{ij}|x_i,\theta_j^{(t)},\lambda_j^{(t)}) z_{ij} \log\left(P(x_i|\theta_j^{(t+1)})\lambda_j^{(t+1)}\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{g} P(z_{ij}=1|x_i,\theta_j^{(t)},\lambda_j^{(t)}) \log\left(P(x_i|\theta_j^{(t+1)})\lambda_j^{(t+1)}\right) \end{aligned} \tag{6.7}$$

The key term that we need to compute in the $Q(\theta^{(t+1)}|\theta^{(t)})$ terms is the probabilities of the z_{ij} in terms of the current parameters $\theta^{(t)}$. However, it is clear that these are essentially the posterior probabilities, when we use Bayes' rule:

$$P(z_{ij}=1|x_i,\theta_j^{(t)},\lambda_j^{(t)}) = \frac{P(x_i|\theta_j^{(t)})\lambda_j^{(t)}}{\sum_{k=1}^{g} P(x_i|\theta_k^{(t)})\lambda_k^{(t)}}$$

Let's define this term as $z_i^{(t)}$ since it is the expected value of z_i with the current parameters $\theta_j^{(t)}$. That is,

$$z_i^{(t)} = 1 \times P(z_i = 1|x_i,\theta^{(t)},\lambda^{(t)}) + 0 \times P(z_{ij}=0|x_i,\theta_j^{(t)},\lambda_j^{(t)}) = P(z_{ij}=1|x_i,\theta_j^{(t)},\lambda_j^{(t)})$$

Plugging these values back into our equation for $Q(\theta^{(t+1)}|\theta^{(t)})$ gives us:

$$Q(\theta^{(t+1)}|\theta^{(t)}) = \sum_{i=1}^{n}\sum_{j=1}^{g} z_{ij}^{(t)} \log\left(P(x_i|\theta_j^{(t+1)})\lambda_j^{(t+1)}\right)$$

To compute our updated parameters, we choose those that maximize $Q(\theta^{(t+1)}|\theta^{(t)})$. First let compute the updated values for $\lambda^{(t+1)}$ by using a Lagrangian with the constraint equation that $\sum_j \lambda_j - 1 = 0$

$$\mathcal{L} = Q(\theta^{(t+1)}|\theta^{(t)}) + \alpha\left(\sum_j \lambda_j - 1\right)$$

Then differentiating and setting equal to zero gives us

$$\frac{\partial \mathcal{L}}{\partial \lambda_j^{(t+1)}} = 0$$

implies that

$$\lambda_j^{(t+1)} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)}}{n}$$

Now for computing optimal values of $\theta_j^{(t+1)}$. To do this we need to write the expression for $P(x_i|\theta_j)$. This was given in the previous lecture as:

$$P(x_i|\theta_j) = \prod_{k=1}^{K} \prod_{b \in A} f_{jkb}^{x_{ikb}}$$

So we can therefore compute the optimal updated parameters for each f_{jkb} term of the updated parameters $\theta^{(t+1)}$. Our Lagrangian will be defined analogously as before, but with the constraint equation $\sum_b f_{jkb} - 1 = 0$ for all $j = 1, ..., g$ and $k = 1, ..., K$. In doing so,

$$\frac{\partial \mathcal{L}}{\partial f_{jkb}^{(t+1)}} = 0$$

Implies that

$$f_{jkb}^{(t+1)} = \frac{\sum_i z_{ij}^{(t)} x_{ikb}}{\sum_i \sum_{b \in A} z_{ij}^{(t)} x_{ikb}}$$

Which gives the probability matrix terms as the counts contributed from each $K-$mer x_i weighted by its degree of membership z_{ij} into the motif j.

In summary, we can simplify our procedure by iterating the following steps for a set number of iterations, or until convergence:

1. Compute $z_{ij}^{(t)}$ in terms of $\lambda_j^{(t)}$ and $f_{jkb}^{(t)}$

$$z_{ij}^{(t)} = \frac{P(x_i|\theta_j^{(t)})\lambda_j^{(t)}}{\sum_{k=1}^{g} P(x_i|\theta_k^{(t)})\lambda_k^{(t)}}$$

2. Compute the updates for $\lambda_j^{(t+1)}$ and $f_{jkb}^{(t+1)}$

$$\lambda_j^{(t+1)} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)}}{n}$$

and

$$f_{jkb}^{(t+1)} = \frac{\sum_i z_{ij}^{(t)} x_{ikb}}{\sum_i \sum_{b \in A} z_{ij}^{(t)} x_{ikb}}$$

Continue this process until convergence of the motif, or after some number of iterations.

Let's explicitly compute the optimal parameter choices. To compute our updated parameters, we choose those that maximize $Q(\theta^{(t+1)}|\theta^{(t)})$. First let compute the updated values for $\lambda^{(t+1)}$ by using a Lagrangian with the constraint equation that $\sum_j \lambda_j - 1 = 0$

$$\mathcal{L} = Q(\theta^{(t+1)}|\theta^{(t)}) + \alpha\left(\sum_j \lambda_j - 1\right)$$

Expanding this expression gives us:

$$\mathcal{L} = \sum_{i=1}^{n}\sum_{j'=1}^{g} z_{ij'}^{(t)} \log\left(P(x_i|\theta_{j'}^{(t+1)})\lambda_{j'}^{(t+1)}\right) + \alpha\left(\sum_{j'} \lambda_{j'} - 1\right)$$

The expression in the first sum can be broken up into to terms in the log: one that depends on $\lambda_j^{(t+1)}$ and one that doesn't:

$$\mathcal{L} = \sum_{i=1}^{n}\sum_{j'=1}^{g} z_{ij'}^{(t)} \log\left(P(x_i|\theta_{j'}^{(t+1)})\right) + \sum_{i=1}^{n}\sum_{j'=1}^{g} z_{ij'}^{(t)} \log\left(\lambda_{j'}^{(t+1)}\right) + \alpha\left(\sum_{j'} \lambda_{j'} - 1\right)$$

Our derivative with respect to $\lambda_j^{(t+1)}$ will only include the second and last sum of the Lagrangian. Differentiating with respect to $\lambda_j^{(t+1)}$ gives us:

$$\frac{\partial \mathcal{L}}{\partial \lambda_j^{(t+1)}} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)}}{\lambda_j^{(t+1)}} + \alpha$$

Differentiating with respect to α gives us

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \sum_j \lambda_j - 1$$

Setting these two equations equal to zero, and solving for $\lambda_j^{(t+1)}$ gives us:

$$\lambda_j^{(t+1)} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)}}{n}$$

Now for computing optimal values of $\theta_j^{(t+1)}$. To do this we need to write the expression for $P(x_i|\theta_j)$. This was given in the previous lecture as:

$$P(x_i|\theta_j) = \prod_{k=1}^{K}\prod_{b\in A} (f_{jkb}^{(t+1)})^{x_{ikb}}$$

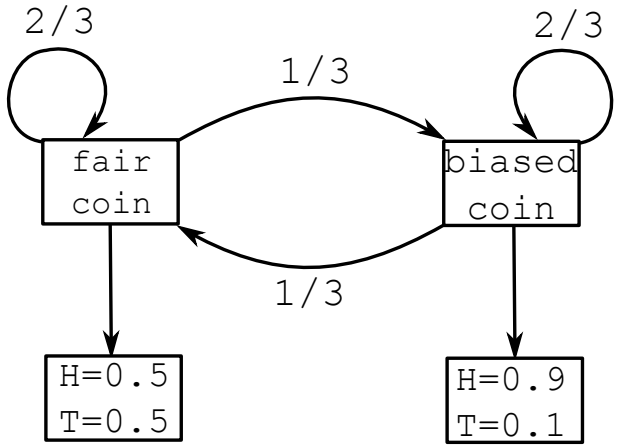
Where x_{ikb} is the indicator matrix defined previously.

Our Lagrangian will be defined analogously as before, but with the constraint equation $\sum_b f_{jkb}^{(t+1)} - 1 = 0$ for all $j = 1, ..., g$ and $k = 1, ..., K$. We will now compute the optimal updated parameters for each $f_{jkb}^{(t+1)}$ term of the updated parameters $\theta^{(t+1)}$.

Writing out the Lagrangian in terms of our expression for $P(x_i|\theta_j)$ from the previous slide gives us:

$$\mathcal{L} = \sum_{i=1}^{n}\sum_{j'=1}^{g}\sum_{k'=1}^{K}\sum_{b'} z_{ij'}^{(t)} x_{ik'b'} \log\left(f_{j'k'b'}^{(t+1)}\right) + \sum_{i=1}^{n}\sum_{j'=1}^{g} z_{ij'}^{(t)} \log\left(\lambda_{j'}^{(t+1)}\right) + \alpha\left(\sum_{b'} f_{j'k'b'} - 1\right)$$

Differentiating with respect to $f_{jkb}^{(t+1)}$ gives us:



$$\frac{\partial \mathcal{L}}{\partial f_{jkb}^{(t+1)}} = \sum_{i=1}^{n} \frac{z_{ij}^{(t)} x_{ikb}}{f_{jkb}^{(t+1)}} + \alpha$$

Differentiating with respect to α gives us:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \sum_{b} f_{jkb}^{(t+1)} - 1$$

$$f_{jkb}^{(t+1)} = \frac{\sum_i z_{ij}^{(t)} x_{ikb}}{\sum_i \sum_{b \in A} z_{ij}^{(t)} x_{ikb}}$$

Which gives the probability matrix terms as the counts contributed from each $K-$mer x_i weighted by its degree of membership z_{ij} into the motif j.

6.3 Hidden Markov Models

6.3.1 Markov Models

Example: The opening coin toss Say there is a crooked referee for college football that is alleged to carry a biased coin, such that it comes up heads 90% of the time. To make matters more difficult, he only uses it some of the time. Let's further assume that when he uses it, he is twice as likely to use it for the next game, and likewise if he uses his fair coin, he is twice as likely to use the fair coin for the next game.

In such a situation, we have a hidden variable, a state variable, which describes if he is using his fair coin or his biased coin. Given we know what coin is being used, we know the expected probability of coming up heads.

The hidden variable here is the coin that is being used, because we can't see which coin was used in which game. We can only observe the sequence of heads and tails X, but we can't see the sequence of state variables π:

$$X = \texttt{HHTTHHTHHTHTHHHHHHHHHTHTTTHTHHTHTT}$$

$$\pi = \texttt{00000001100111111111110000000000}$$

So for a given coin toss i, we have the observed coin toss X_i, and the unobserved state variable π_i.

Let's call the state for the fair coin 0, and the state of being the biased coin 1. As stated above, we have a set of transition probabilities to go between our states $\{0, 1\}$, such that $\beta_{00} = \beta_{11} = \frac{2}{3}$ and $\beta_{01} = \beta_{10} = \frac{1}{3}$.

In addition, we have "emission probabilities" $e_k(x)$ such that

$$e_k(x) = P(X_i = x|\pi_i = k)$$

Essentially, giving the us the probability of the coin toss being heads or tails for a given coin state. So in our example, $e_1(H) = 0.9$.

So assuming we know the state sequence of hidden variables $\pi = \pi_1, \pi_2, ..., \pi_N$, then we can compute the probability of the observed and hidden variables as

$$P(x, \pi) = e_{\pi_N}(X_N) \prod_{i=1}^{N-1} e_{\pi_i}(X_i)\beta_{\pi_i,\pi_{i+1}}$$

The key concept of the HMM is it gives us a model with which to describe hidden, unobserved variables. It also gives us a formalize with which to compute a prediction of what the hidden states are.

6.3.2 The Viterbi Algorithm

The approach of the Viterbi Algorithm is to choose the state path π that maximizes the probability of the observed and hidden variables:

$$\pi^* = \arg\max_{\pi} P(x, \pi)$$

The Viterbi Algorithm solves this problem recursively. It starts by defining the probability $p_k(i)$ which is the probability of the most probable state path ending in state k, such that $\pi_i = k$, and which depends on the data up to i, that is $x_1, ..., x_i$. We can compute the probability of $p_k(i+1)$ by the following recursion relation:

$$p_k(i + 1) = e_k(x_{i+1}) \max_{k'}\{p_{k'}(i)\beta_{k',k}\}$$

In addition, similar to the traceback matrix used in smith-waterman and other dynamic programming treatments of sequence alignment, we define a traceback term $T_i(k)$. This term holds the optimal state from which we could come from to get to k at sequence position i when coming from sequence position $i - 1$.

---

function VITERBI ALGORITHM(x)
  $p_0(0) = 1, p_k(0) = 0$ for $k > 1$ ▷ Initialization
  for $i = 0, ..., N$ **do**
    $p_k(i) = e_k(x_{i+1}) \max_{k'}\{p_{k'}(i)\beta_{k',k}\}$ ▷ Update
    $T_i(k) = \arg\max_{k'}\{p_{k'}(i-1)\beta k', k\}$
  $P(x, \pi) = \max_k\{p_k(N)\beta_{k,0}\}$ ▷ Final values
  $\pi^*_N = \arg\max_k\{p_k(N)\beta_{k,0}\}$
  for $i = L, ..., 1$ **do**
    $\pi^*_{i-1} = T_i(\pi^*_i)$
  return π^*

The Viterbi algorithm computes the optimal path π^* using the data. Another approach would be to compute the probability $P(x)$ in terms of all possible paths. That is, consider expanding $P(x)$ into all possible paths:

$$P(x) = \sum_{\pi} P(x, \pi) = \sum_{\pi} P(x|\pi)P(\pi)$$

The Forward Algorithm and Backward Algorithm employ such a strategy

6.3.3 The Forward Algorithm

The Forward Algorithm defines a term $f_k(i)$ that corresponds to the probability of the observed sequence up to position i and ending at state k, analogously to what we defined as $p_k(i)$ for the Viterbi algorithm. The recursion relation for $f_k(i)$ is given by:

$$f_k(i+1) = e_k(x_{i+1}) \sum_{k'} f_{k'}(i)\beta_{k',k}$$

Note: this algorithm doesn't give us the sequence π, but gives a result that essentially takes into account all possible sequences of hidden states.

---

function FORWARD ALGORITHM(x)
  $f_0(0) = 1, f_k(0) = 0$ for $k > 1$ ▷ Initialization
  for $i = 0, ..., N$ **do**
    $f_k(i) = e_k(x_i) \sum_{k'} f_{k'}(i-1)\beta_{k',k}$ ▷ Update
  $P(x) = \sum_k f_k(N)\beta_{k,0}$ ▷ Final probability sum ending at state 0
return $P(x)$

6.3.4 The Backward Algorithm

Now we turn to the Backward Algorithm. The intent of the backward algorithm is to produce posterior probabilities $P(\pi_i = k|x)$ such that we know that position i in the sequence, and corresponding value x_i came from the hidden state $\pi_i = k$. We can compute this using Bayes' Law

$$P(\pi_i = k|x) = \frac{P(x, \pi_i = k)}{P(x)}$$

The Forward algorithm gives us $P(x)$, so we just need to compute the numerator, $P(x, \pi_i = k)$.
We can compute $P(x, \pi_i = k)$ by recognizing that it can be expressed as a product of two terms:

$$\begin{aligned} P(x, \pi_i = k) &= P(x_1, ..., x_i, \pi_i = k)P(x_{i+1}, ..., x_N|x_1, ..., x_i, \pi_i = k) \\ &= P(x_1, ..., x_i, \pi_i = k)P(x_{i+1}, ..., x_N|\pi_i = k) \\ &= f_k(i)b_k(i) \end{aligned} \tag{6.8}$$

where

$$b_k(i) = P(x_{i+1}, ..., x_N|\pi_i = k)$$

---

function BACKWARD ALGORITHM(x)
  $b_k(N) = \beta_{k,0}$ for all k ▷ Initialization
  for $i = N-1, ..., 1$ **do**
    $b_k(i) = \sum_{k'} e_{k'}(x_{i+1})b_{k'}(i+1)\beta_{k,k'}$ ▷ Update
  $P(x) = \sum_k e_k(x_1)b_k(1)\beta_{0,k}$
return $P(x)$

Finally, we can compute our desired probabilities using the terms computed from the forward and backward algorithm.

$$P(\pi_i = k|x) = \frac{f_k(i)b_k(i)}{P(x)}$$

Chapter 7

RNA folding

7.1 RNA folding: secondary structure

We begin with the Nussinov Algorithm, which effectively counts base pairs. The optimal structure is that which contains the most base pairs.

Let's consider an RNA sequence r such that $r[i] \in \{A, C, G, U\}$. The length of r given by $|r|$ could be on the order of hundreds or thousands.

We can define a structure S of r to be such that S is a set of pairs of nucleotides of r called base-pairs. That is, $(r[i], r[j]) \in S$, where $1 \leq i < j \leq |r|$.

We simply require that $r[i]$ and $r[j]$ be complementary nucleotides, and that they are further than some minimum separation distance d_{min} by requiring that $j - i > d_{min}$.

We want to exclude pseudoknots, which we will see enables us to formulate a much simpler recursion relation for determining structure. A pseudoknot can be defined as a set of pairs $(r[i], r[j]) \in S$ and $(r[k], r[l]) \in S$ such that $i < k < j < l$.

By excluding pseudoknots, we can also represent the structure as a planar graph. A pseudoknot would correspond to edges crossing in a planar representation.

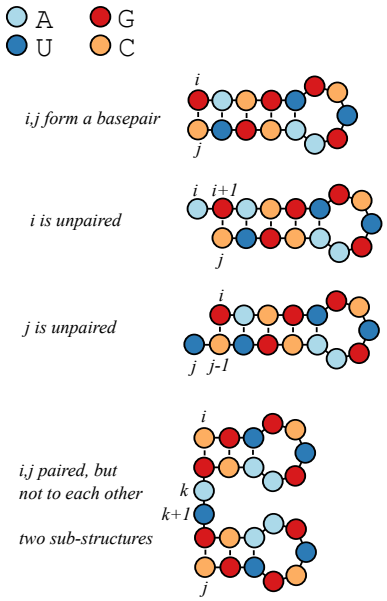
7.2 Nussinov Algorithm

1. i and j are a base pair, add to the structure for $i + 1$ to $j - 1$
2. i is unpaired, but j is paired, add to the structure for $i + 1$ to j
3. j is unpaired, add to the structure for i to $j - 1$
4. bifurcation: i and j are paired, but not to each other.

The recursion relation for the Nussinov algorithm can be stated as:

$$B_{i,j} = max \begin{cases} B_{i+1,j-1} + 1 & r[i] \text{ and } r[j] \text{ form a base pair} \\ B_{i+1,j} & r[i] \text{ is unpaired, but } r[j] \text{ is paired.} \\ B_{i,j-1} & r[i] \text{ is paired, but } r[j] \text{ is unpaired} \\ max_{i<k<j} B_{i,k} + B_{k+1,j} & \text{bifurcation: two sub-structures} \end{cases} \tag{7.1}$$

Input: S, i, j
Output: Number of base pairs in RNA structure.
Initial call: $traceback(S, 1, |r|)$



---

function TRACEBACK(S, i, j) ▷ Traceback for Nussinov Algorithm
  if $i < j$ **then**
    if $S(i,j) = S(i+1,j)$ **then**
      traceback(S,i+1,j)
    else if $S(i,j) = S(i,j-1)$ **then**
      traceback(S,i,j-1)
    else if $S(i,j) = S(i+1,j-1)+1$ **then**
      print(base pair at (i,j))
      traceback(i+1,j-1)
    else
      for $k \leftarrow i+1..j-1$ **do**
        if S(i,j) = S(i,k) + S(k+1,j) **then**
          traceback(S,i,k)
          traceback(S,k+1,j)

7.3 Zucker Algorithm

The problem with the Nussinov Algorithm in this form is that the predicted structure only counts basepairs. It does not count the destabilizing energy contributions from loops, internal loops, and bulges.

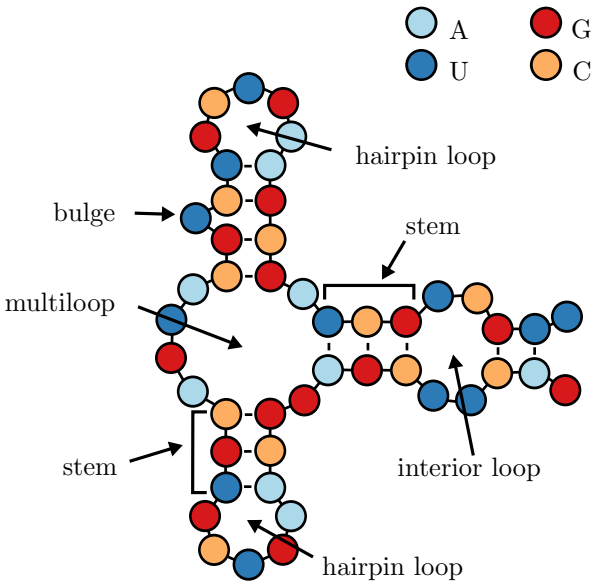
Let's assume that each base pair contributes an amount of energy to the total structure's free energy. The total energy of the structure is then given by:

$$E(S) = \sum_{(r[i],r[j]) \in S} \Delta G(r[i], r[j])$$

Similarly, we can consider the sub-structure $S_{i,j}$ corresponding to the subsequence $r[i..j]$. The exclusion of pseudoknots makes it possible that $S_{i,j} \subseteq S$

We impose that $\Delta G(r[i], r[i]) = 0$, and it's negative (stabilizing energy) for all base pairs.

Consider adding $r[j]$ to the sequence $r[i..j-1]$. If it doesn't contribute a base pair, then the energy doesn't change, that is $E(S_{i,j}) = E(S_{i,j-1})$.



If $r[i]$ is paired with $r[j]$, then we just add it's energy contribution: $E(S_{i,j}) = E(S_{i+1,j-1}) + \Delta G(r[i], r[j])$.

Alternatively, if $r[j]$ is paired with some other nucleotide $r[k]$ such that $i < k \leq j$, then we can say that the total energy is the sum of $E(S_{k,j})$ and the energy o the other portion of the sequence $E(S_{i,k-1})$

We can combine these with the recurrence relation:

$$E(S_{i,j}) = min \begin{cases} E(S_{i+1,j-1}) + \Delta G(r[i], r[j]) & i, j \text{ paired} \\ \min_{i<k\leq j}(E(S_{i,k}) + E(S_{k+1,j})) & j \text{ paired with} k \end{cases} \tag{7.2}$$

Let us now consider destabilizing energy contributed by loops and unpaired bases.

The energy $E(S_{i+1,j-1})$ may be weak and positive due to being unpaired, but the contribution of $(r[i], r[j])$ may end up stabilizing the energy $E(S_{i,j})$. For example, the $(r[i], r[j])$ base pair may balance out the destabilizing energy of a loop in between those positions.

In general, to consider destabilizing energy, we have the following recurrence relation:

$$E(S_{i,j}) = min \begin{cases} E(S_{i+1,j}) & i \text{ is unpaired} \\ E(S_{i,j-1}) & j \text{ is unpaired} \\ \min_{i<k<j}(E(S_{i,k}) + E(S_{k+1,j})) & i, j \text{ paired, but not to each other} \\ E(L_{i,j}) & i, j \text{paired to each other} \end{cases} \tag{7.3}$$

The last expression $E(L_{i,j})$ means that we have to compute all the destabilizing effects in the loop structures $L_{i,j}$ corresponding to the sequence from i to j.

We can consider the following types of energy contributions:

- $\alpha(k)$ = the destabilizing energy of a hairpin loop of length k.
- $\beta(k)$ = the destabilizing energy of a bulge of length k.
- $\gamma(k)$ = the destabilizing energy of an interior loop of length k.
- ϵ = the stabilizing stacking energy of neighboring base pairs.

We can consider the following types of energy contributions:

$$E(L_{i,j}) = \begin{cases} \Delta G(r[i], r[j]) + \alpha(j - i - 1) & \text{if } L_{i,j} \text{ is a hairpin loop} \\ \Delta G(r[i], r[j]) + \epsilon + E(S_{i+1,j-1}) & j \text{ if } L_{i,j} \text{ is a helical region} \\ \min_{k \geq 1}(\Delta G(r[i], r[j]) + \beta(k) + E(S_{i+k+1,j-1})) & \text{if } L_{i,j} \text{ is a bulge on } i \\ \min_{k \geq 1}(\Delta G(r[i], r[j]) + \beta(k) + E(S_{i+1,j-k-1})) & \text{if } L_{i,j} \text{ is a bulge on } j \\ \min_{k_1,k_2 \geq 1}(\Delta G(r[i], r[j]) + \gamma(k_1 + k_2) + E(S_{i+1+k_1,j-i-k_2})) & \text{if } L_{i,j} \text{ is an interior loop} \end{cases} \tag{7.4}$$

Bibliography

[1] F.H.C. Crick. On protein synthesis. *Symp. Soc. Exp. Biol*, XII:139–163, 1956. `http://profiles.nlm.nih.gov/SC/B/B/F/T/_/scbbft.pdf`.

[2] F. Crick. Central dogma of molecular biology. *Nature*, 227(5258):561–563, Aug 1970.

[3] Dayhoff M.O., R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(3):345–352, 1978.

[4] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, May 1994.

[5] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.