Biological Computing (CS515) Class Notes

Robert B. Heckendorn
Initiative for Bioinformatics and Evolutionary STudies (IBEST)
University of Idaho
Moscow, Idaho

April 19, 2015

Contents

Introduction

These are some of the notes for Biological Computation class. Missing are various hand written notes and slides. These notes are provided as a study and lecture aid. As all material produced in the act of teaching at the University of Idaho, this material is copyrighted by the instructor (Robert Heckendorn) and the University of Idaho who retains all intellectual rights of the material for future publication or classes. I apologize for the lack of completeness, spelling errors, and casual manner of these notes. I will correct this over time, but after all, they are just my class notes ☺.

The material in these notes covers chapters from at least these three books and numerous articles. The three books, all three of which I highly recommend, are:

- *Biological Sequence Analysis* by Durbin, Eddy, Krogh, and Mitchison, Cambridge Press 2001
- *Inferring Phylogenies* by Joseph Felsenstein, Sinauer 2004
- *Introduction to Computational Biology* by Michael S. Waterman, Chapman and Hall/CRC 2000

Chapter 1

Introductory Math

1.1 Basic Probability

1.1.1 Mutually Exclusive Events

Consider mutually exclusive events A, B, C. The universe of all discrete events that could be observed is U. Let $P(A)$ is the probability of seeing A or observing an event of type A. Let $P(B)$ is the probability of seeing B or observing an event of type B. Any probability P is non-negative. In fact:

$$P(i) \in [0,1] \forall\ i \in U$$

The probability of seeing either A or B that are known to be mutually exclusive events is

$$P(A \text{ or } B) = P(A) + P(B)$$

If the events A or B are not necessarily **mutually exclusive** then

$$P(A \text{ or } B) = P(A) + P(B) - P(A, B)$$

Where $P(A, B)$ denotes the **joint probability**, that is the probability of both A **and** B occurring. Another way to think of this is the probability of something being both of type A and type B. Note that by definition:

$$P(A, B) = P(B, A)$$

For example:

If $P(A)$ is probability of a die roll being 1 and $P(B)$ is the probability of a die roll being 3 then

$$P(A) + P(B) - P(A, B) = \frac{1}{6} + \frac{1}{6} - 0 = \frac{2}{6}$$

If $P(A)$ is probability of a die roll being less than 4 and $P(B)$ is the probability of a die roll being even, then

$$P(A) + P(B) - P(A, B) = \frac{1}{2} + \frac{1}{2} - \frac{1}{6} = \frac{5}{6}$$

The sum of the probabilities of all events in U, if they are mutually exclusive, is 1.

$$\sum_{i \in U} P(i) = 1$$

1.1.2 Independent Events

Consider events A, B where A and B are independent of each other. They are **independent** if whether A has occurred or not does not change the probability of B occurring. The probability that A on one observation and B on the next is denoted $P(AB)$:

$$P(AB) = P(A)P(B) \qquad \textbf{provided } A \textbf{ and } B \textbf{ are independent!}$$

1.1.3 Bayesian Probabilities

Let $P(A, B)$ denote the **joint probability**, that is the probability of both A and B occurring also denoted $(A \cap B)$.

NOTE: $P(A, B) \neq P(AB)$. The first is the probability of the event being classified as both of type A and type B. For our purposes, $P(AB)$ is the probability of B following A as in an ordered sequence.

Let $P(A|B)$ denote the **conditional probability**, that is the probability of A occurring given B has occurred. This is defined as:

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

Since

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

we see that

$$P(A|B)P(B) = P(A, B) = P(B|A)P(A)$$

therefore

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

Bayesian probabilities (and statistics sort of) can be viewed as solving unknowns in a 2×2 matrix. Consider this example matrix of people with red or black hair and male or female sex. Row and column sums are provided for convenience.

	red hair	black hair	row sum
male	10	31	41
female	36	23	59
col sum	46	54	100

$P(\text{red}) = 46/100$

$P(\text{red}, M) = 10/100$

$P(\text{red}|M) = 10/41$

$P(M|\text{red}) = 10/46$

Let's compute $P(\text{red}, M)$ and $P(M, \text{red})$ (which is really the same thing).

$$\begin{array}{lll} P(\text{red}, M) & = P(\text{red}|M) & P(M) \\ 10/100 & = 10/41 & 41/100 \end{array}$$

$$\begin{array}{lll} P(M, \text{red}) & = P(M|\text{red}) & P(\text{red}) \\ 10/100 & = 10/46 & 46/100 \end{array}$$

therefore $P(\text{red}, M) = P(M, \text{red}) = 10/100$.

Example Problem: Now suppose I didn't give you the matrix above but gave you these three facts:

1. $P(\text{red}|M) = 10/41$
2. $P(M) = 41/100$
3. $P(\text{red}) = 46/100$

What is $P(M|\text{red})$?

Answer:

$$P(\text{red}|M)P(M)/P(\text{red}) = (10/41)(41/100)/(46/100) = (10 * 41 * 100)/(41 * 100 * 46) = 10/46$$

BINGO! ... just as in the table.

The above table converted to probabilities looks like:

	red hair	black hair	row sum
male	P(M, red)	P(M, black)	P(M)
female	P(F, red)	P(F, black)	P(F)
col sum	P(red)	P(black)	1.0

We know that the $P(M, red) = P(M)P(red)$ if M and red are independent. Clearly that isn't true for our example data. In particular the values inside the matrix are not normally able to be computed from the row and column sum information unless an assumption of independence in row and column is made. This assumption is frequently made in bioinformatics to make the problem solvable.

Here are the general rules we have created in playing with the example matrix:

$$P(A,B) = P(B,A) = P(B|A)P(A) = P(A|B)P(B)$$

or

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

This is called **Bayes' Theorem** after Reverend Thomas Bayes (1702-1761).

The major point to see here is that if we make some assumptions about $P(A)$ and $P(B)$ or even just their ratio, we can convert $P(B|A)$ into $P(A|B)$ This will allow us to convert between $P(\text{data}|\text{model})$ and $P(\text{model}|\text{data})$ if we know the $P(\text{data})$ and $P(\text{model})$. In the following:

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

$P(A)$ and $P(B)$ are **prior probabilities**
$P(B|A)$ is the **sampling probability** or **likelihood**
$P(A|B)$ is the **posterior probability**
Put another way if $P(A\,|\,B)$ is a function of A then we say "it is **the probability of** A". If $P(A\,|\,B)$ is a function of B then we say "it is **the likelihood of** B".

The second thing to see is that

$$P(A,B) \neq P(A|B)$$

unless $P(B) = 1$.

In terms of data D and parameters θ we might write:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} = \frac{P(\theta,D)}{P(D)}$$

Marginalizing probabilities. If the independent choices for A can be enumerated as A_i:

$$P(B) = \sum_i P(B|A_i)P(A_i) = \sum_i P(B,A_i)$$

$$\begin{aligned} P(B) &= P(B|A_1)P(A_1) + P(B|A_2)P(A_2) \\ &= P(B,A_1) + P(B,A_2) \end{aligned}$$

Marginalizing probabilities gives yet another useful way of looking at Bayes' theorem.

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{\sum_i P(B|A_i)P(A_i)} = \frac{P(B,A_k)}{P(B)}$$

So if we are given an amino acid sequence X what is the probability it is an external protein as opposed to an internal protein? That is, what is $P(ext|X)$? Consider an amino acid sequence $X = x_1x_2x_3x_4...x_n$. Let q_a^{int} be the probability of amino acid a being in an internal protein and

q_a^{ext} be the probability of amino acid a being in an external protein. $P(int)$ is the probability that a tested sequence is internal and $P(ext)$ is the probability that a tested sequence is external.

$$p(ext|X) = \frac{P(X|ext)P(ext)}{P(X|int)P(int) + P(X|ext)P(ext)}$$

Assume all loci are independent distributions.

$$p(ext|X) = \frac{P(X|ext)P(ext)}{P(X|int)P(int) + P(X|ext)P(ext)}$$

$$P(X|ext) = \prod_{i \in EXT} q_i^{ext}$$

therefore

$$p(ext|X) = \frac{\left(\prod_{i \in EXT} q_i^{ext}\right) P(ext)}{\left(\prod_{i \in INT} q_i^{int}\right) P(int) + \left(\prod_{i \in EXT} q_i^{ext}\right) P(ext)}$$

If you can estimate the **priors** $P(ext)$ and $P(int)$. For example if you assume they are all equal you can cancel them and use the following to estimate $p(ext|X)$:

$$p(ext|X) = \frac{\left(\prod_{i \in EXT} q_i^{ext}\right)}{\left(\prod_{i \in INT} q_i^{int}\right) + \left(\prod_{i \in EXT} q_i^{ext}\right)}$$

1.1.4 Maximum Likelihood

zzz

1.2 Assumptions

We want something that is biologically reasonable given the model that we are using.

Let's look at structure of proteins [SLIDE of Classes]

Let's look at multiple sequence alignment [SLIDE of alignment] of proteins and see it is true

First let's assume in a pair that prob of any character in the sequence is as if the character is drawn from an fixed alphabet that is **independent and identically distributed** or **i.i.d.** This means we can treat each column separately and identically.

1.3 A Bit of Information Theory

More information is conveyed by things that are unexpected than expected, that is, information is a measure of the level of surprise. For example: The statement “gravity is working today” contains practically no information. If we ask a child, “Do you like ice cream?” and the child answers “yes” then the answer did not convey as much information as the surprising answer “no”.

Let p_i is the probability of symbol s_i in a data stream. If p_i is small then $1/p_i$ is large and $1/p_i$ can be thought of as the level of surprise.

The **information content** of symbol s_i, $I(s_i)$, in a data stream is

$$I(s_i) = \log_2(1/p_i) = -\log_2(p_i) >= 0$$

Note that only when you are certain of the outcome is the information content zero. This is what our intuition would say. Note also that information is a measure of a instance of a symbol in a data stream.

If the base of the log is 2 then the answer is in **bits** of information (as in binary). In a sense, this measure tells us how many bits of information are needed to encode the message. If the base of the log is e (natural log) then the answer is in **nats** (I’m not kidding).

Back to the ice cream problem. If the probability of a child answering “no” is only .1 then the information content of the “no” answer is:

$$\log_2(1/.1) = 3.3 \;\; \text{bits}$$

and for “yes”

$$\log_2(1/.9) = .15 \;\; \text{bits}$$

The term **entropy** applies to a set of symbols in an expected symbol stream or **message**. Consider a set of symbols S with each symbol s_i having a probability of occurrence of p_i. This creates a distribution for S of P as vector of probabilities. The entropy of P is the expected information content per symbol or:

$$E(I(s)) = H(P) = -\sum_i p_i \log_2(p_i)$$

Entropy does not answer a question about a particular message only the encoding of the message with a specific probability of each symbol occurring. If the log is base 2 as above then entropy represents the **expected number of bits needed to represent one character** in the stream of characters or message and so if you have k symbols in the message you can expect $kH(P)$ bits to represent the message on average.

Consider a fair coin. Compute the entropy of a stream of fair tosses.

$$-(.5\log_2(.5) + .5\log_2(.5)) = 1 \ \text{ bit}$$

So it should take 1 bit on average to represent whether it is a head or tail.

So a string of 1000 heads and tails with this probability distribution can be represented by 1000 bits on average. Without more information about the message this is the **minimum description length (MDL)** of a message from the distribution described by the probabilities of occurrence of the symbols in the set.

Now consider an unfair coin that returns 99 heads for every tail. You are on average much more certain of the outcome. The tail is a surprise. The information content of a "head" in the data stream is .0145 bits and the information content of the "tail" is 6.64 bits. The entropy is:

$$\begin{aligned} -(.99\log_2(.99) + .01\log_2(.01)) &= -(.99(-.0145) + .01(-6.64) \\ &= -((-.01435) + (-.0644)) \\ &= .081 \ \text{ bits} \end{aligned}$$

So it should take only .081 bit on average to represent whether it is a head or tail. So a string of 1000 heads and tails with this "unfair" probability distribution can be theoretically represented by only 81 bits. The information content is less per toss.

The entropy of a random DNA sequence is $4 * .25(-\log_2(1/4)) = 2$ bits. But suppose there were 90% A or T (45% of each) and 10% G or C (5% or each). That would give 1.47 bits.

The entropy of a set of n equally likely symbols is

$$n * 1/n * (-\log_2(1/n)) = \log_2(n)$$

Equal probabilities is the point of maximum unpredictability and hence maximum entropy for the symbol set. If you don't know the true distribution of the symbols then assuming an equal distribution will produce the maximum expected MDL for a message.

The **relative entropy** of a message is a comparison of the the expected distribution P with the actual observed distribution Q.

$$H(Q||P) = \sum_i (q_i \log_2(q_i) - q_i \log_2(p_i)) = \sum_i q_i \log_2(q_i/p_i)$$

$\log_2(q_i/p_i) > 0$ if s_i was more likely in the true distribution than our expected distribution. Note that often $H(Q||P) \neq H(P||Q)$. This is clearly what we are doing when we "design" a scoring matrix.

If we change the problem to one of determining the amount of information you can infer from P by observing Q then we can adapt relative entropy to this purpose. This is called **mutual information** $I(Q; P)$.

$$I(Q;P) = \sum_{i,j} Prob(i,j) \log_2 \frac{Prob(i,j)}{q_i p_j}$$

where $Prob(i,j)$ is the joint probability of seeing symbol i from Q and j from P.

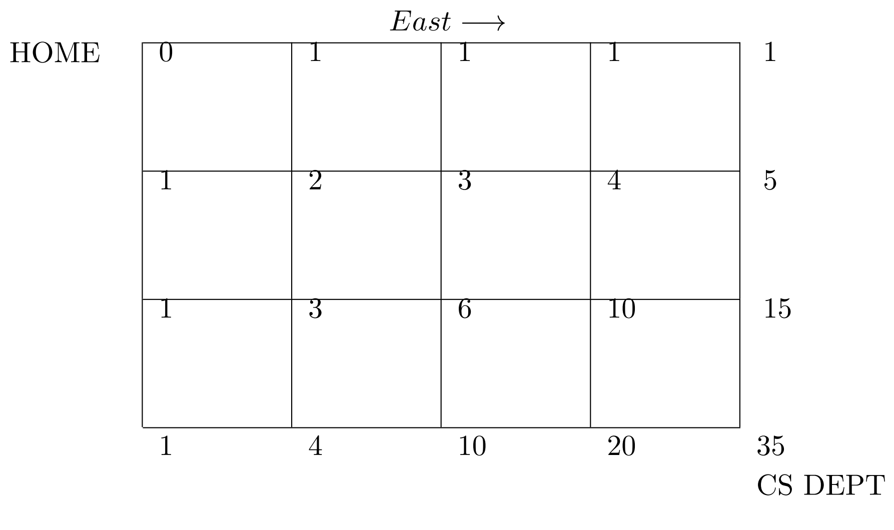
Chapter 2

Dynamic Programming and Sequence Alignment

2.1 Number of Ways to Walk to Work

bf Problem: When I was in grad school at the University of Arizona in Tucson, I used to live some number of blocks north and west of the Computer Science Department. I take different routes to work all the time. If the Computer Science Department was e blocks east and s blocks south, how many ways could I walk to work?

Tucson map layout with North at the top and East to the right:



This calculation can be written as this **recursive function** definition:

$$\begin{aligned}
&F(0,0) = 0\\
&F(0,s) = 1\\
&F(e,0) = 1\\
&F(e,s) = F(e,s-1) + F(e-1,s)
\end{aligned}$$

where e is the number of blocks you go east and s is the number of blocks you go south. $F(e,s)$ is the number of ways you cat get from HOME to the intersection e blocks east and s blocks south. Recursion makes the computation of $F(e,s)$ easy. Just start computing from $F(0,0)$ and fill in the table from upper left to lower right.

Problems where the results can be quickly computed from a problem of a smaller size in a systematic way means this can be used to solve the problem quickly if you then build up from the small problem to the intended one. This technique is called **dynamic programming**.

The total number of paths is given in closed form as:

$$F(e,s) = \binom{s+e}{e} = \binom{s+e}{s}$$

This can be seen by considering that you always travel $s+e$ blocks in going from HOME to the intersection at (e,s). Of those exactly e of them are always spent going east. And all paths consisting of $s+e$ blocks of travel with e of them going east take you to intersection (e,s).

How much work did we have do to find out the number of paths? $\mathcal{O}(s*e)$ had had to be done using dynamic programming even though the number of paths is fantastically larger.

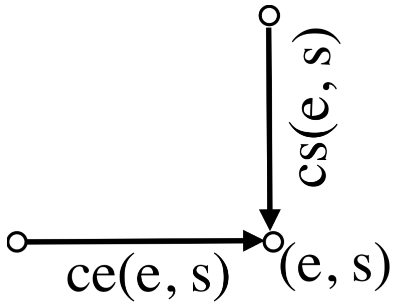
2.1.1 Maximum Number of Cans

Many problems involving maximizing profit on a path of travel is solved by leveraging the idea of recursion and using dynamic programming.

Problem: Let's look at the problem of maximizing the number of cans collected while walking to work. Let $ce(e,s)$ be the number of cans you collect in the last block if you arrived at intersection (e,s) going East. Let $cs(e,s)$ be the number of cans you collect in the last block if you arrived at intersection (e,s) going South. We want to pick the edges in our path so that the maximum number of cans are picked up.

Number of Blocks East	Computation Time	Number of Paths
1	1	2
2	4	6
3	9	20
4	16	70
5	25	252
6	36	924
7	49	3432
8	64	12870
9	81	48620
10	100	184756
11	121	705432
12	144	2704156
13	169	10400600
14	196	40116600
15	225	155117520
16	256	601080390
17	289	2333606220
18	324	9075135300
19	361	35345263800
20	400	137846528820

Table 2.1: The computation time using dynamic programming for going across a grid of blocks where the number of blocks south is the same as the number of blacks east.



This can be expressed recursively as:

$$
\begin{aligned}
F(0,0) &= 0 \\
F(0,s) &= F(0,s-1) + cs(0,s) && \text{// \textbf{add the cans for the last block going south}} \\
F(e,0) &= F(e-1,0) + ce(e,0) && \text{// \textbf{add the cans for the last block going east}} \\
F(e,s) &= \max(F(e,s-1) + cs(e,s), && \text{// \textbf{select the cans that made the most profit}} \\
&\qquad F(e-1,s) + ce(e,s))
\end{aligned}
$$

A very important question at this point is: **How do we recover the path?** What path to take is made at each recursive step in the function, for example, the point where pick *cs* or *ce* in the max function. If you picked *cs* as the maximum then that is route that gave you the number you find at $F(e,s)$. To find the path we work backwards from CS DEPT to HOME. At each point in the grid we have either:

1. save which way gave us the maximum value
2. we recompute it and redecide as we go backwards through the matrix

The choice of which way may depend on the details of what is being requested of the algorithm and how the algorithm is implemented for the particular application. When there get to be a lot of complex options or a search of the resulting matrix based on paths is needed then saving it might be the best choice. If it is relatively simple and space is an issue then recomputing is not hard. Some questions the implementer might want to ask are: What if the args to max are equal? Can there be multiple routes? How does the selection of which arg in the equal case affect the result?

2.1.2 Allowing Diagonal Streets

Suppose I am allowed to cut across the diagonally across the middle of the blocks from northwest to southeast? How do we change the formula?

$$\begin{aligned}
F(0,0) &= 0\\
F(0,s) &= F(0,s-1)+cs(0,s)\\
F(e,0) &= F(e-1,0)+ce(e,0)\\
F(e,s) &= \max(F(e,s-1)+cs(e,s),\\
&\qquad F(e-1,s)+ce(e,s),\\
&\qquad F(e-1,s-1)+cd(e,s)) \qquad \textbf{// here is the diagonal case}
\end{aligned}$$

2.2 Global Alignment (Needleman-Wunsch)

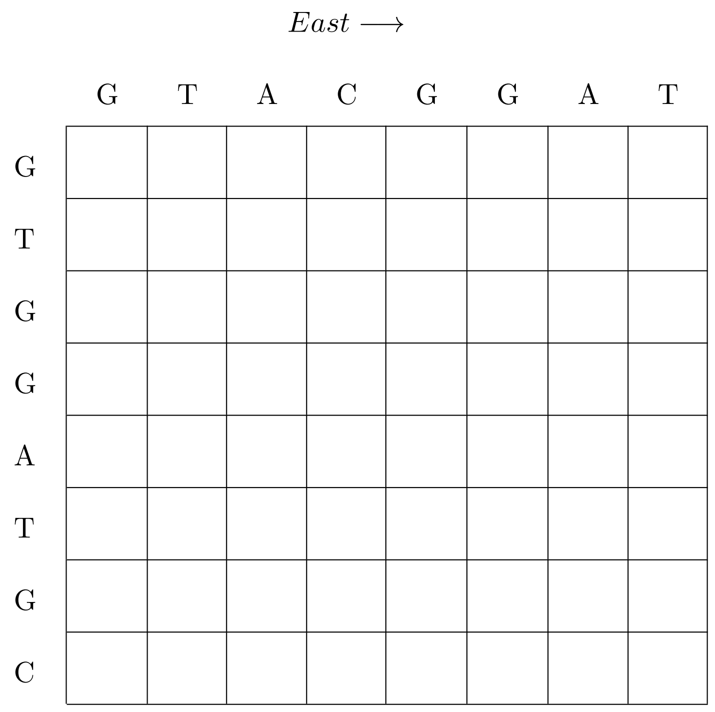
How do you match two protein or DNA sequences? What do we mean by match? What if the strings are not equal what do we say? Can there be a degree of equal? That is a very complex idea. We can create a simple approximate match based on editing a string.

Consider these two strings:

```
G   T   A   C   G   G   A   T
G   T   G   G   A   T   G   C
```

Matching two sequences is like taking a walk in which going east is moving through one string and going south is moving through the other!

```
 East: G   T   A   C   G   G   A   T
South: G   T   G   G   A   T   G   C
```



Going diagonally matches a character in both strings. Going south matches in the south string but not in the east string. That is there is a **gap** in the east string. Going east matches in the east string but not in the south string. It is important to notice that if the south string has k letters then there a $k+1$ eastward "streets". Similarly for southward streets.

Here is the can collection function:

$$
\begin{aligned}
F(0,0) &= 0 \\
F(0,s) &= F(0,s-1) + cs(0,s) && \text{// } \textbf{gap cost} \\
F(e,0) &= F(e-1,0) + ce(e,0) && \text{// } \textbf{gap cost} \\
F(e,s) &= \max(F(e,s-1) + cs(e,s), && \text{// } \textbf{gap cost} \\
&\qquad\quad F(e-1,s) + ce(e,s), && \text{// } \textbf{gap cost} \\
&\qquad\quad F(e-1,s-1) + cd(e,s)) && \text{// } \textbf{subst cost}
\end{aligned}
$$

So if cs and ce are the cost of matching a gap and cd is the profit from making a match then we try to maximize profit and we have a matching function! Here it is written out:

$$\begin{aligned}
F(0,0) &= 0\\
F(0,s) &= F(0,s-1)+g\\
F(e,0) &= F(e-1,0)+g\\
F(e,s) &= \max(F(e,s-1)+g,\\
&\qquad F(e-1,s)+g,\\
&\qquad F(e-1,s-1)+\mathit{subst}(E_e,S_s))
\end{aligned}$$

Our model is now a system of g gap profit (a negative number) and a substitution profit based on how reasonable it is to expect one character to be substituted for another: $\mathit{subst}(E_e, S_s)$. The last function is called a **substitution matrix** or **scoring matrix**. Let's assume for the moment:

$$\mathit{subst}(x,y) = \begin{cases} 2 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$
$$g = -2$$

This is one kind of match. Something we want for biologists will be to associate these rewards with the biology of a match. Let's assume that we can make biologically relevant assignments of reward.

2.3 Overlap Matching

In the overlapping match problem we only care about assigning a score for the best overlapping part of two strings. We do that by **not counting gaps at the beginning or end**.

$$\begin{aligned}
F(0,0) &= 0\\
F(0,s) &= 0\\
F(e,0) &= 0\\
F(e,s) &= \max(F(e,s-1)+g,\\
&\qquad F(e-1,s)+g,\\
&\qquad F(e-1,s-1)+\mathit{subst}(E_e,S_s))
\end{aligned}$$

AND you get to **choose the max in either final column or final row**.

When might this kind of match be important?

2.4 Local Alignment (Smith-Waterman Algorithm)

In local alignment you want to find the match of strings that has the best subsection that overlaps. Finding the best matched substring.

$$\begin{aligned}
F(0,0) &= 0 && \textbf{// could start here}\\
F(0,s) &= 0 && \textbf{// could start here}\\
F(e,0) &= 0 && \textbf{// could start here}\\
F(e,s) &= \max(0, && \textbf{// could start here}\\
&\qquad F(e,s-1)+g,\\
&\qquad F(e-1,s)+g,\\
&\qquad F(e-1,s-1)+subst(E_e,S_s))
\end{aligned}$$

AND you get to **find the largest score anywhere in the whole matrix**.

2.5 Repeat Alignment

Search through east string for best combination of segments matching from the south string.

Assume a threshold of T, to make life easier. The threshold in the algorithm prefers to not have match that is accidental, short, and irrelevant.

First row has special meaning. It stores the cost for starting a new matching segment of the south string. The function definition for $F(e,0)$ shows where we save the value of the substring that exceeds T. $F(0,0)=0$, of course. And $F(e,s)$ shows how we work it back into the matching process.

$$\begin{aligned}
F(e,0) &= \max(F(e-1,0),\\
&\qquad F(e-1,j)-T, \quad \text{for} \quad j=1..maxS) && \textbf{// if substring match is > T then save it}\\
F(e,s) &= \max(F(e,0), && \textbf{// start a new subsequence}\\
&\qquad F(e,s-1)+g,\\
&\qquad F(e-1,s)+g,\\
&\qquad F(e-1,s-1)+subst(E_e,S_s))
\end{aligned}$$

2.6 Affine Gapping

$$\begin{aligned}
F(0,0) &= 0\\
F(0,s) &= \gamma(s)\\
F(e,0) &= \gamma(e)\\
F(e,s) &= \max(F(e,z)+\gamma(s-z), \quad \text{for} \quad z=0..s-1\\
&\qquad F(z,s)+\gamma(e-z), \quad \text{for} \quad z=0..e-1\\
&\qquad F(e-1,s-1)+subst(E_e,S_s))
\end{aligned}$$

with $\gamma(x)=g_{ext}(x-1)+g_{init}$.

The above is a $\mathcal{O}(n^3)$ computation, but we can actually do better than that with a little thought.

Affine means a linear function of the form $ax + b$. So **affine gapping** means a score that has an **initial cost** for opening up a gap called g_{init} and an **extension cost** of g_{ext} per extra gap.

There are really three states you can be in when at an intersection. You can be matching the characters from both strings, matching from the east string and gapping the south, or you can match the south and gap the east.

```
case 1:
ATCGATCG
ATCGATCG

case 2:
ATCGATCG
ATCGA---

case 3:
ATCGA---
ATCGATCG
```

To convert this into a dynamic programming problem we will need to save the score from all three state at each intersection. This makes three functions rather than the single one we called F. Let M be the matching value (case 1), G_e be matching the east (case 2) and G_s matching the south (case 3).

$M(0, s)$ and $M(e, 0)$ are undefined or $-\infty$. $G_e(0, s)$ is undefined or $-\infty$. $G_s(e, 0)$ is undefined or $-\infty$.

$$\begin{aligned}
M(e,s) &= \max(M(e-1,s-1) + \mathit{subst}(E_e, S_s), \\
&\qquad G_e(e-1,s-1) + \mathit{subst}(E_e, S_s), \\
&\qquad G_s(e-1,s-1) + \mathit{subst}(E_e, S_s)) \\
G_e(e,s) &= \max(M(e-1,s) + g_{init}, \\
&\qquad G_e(e-1,s) + g_{ext}) \\
G_s(e,s) &= \max(M(e,s-1) + g_{init}, \\
&\qquad G_s(e,s-1) + g_{ext})
\end{aligned}$$

2.7 The Structure of Substitution Matrices

We want something that is biologically reasonable given the model that we are using. First, let's assume, in a pair we are matching, that probability of that pair is I.I.D. This means we can treat each column as independent of its neighbors.

Margaret Dayhoff in early 70s did a study of ungapped protein streams and derived the **PAM** (Percent Accepted Mutation) matrices as a biologically based substitution matrix for local alignments.

She asked what is the **odds ratio** for two symbols i and j being compared in two protein stings.

$$\frac{q_{i,j}}{p_i p_j} = \frac{\text{observed } (i,j) \text{ frequency}}{\text{predicted } (i,j) \text{ frequency by random chance}}$$

$q_{i,j}$ is sometimes called the **target frequency**. If $q_{i,j}/(p_i p_j) > 1$ then the pair (i, j) occurs more often than by random chance. If $q_{i,j}/(p_i p_j) < 1$ then the pair (i, j) occurs less often than by random chance.

Deviation of the odds ratio from 1 suggests nonrandom forces at work. Assume we have two strings s and t.

$$Prob(s, t|R) = \prod_i p(s_i) \prod_j p(t_j) = \prod_i p(s_i)p(t_i)$$

where R is a model of random selection from a distribution and so s_i and t_i are independent.

Now let model M be at work that assumes there is a relation between s_i and t_i because they share a common ancestor. Continuing to assume we have a correct alignment with no gaps and that the influence represented by model M applies to each column independently:

$$Prob(s, t|M) = \prod_i q(s_i, t_i)$$

$$\frac{Prob(p, q|M)}{Prob(p, q|R)} = \frac{\prod_i q(s_i, t_i)}{\prod_i p(s_i)p(t_i)} = \prod_i \frac{q(s_i, t_i)}{p(s_i)p(t_i)}$$

Looking for an additive measure we can use log:

$$\log\left(\frac{Prob(p, q|M)}{Prob(p, q|R)}\right) = \log\left(\prod_i \frac{q(s_i, t_i)}{p(s_i)p(t_i)}\right) = \sum_i \log\left(\frac{q(s_i, t_i)}{p(s_i)p(t_i)}\right)$$

$\log(q(s_i, t_i)/p(s_i)p(t_i))$ is **log odds ratio** sometimes abbreviated **lod**. The log is often log base 2 and hence the lod may be referred to in units of bits.

The choice of lod measure rather than just using the odds ratio has several **computational advantages**. Addition is much faster than multiplication on a computer. Furthermore, by keeping things in terms of log we are really looking at the exponent. Limit on the exponent in 32 bit IEEE double precision arithmetic is only about 300 (in base 10) and in 64 bit double precision is about 4930 (base 10). This means in 64 bit arithmetic on a computer from 2010, the closest you can get to 0 without being 0 is about 10^{-4930}. It won't take long doing the probability calculations associated with DNA and protein to exceed the limited available *precision* for exponents. But if we use the mantissa as the exponent by working with the logs of the numbers, things will run faster and we can preserve the precision of the numbers without overflow!

Let's take a look at some examples:

- If $p_a = .1$ and $p_b = .01$ then $p_a p_b = .001$ but if we observe .002 then odds ratio is 2. The lod is 1 in base 2.
- If $p_a = .1$ and $p_b = .01$ then $p_a p_b = .001$ but if we observe .0005 then odds ratio is .5

So what does log odds ratio give us? Let's look at a real sample **substitution matrix** or **scoring matrix**.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1

Figure 2.1: The BLOSUM62 matrix.

Note they are integers. Integers are faster and more compact than floating point numbers in computers. If we just truncated then we would lose too much precision (detail) so we first scale by a constant c:

$$S_{i,j} = Trunc[c * \log(q_{i,j}/p_i p_j)] \rightarrow \quad \text{integer}$$

$S_{i,j}$ is often called the **raw score** not the same as a lod score. The raw score is just a computationally efficient approximation for the lod score. Is there a way to recover the target frequencies from the raw score? Let's reform the problem to that of finding the more commonly used **scaling constant** λ. λ is a realvalued approximation of the effect of c and truncation in the previous formula. The relationship is:

$$\lambda S_{i,j} = \log(q_{i,j}/p_i p_j)$$

with the constraint that $\sum_{i,j} q_{i,j} = 1$, since the target frequencies are really probabilities. λ is called the **scaling constant** for the substitution matrix and $\lambda S_{i,j}$ is called the **normalized score**.

Raising exponentiating both sides of the previous equation we get

$$q_{i,j} = p_i p_j e^{\lambda S_{i,j}}$$

solved for lambda so that $\sum_{i,j} q_{i,j} = 1$ Which is the same as:

$$\sum_{i,j} p_i p_j e^{\lambda S_{i,j}} = 1$$

In fact if you give me a matrix S and observed freq p_k then I can easily apply numerical methods to approximate λ.

Example Problem: How do we compute the **percentage identical** of a substitution matrix? Percentage identical is the average number of base pairs that are equal.

Answer:

$$\sum_i q_{i,i} = \sum_i p_i^2 e^{\lambda S_{i,i}}$$

We'll see λ is important in computing the expected number of matches in a protein database search later. λ is also a value that appears on many BLAST reports.

Once we have computed our estimate of $q_{i,j}$ given S, p_i, p_j, and λ we can compute the **relative entropy** for a substitution matrix based on the idea of relative entropy of the substitution matrix vs. a hypothetical random substitution matrix as suggested by Altschule in 1991:

$$H(S) = -\sum_{i,j} q_{i,j} \log(q_{i,j}/p_i p_j)$$

which is computed using λ:

$$H(S) = -\sum_{i,j} q_{i,j} \lambda S_{i,j}$$

$H(S)$ is clearly 0 if the observed sequences are random. $H(S)$ is the average number of bits of information per position in an alignment relative to a random string.

Another statistic about the matrix is the **expected score**:

$$E(S) = \sum_{i,j} p_i p_j S_{i,j}$$

2.8 Common Substitution Matrices for proteins

2.8.1 PAM

How did Dayhof get PAM?

- selected reliable seq with no more than 15% differences
- align seq with no gaps
- constructed phylogenetic trees (using parsimony)
- looked at substitution freq as above
- and created PAM1

PAM1 is designed to represent a substitution matrix in the case where there is 1% difference between a pair of protein sequences. Repeatedly multiplying PAM1 matrices k times gives PAMk. That is

$$\text{PAM}k = \text{PAM1}^k$$

PAM250 has 250% substitutions. That is each residue pair gets a substitution 2.5 times **on average**. Note that the higher the number following PAM the less the pair match, the more substitutions, and the higher the entropy of the matrix.

Problems with Dayhof approach:

- PAM uses short term substitutions to extrapolate long term substitutions by multiplying PAM1 together. This model may not be biologically valid.
- Errors in calculations can accumulate.

Most people use BLOSUM matrices these days.

2.8.2 BLOSUM (BLOcks SUbstitution Matrix)

In the 90's we had much more protein data so we could generate better substitution matrices. What was done to create BLOSUM matrices?

- Blocks of multiply aligned gapless proteins organized by **percent identity**.

BLOSUM percentage identity so BLOSUM62 is 62% identical and BLOSUM200 doesn't exist.

More disorganized (higher entropy) as BLOSUM number goes down or PAM number goes up.

2.9 Why Log Odds Ratio Might be Related to $P(M|data)$

substitution matrix $\rightarrow$ raw score (punish and reward)

2.9.1 Applying Bayesian Reasoning to Substitution Matrices

Here is our problem: assume we are looking at a singly aligned pair of bases or residues. Let's assume we can use the comparison of just one residue with another as a model for an arbitrarily long string of ungapped identically independent residues. Given residues x and y what we want is

$$P(M|x,y)$$

We are given two models of what might be happening: M=match R=random. That means our observations are either:

$$P(x,y|M) \qquad \text{and} \qquad p(x,y|R)$$

So we should be able to use Bayes Rule here

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

Assume that either we have M or we have R then by marginalization:

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

divide through top and bottom by $P(x,y|R)P(R)$

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

note the recurrence of $\frac{P(x,y|M)P(M)}{P(x,y|R)P(R)}$:

Let $S' = \log(\frac{P(x,y|M)P(M)}{P(x,y|R)P(R)})$ then

$$S' = \underset{\text{likelihood ratio}}{\log(P(x,y|M)/P(x,y|R))} + \underset{\text{priors}}{\log(P(M)/P(R))}$$

But wait! Isn't $\log(P(x,y|M)/P(x,y|R))$ really the log odds ratio that we were computing with $\log q_{i,j}/p_ip_j$? And isn't this approximated with the $\lambda S_{i,j}$?!

This is what we compute. If we now take e to the S' we find:

$$e^{S'} = \frac{P(x,y|M)P(M)}{P(x,y|R)P(R)}$$

therefore

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

A **sigmoid function** is a function:

$$f(x) = \frac{e^x}{1+e^x}$$

This is a sigmoid function converting lod represented as S' using the sigmoid function. This converts: $[-\infty, +\infty] \rightarrow [0, 1]$. The only problem with this grand plan is that what we compute is not

$$S' = \log(P(x,y|M)/P(x,y|R)) + \log(P(M)/P(R))$$

but rather we compute the raw score

$$S' = S_{x,y}$$

We now will use λ and the sigmoid function to give meaning to the S' we do compute, even though we may not actually apply the sigmoid to our computed S'.

Let's return to our problem of estimating

$$S' = \log(P(x,y|M)/P(x,y|R)) + \log(P(M)/P(R))$$

using the substitution matrix S. From our understanding of λ a first approximation would be

$$S' = \lambda S_{x,y} + \log(P(M)/P(R))$$

What remains is what do we do about the priors $\log(P(M)/P(R))$?

Normally a good guess is $P(M)/P(R) = 1$ or $\log(P(M)/P(R)) = 0$ But, is the question we are asking is what are the odds of a single attempted match being a match? Consider searching through a large number of sequences in a database for a match. As the size of the database increases the chance of a match my accident increases linearly with the size of the database (assuming unrelated sequences). If we want a fixed number of false positives then we must set $\log(P(M)/P(R)) = \log(1/N)$ where N is the size of the database. Remember that this assumption depends on unrelated sequences in the database. The bottom line is it should reflect the anticipated number of matches.

2.10 Programs that do Sequence Alignment

Smith-Waterman (local alignment) finds the best. It is the standard by which all alignment algorithms are compared. But is that what we really want? Maybe we want the statistically significant matches. Also, we probably want to use less space and to run a faster search without loss of **sensitivity** to difficult alignments. Our goal now is to find using local alignment **High-scoring Segment Pairs HSP**s given a **query sequence**.

BLAST programs are changing and improving. These are some techniques. Not all BLAST programs use the same algorithms even if they have the same names. There are probably better ways to do this. Your mileage may vary.

2.10.1 FASTA

This FASTA format description is taken from a page on the NCBI website of the same name.

A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol in the first column. It is recommended that all lines of text be shorter than 80 characters in length. Note that **no comment lines beginning with semicolons are allowed in the NCBI format.** An example sequence in FASTA format is:

```
>gi|532319|pir|TVFV2E|TVFV2E envelope protein
ELRLRYCAPAGFALLKCNDADYDGFKTNCSNVSVVHCTNLMNTTVTTGLLLNGSYSENRT
QIWQKHRTSNDSALILLNKHYNLTVTCKRPGNKTVLPVTIMAGLVFHSQKYNLRLRQAWC
HFPSNWKGAWKEVKEEIVNLPKERYRGTNDPKRIFFQRQWGDPETANLWFNCHGEFFYCK
MDWFLNYLNNLTVDADHNECKNTSGTKSGNKRAPGPCVQRTYVACHIRSVIIWLETISKK
TYAPPREGHLECTSTVTGMTVELNYIPKNRTNVTLSPQIESIWAAELDRYKLVEITPIGF
APTEVRRYTGGHERQKRVPFVXXXXXXXXXXXXXXXXXXXXXXVQSQHLLAGILQQQKNL
LAAVEAQQQMLKLTIWGVK
```

Sequences are expected to be represented in the standard IUB/IUPAC amino acid and nucleic acid codes, with these exceptions: lower-case letters are accepted and are mapped into upper-case; a single hyphen or dash can be used to represent a gap of indeterminate length; and in amino acid sequences, U and * are acceptable letters (see below). Before submitting a request, any numerical digits in the query sequence should either be removed or replaced by appropriate letter codes (e.g., N for unknown nucleic acid residue or X for unknown amino acid residue).

The nucleic acid codes supported are:

```
A --> adenosine          M --> A C (amino)
C --> cytidine           S --> G C (strong)
G --> guanine            W --> A T (weak)
T --> thymidine          B --> G T C
```

```
U --> uridine             D --> G A T
R --> G A (purine)        H --> A C T
Y --> T C (pyrimidine)    V --> G C A
K --> G T (keto)          N --> A G C T (any)
                          -  gap of indeterminate length
```

For those programs that use amino acid query sequences (BLASTP and TBLASTN), the accepted amino acid codes are:

```
A  alanine                     P  proline
B  aspartate or asparagine     Q  glutamine
C  cystine                     R  arginine
D  aspartate                   S  serine
E  glutamate                   T  threonine
F  phenylalanine               U  selenocysteine
G  glycine                     V  valine
H  histidine                   W  tryptophan
I  isoleucine                  Y  tyrosine
K  lysine                      Z  glutamate or glutamine
L  leucine                     X  any
M  methionine                  *  translation stop
N  asparagine                  -  gap of indeterminate length
```

2.10.2 The BLAST Programs

Basic Local Alignment Search Tool (BLAST)

Name	Query	Database	Uses
BLASTP	Protein	Protein	good for finding distant relationships
BLASTN	Nucleotide	Nucleotide	tuned for high scoring close relationships
BLASTX	T(Nucleotide)	Protein	useful for a first pass look at new DNA and ESTs (cDNA)
TBLASTN	Protein	T(Nucleotide)	finding unannotated coding regions in database
TBLASTX	T(Nucleotide)	T(Nucleotide)	useful for EST analysis, computationally expensive

BLAST proceeds by:

$$\text{SEEDING} \rightarrow \text{EXTENSION} \rightarrow \text{EVALUATION}$$

A **word** is a w**-mer** from a sequence.

Step 1: Ignore much of the search space by seeding: Find words that score at least T relative to a word in the query sequence. This is called a **word hit**. We will only consider sequences that contain word hits.

As T increases the number of word hits goes down but the chance of missing something goes up. What does T really say about HSPs? It is a speed/sensitivity trade-off.

Word size W.

In order for this to work effectively the dictionary of W-mers must be computed once. Then as many queries can be made as you like at little cost.

Step 2: (two hit filter)

Consider only sequences which have two word hits on a diagonal within distance D of one another. Sorting hits by difference of coordinates picks diagonals so you can search.

NCBI versions: BLASTN uses identical words and no T. $W \geq 7$. No two hit filter. BLASTP uses $W = 2, 3$ WU versions: W can have any value for any program. If $W > 5$ then T is not used.

Step 3: (low complexity filtering) Look at this later.

Step 4: (banded alignment)

bandwidth

Step 5: (ungapped extension)

Example: +1 match −1 mismatch $X = 5$ which is a **dropoff limit**

```
The quick brown fox jumped over the lazy dogs back
The quiet brown cat purred when she saw him
123 45654 56789 876 565456
000 00012 10000 123 434543
...  ..cutoff | xxx xxxxxx
```

The end of extension occurs when the difference between the maximum and the current score is X or less, which happens at the second "`r`" in `purred`. You keep everything up to the current max. Do this in both directions. In the above case what happens when $X = 2$? $X = 3$? $X = 666$?

Step 6: (ungapped threshold filter or significance evaluation)

We now test against the expected number of matches and keep the matches that are statistically significant. Those are called **HSP**s. or **high scoring pairs**.

Step 7: (evaluation and gapped extension)

Assembly of ungapped HSPs from **consistent** subHSPs. **consistent** subHSPs are HSP that continue from one to the next by proceeding down and to the right in the dynamic programming matrix without overlapping. A **gapped dropoff limit** is used to find the cutoff for assembling ungapped subHSPs

Step 8: (final threshold filter)

The total score must be better than this to make the final cut.

2.10.3 BLAST statistics

As we all know if we define a distribution based on the sum of samples:

$$\sum(x_1, x_2, \ldots, x_N) = M_N$$

where the x_i are drawn from independent random populations then M_N is normally distributed. What if we have

$$\max(x_1, x_2, \ldots, x_N) = M_N$$

Then M_N is distributed with the **extreme value distribution (EVD)** The expected number of matches

$$E(M_N < x) \approx e^{-KNe^{\lambda(x-\mu)}}$$

Consider a local search as comparing an m length string against an n length string. Then the dynamic programming algorithm will look at nm potential starting points for a sum of scores to find the maximal score. This lead Karlin-Altschul (ref to follow) to use the extreme value distribution above in the context of a BLAST search. To complete the analogy with the EVD case we assume that a BLAST search is comparing the m character query string with all the strings in the database concatenated together to create an n length string. The conclusion is called the **Karlin-Altschul Statistic**

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

where: $E(S)$ is the expected number of gapless matches with a raw score **greater than** S and a scaling constant λ for that S. m and n are the size of the query string and the database in symbols respectively. And K is a constant that measures the relative independence of the scores in the dynamic programming matrix (insert hand waving here).

Now we can consider the "expected number of gapless matches with a raw score **greater than** S" to be an event in a Poisson process. (Why?) Then we see that probability by chance of having a score higher than the value S we found is

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

This assumes:

1. that i.i.d. sequences
2. the sequences are effectively infinite in length
3. that average value of the substitution matrix is negative but that positive alignment scores are possible

If the sequences are short, that is m or n are short then we need to compensate for length. As dynamic programming approaches the end of a string in the matching process a good match may be cut short by the end of the string. This is called **edge effects**. This means that potentially good starting points in the dynamic programming matrix will be cut off, but they were allowed for in the count of nm possible beginning points of traceback paths in the dynamic programming matrix. So some algorithms reduce the values of n and m by the expected length of a match. Some chose this value to be proportional to $\log(n)$.

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

One more statistic. The **query coverage** is how long piece of the sequence is covered by the one found. It is the percentage of the query sequence that was aligned in the final local alignment.

2.10.4 The dotter Program

One way to visualize alignments between two strings, x and y, is to use the **dotter program**. Imagine a rectangular field of pixels where the pixel at location (i, j) is indexed where i is the i^{th} character in the first string and j is the j^{th} character in the second string. For all $(i, j) : x_i = x_j$ then place a dot at pixel (i, j). Visually, diagonals form where a series of contiguous matches occur.

Dotter can be used to visually locate repeated matches between subsequences of two strings.

Dotter is also useful in visually locating areas called **low complexity regions LCR**s. These are areas with low information content. For example a significant region that might be composed of nothing but aminio acids G, Y, and A. This causes a square smudge on the dotter picture. See images in lecture slides.

2.10.5 SEG program

The **SEG program** is used to look for LCRs. SEG uses a moving window and computes LCR in two passes.

In phase 1 compositional complexity is measured for a window of length L and alphabet of size N as:

$$K_1 = \frac{1}{L} \log_N \frac{L!}{\Pi_{i=1}^{N} n_i!}$$

where n_i is the number of occurrences of the i^{th} symbol in the window. Note that $0 \leq K_1 \leq 1$. This represents the number of possible substrings of length L with the given set of n_i.

A second "more efficient" measure to compute is used which is based on the information content:

$$K_2 = -\sum_{i=1}^{N} \frac{n_i}{L} \log_2 \left(\frac{n_i}{L}\right)$$

The window is moved along one symbol at a time measuring the value of K_2. Overlapping windows where $K_2 \leq \widehat{K}_2$ are merged to form potential LCRs. potential LCRs are merged if the windows between have complexity $K_2 \leq \widehat{\widehat{K}}_2$ are merged to form larger LCRs where $\widehat{\widehat{K}} > \widehat{K}$.

In the second phase each LCR is reduced to the subsequence which minimizes

$$P_0 = \frac{1}{N^L} \frac{L!}{\Pi_{i=1}^{N} n_i!} \frac{N!}{\Pi_{i=1}^{L} r_i!}$$

where r_k is the count of the number to times k occurs in the vector of n_i is found by exhaustive search.

Chapter 3

Markov Models and Hidden Markov Models

3.1 The Monkey Typewriter Problem

```
ThermusThermophilus.out
AA  0.030
AC  0.040
AG  0.067
AT  0.016

CA  0.046
CC  0.152
CG  0.084
CT  0.067

GA  0.062
GC  0.095
GG  0.149
GT  0.040

TA  0.015
TC  0.063
TG  0.045
TT  0.028
gcContent:  0.695
```

Table of **adjacent** nucleotides.

```
The Raven
by Edgar Allen Poe

  Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
"'Tis some visitor," I muttered, "tapping at my chamber door-
Only this, and nothing more."

Ah, distinctly I remember it was in the bleak December,
And each separate dying ember wrought its ghost upon the floor.
Eagerly I wished the morrow;- vainly I had sought to borrow
From my books surcease of sorrow- sorrow for the lost Lenore-
For the rare and radiant maiden whom the angels name Lenore-
Nameless here for evermore.


==============================+==============================
theRaven1.txt

ihrie n rrrelahenaem la rnmdit iew dofiaatduutes-rrdusntl ogawnibn eriw;h n
rti"ptnaosiboh;tpagr-g h awlpt p svtogeb gtiadaflhcerdee"  visr wmh
reiorh!o.vmfsli mth"o,ol'llewaiere lt baslr-rit mm c,ors


==============================+==============================
theRaven2.txt

teraththence bere thaire thatapith modathermomy ebeng berumulevephean, "g
meave-nof funner! ncon'erinco nt ndilishet g s s hor-tle hearcid "lin,
sthifrk nbyoororgllotifeves s theveabeeveghapate lotho


==============================+==============================
theRaven3.txt

t but and curplon theashe i scul have re; hesto teme wing beand thy
bermonevere. neventh any hut, untrave saidess taphentor daunt derechamilency
bled nep a shin thiscorrophe th sudden getch, disird thee


==============================+==============================
theRaven4.txt

at there floor. "thought's perciful his dreams nappy me, angels help again if
bird, "'tis dreams name wondering, stor, with sent of pall my dared and soul
front is thy god word or that wand he streaming


==============================+==============================
theRaven5.txt

ntly you came as a tapping, still leaven stillness ungainly word, "thought
tossed this kind nothing than muttered, "thy fowl whose fiery expresent this
hope thy beak and and curious bird or we both ther t


==============================+==============================
theRaven6.txt

i implore-is there balm in guessing, stillness gave no craven, "nevermore.
deep into smiling, and weary, over many a quaint and nothing more that
melancholy burden bore-tell me with mien of lordly nappi
```

3.2 Markov Models

Last time we assumed that the strings we were matching were i.i.d. We know there are complicated patterns in DNA/proteins this requires that we be able to identify and model variations from one base/residue to the next in the sequence. Markov models will help us do that.

A **Markov model** is a set of states S and a matrix of probabilities P where $p_{i,j}$ is the probability of going from state i to j. Notice that the **state transition probability** is dependent only on i and j, where you are and where you are going, and not on any history of how you got to i. Clearly:

$$\sum_j p_{i,j} = 1$$

but NOT necessarily:

$$\sum_i p_{i,j} = 1$$

As can be seen in the first example below.

A Markov model can be thought of as a set of nodes connected by probability arcs. Consider this example of a model of who buys what cereal (Wheaties or Count Chocula):

It can be modeled as this modeled as this matrix:

$$P = \begin{bmatrix} .8 & .2 \\ .3 & .7 \end{bmatrix}$$

If we think the initial purchasers of the ceral are .2 for Wheaties and .8 for Count Chocula we can represent these initial conditions as a row vector:

$$\pi = \begin{bmatrix} .2 & .8 \end{bmatrix}$$

Now multiplying πP gives the projected distribution after one move through the network. By repetition of that process πP^N gives us the projected distribution after N moves through the network. P^N are the **multistep transition probabilities** for N steps. So

$$\pi(N) = \pi P^N$$

where $\pi(N)$ is the **state probabilities** at step N with $\pi(0)$ being the initial state probabilities.

$$P^1 = \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}$$

$$P^2 = \begin{bmatrix} 0.7 & 0.3 \\ 0.45 & 0.55 \end{bmatrix}$$

$$P^3 = \begin{bmatrix} 0.65 & 0.35 \\ 0.525 & 0.475 \end{bmatrix}$$

$$P^4 = \begin{bmatrix} 0.625 & 0.375 \\ 0.5625 & 0.4375 \end{bmatrix}$$

$$P^5 = \begin{bmatrix} 0.6125 & 0.3875 \\ 0.58125 & 0.41875 \end{bmatrix}$$

$$\vdots$$

$$P^\infty = \begin{bmatrix} .6 & .4 \\ .6 & .4 \end{bmatrix}$$

P^∞ is the **limiting transition probability**. Note that initial distribution is irrelevant *in this case.* That is, it increasingly forgets the initial condition. This is true of most but not all MMs.

The probability of seeing a given path of states x of length L under the Markov model P is:

$$\begin{aligned} Prob(x \mid P) &= Prob(x_L \mid x_{L-1}) Prob(x_{L-1} \mid x_{L-2}) ... Prob(x_1) \\ &= Prob(x_1) \prod_{i=2}^{L} Prob(x_i \mid x_{i-1}) \\ &= Prob(x_1) \prod_{i=2}^{L} p_{x_{i-1},x_i} \end{aligned}$$

and

$$\begin{aligned}\log Prob(x \mid P) &= \log\left(Prob(x_1)\prod_{i=2}^{L} Prob(x_i \mid x_{i-1})\right)\\ &= \log(Prob(x_1)) + \sum_{i=2}^{L}\log(p_{x_{i-1},x_i})\end{aligned}$$

Now lets apply this to the problem in Chapt 3. We will look for CpG islands. These are regions on DNA where Cs and Gs occur much more frequently. Let's see if we can identify those regions.

Since Markov Models only need to know the probabilities of a transition from what state they are in to any next state we need only compute that information from examining a set of **training data**. In our case we have two sets. the CpG islands and the nonCpG islands. This gives us two matrices P^+ for inside the island and P^- for outside the island.

$$P^+ = \begin{bmatrix} .180 & .274 & .426 & .120 \\ .171 & .368 & .274 & .188 \\ .161 & .339 & .375 & .125 \\ .079 & .355 & .384 & .182 \end{bmatrix} \quad P^- = \begin{bmatrix} .300 & .205 & .285 & .210 \\ .322 & .298 & .078 & .302 \\ .248 & .246 & .298 & .208 \\ .177 & .239 & .292 & .292 \end{bmatrix}$$

Note the concentration of higher values in the center for C and G mixes in the P^+ over the P^-.

So let's look at the log odds ratio for a given string x:

$$S(x) = \log\left(\frac{Prob(x \mid P^+)}{Prob(x \mid P^-)}\right) = \log\left(\frac{Prob(x_1 \mid P^+)\prod_{i=2}^{L} p^+_{x_{i-1},x_i}}{Prob(x_1 \mid P^-)\prod_{i=2}^{L} p^-_{x_{i-1},x_i}}\right) = \log\left(\frac{Prob(x_1 \mid P^+)}{Prob(x_1 \mid P^-)}\right) + \sum_{i=2}^{L}\log\left(\frac{p^+_{x_{i-1},x_i}}{p^-_{x_{i-1},x_i}}\right)$$

If you assume $Prob(x_1 \mid P^+)/Prob(x_1 \mid P^-) \approx 1$ then you can get a score from:

$$S(x) = \log\left(\frac{Prob(x \mid P^+)}{Prob(x \mid P^-)}\right) \approx \sum_{i=2}^{L}\log\left(\frac{p^+_{x_{i-1},x_i}}{p^-_{x_{i-1},x_i}}\right)$$

This gives us a new kind of scoring matrix β where each entry is the $\log_2$ of the ratio of the two models we want to test.

$$\beta_{i,j} = \log\left(\frac{p^+_{x_i,x_j}}{p^-_{x_i,x_j}}\right)$$

$$\beta = \begin{bmatrix} -0.734 & 0.419 & 0.580 & -0.807 \\ -0.913 & 0.304 & 1.813 & -0.684 \\ -0.623 & 0.463 & 0.332 & -0.735 \\ -1.164 & 0.571 & 0.395 & -0.682 \end{bmatrix}$$

NOTE: The scoring is not performed between two corresponding base/residues in two sequences but rather between two adjacent base/residues in the same sequence.

We could now use a moving window to repeatedly ask the question: "is this string in the window a CpG island or not?". Seems very *ad hoc* and awkward. We must propose millions of strings as potential strings in an island. I think this is the wrong question to ask. So let's build a model that allows for modeling being in either region as we move through the string. It should identify the CpG islands as we come across them. I am envisioning a Markov like machine that will have two parts as part of the state machine, one for "on the island" and one for "off the island".

To do this we will need a new more powerful model called a Hidden Markov Model.

3.3 The Hidden Markov Model

A **Hidden Markov Model (HMM)** is a set of states S and a matrix of transition probabilities P where $p_{i,j}$ is the probability of going from state i to j or $P(i \mid j)$, just like a MM. (In the remaining sections of this document P will be used to represent probability.) HMM are models that we aren't allowed to view the actual states but rather only get a view of them via **emissions probabilities**. $e_i(j)$ is the probability of observing output j given being in state i. In an HMM it is possible to observe an exponentially large number of sequences x for a given **path** π through the states. For example each possible state in S might have two symbols that could be emitted that have nonzero probabilities. Then a path of π of L states could have 2^L potential sequences x. Also given a sequence x there could be many paths that generate the same sequence of symbols. The assumption with HMM is that the observed sequence x is seen but the path through the states π is unknown.

3.3.1 The Loaded Dice Problem

What it is. **Note: expand**

The HMM generalizes the idea that each condition a system is in has a possibly unique distribution of possible observations.

3.3.2 Probability of Seeing x and Having a State Sequence π

What is the probability of seeing x and having state sequence π? That is:

$$P(x, \pi) = P(x \mid \pi)P(\pi)$$

which can be computed from:

$$P(x \mid \pi) = e_{\pi_L}(x_L)e_{\pi_{L-1}}(x_{L-1}) \dots e_{\pi_1}(x_1)$$

$$P(\pi) = P(\pi_L \mid \pi_{L-1})P(\pi_{L-1} \mid \pi_{L-2}) \dots P(\pi_1)$$

therefore

$$\begin{aligned} P(x,\pi) &= e_{\pi_1}(x_1)P(\pi_1)e_{\pi_2}(x_2)P(\pi_2 \mid \pi_1)\dots e_{\pi_L}(x_L)P(\pi_L \mid \pi_{L-1}) \\ &= e_{\pi_1}(x_1)P(\pi_1)\prod_{i=2}^{L} e_{\pi_i}(x_i)P(\pi_i \mid \pi_{i-1}) \end{aligned}$$

If you know both x and π then $P(x,\pi)$ can be computed.

3.3.3 Most Likely Path (Viterbi Algorithm)

We'd like to know:

$$\operatorname*{argmax}_{\pi} P(x,\pi)$$

This can be found recursively using dynamic programming. Consider that we don't know what state we have at position i in the sequence with symbol x_i emitted. We will build a matrix V out of $v_k(i)$ where k is the hidden state of the Markov model and i is the position in the sequence.

> Let $v_k(i) = \max_{\pi_1,\dots,\pi_{i-1},\pi_i=k} P(x_1, x_2, \dots x_i|\pi_1, \dots, \pi_{i-1}, \pi_i = k)$ be the probability of seeing the sequence up to x_i given that the most likely path was taken to arrive at state k for observed sequence $x_1, x_2, \dots x_{i-1}$.

Then

$$v_k(i) = e_k(x_i)\max_{z} P(k \mid z)v_z(i-1), \qquad i > 1 \qquad\qquad \text{and} \qquad\qquad v_k(1) = e_k(x_1)$$

The transition probabilities between states, $P(k \mid z)$, are known from the HMM. The emission probabilities, $e_k(x_i)$, are also known from the HMM.

Now $\operatorname{argmax}_k v_k(L)$ is the most probable final state. $v_k(i)$ forms a dynamic programming matrix that is $\mid S \mid \times L$ in size indexed by k and i. Traceback can occur by keeping track of where the max at each point came from and starting with the final state and working backwards. This allows us to use the matrix of $v_k(i)$ to compute the path, π, that got us the max or $\operatorname{argmax}_\pi P(x,\pi)$

In the book, we see in the Case of the Dishonest Casino, Part 2, an examination of the matrix $v_k(i)$ gives a estimate of the most probable state, π_i for any x_i and this tells us whether we believe, given our observation of the sequence x, that x_i was generated by a fair or loaded die.

3.3.4 Probability of Observed Sequence (Forward Algorithm)

We want:

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

Again we can answer this with dynamic programming. Where before we wanted the path that is maximum now we replace the idea of taking the max at each step through the matrix with the summing at each step.

> Let $f_k(i) = P(x_1, x_2, \ldots x_i, \pi_i = k)$, that is, the probability of seeing the sequence up through position i and being in state k for position i.

Unlike $v_k(i)$, the forward adds in all possible paths through the states that yield x. In the end we answer our original question with:

$$P(x) = \sum_k f_k(L)$$

where:

$$f_k(i) = e_k(x_i) \sum_z P(k \mid z) f_z(i-1), \qquad i > 1 \qquad \text{and} \qquad f_k(1) = e_k(x_1)$$

$f_k(i)$ now forms a dynamic programming matrix that is $\mid S \mid \times L$ and gives the running probability of x_i in the sequence x being generated by state k.

Remember that although it can save computer memory to compute this matrix using only the last column computed and discarding the rest, the whole matrix may be useful in other computations as we will see.

3.3.5 Probability of Sequence x and $\pi_i = k$ (Backward Algorithm)

This time what we want is $P(x, \pi_i = k)$, which is the probability of seeing all of x given $\pi_i = k$. We find

$$\begin{aligned} P(x, \pi_i = k) &= P(x_1, x_2, \ldots x_i, \pi_i = k) P(x_{i+1}, x_{i+2}, \ldots x_L \mid x_1, x_2, \ldots x_i, \pi_i = k) \\ &= P(x_1, x_2, \ldots x_i, \pi_i = k) P(x_{i+1}, x_{i+2}, \ldots x_L \mid \pi_i = k) \\ &= f_k(i) P(x_{i+1}, x_{i+2}, \ldots x_L \mid \pi_i = k) \end{aligned}$$

This is like having the dynamic programming bottleneck at position i in state k. To see this, let me expand on what was done. What we are reallying to do is:

$$\sum_{\pi, \pi_i = k} P(x) = \sum_{\pi_L} e_{\pi_L}(x_L) P(\pi_L \mid \pi_{L-1}) \ldots e_k(x_i) P(k \mid \pi_{i-1}) \cdots \sum_{\pi_1} e_{\pi_1}(x_1) P(\pi_1)$$

Although we could easily compute it this way using only left to right progressing dynamic programming, we will want to know the value for all possible bottlenecks defined by all possible positions and states in the matrix. We can do that by doing the dynamic programming conceptually from right to left.

> Let $b_k(i) = P(x_{i+1}, x_{i+2}, \ldots x_L, \pi_i = k)$, that is, the probability of seeing the remainder of the sequence if you start from state k.

$$b_k(i) = \sum_z P(z \mid k) e_k(x_{i+1}) b_z(i+1), \qquad i > 1 \qquad \text{and} \qquad b_k(L) = e_k(x_L)$$

3.3.6 Most Probable State for x_i

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

3.3.7 Review of Matrices used

$$\begin{aligned}
v_k(i) &= \max_{\pi_1,\ldots,\pi_{i-1},\pi_i=k} P(x_1, x_2, \ldots x_i | \pi_1, \ldots, \pi_{i-1}, \pi_i = k) \\
b_k(i) &= P(x_{i+1}, x_{i+2}, \ldots x_L, \pi_i = k) \\
f_k(i) &= P(x_1, x_2, \ldots x_i, \pi_i = k)
\end{aligned}$$

3.3.8 Revisiting $\sum_\pi P(x,\pi)$

Consider these reformulations:

Part I: derivation of forward algorithm

$$\begin{aligned}
&\sum_\pi P(x,\pi)\\
&= \sum_{\pi_L}\sum_{\pi_{L-1}}\cdots\sum_{\pi_2}\sum_{\pi_1} e_{\pi_L}(x_L)P(\pi_L \mid \pi_{L-1})e_{\pi_{L-1}}(x_{L-1})P(\pi_{L-1} \mid \pi_{L-2})\ldots e_{\pi_2}(x_2)P(\pi_2 \mid \pi_1)e_{\pi_1}(x_1)P(\pi_1)\\
&= \sum_{\pi_L} e_{\pi_L}(x_L)\sum_{\pi_{L-1}} P(\pi_L \mid \pi_{L-1})e_{\pi_{L-1}}(x_{L-1})\sum_{\pi_{L-2}} P(\pi_{L-1} \mid \pi_{L-2})e_{\pi_{L-2}}(x_{L-2})\sum_{\pi_{L-3}} P(\pi_{L-2} \mid \pi_{L-3})\ldots\\
&\qquad \sum_{\pi_4} P(\pi_5 \mid \pi_4)e_{\pi_4}(x_4)\sum_{\pi_3} P(\pi_4 \mid \pi_3)e_{\pi_3}(x_3)\sum_{\pi_2} P(\pi_3 \mid \pi_2)e_{\pi_2}(x_2)\sum_{\pi_1} P(\pi_2 \mid \pi_1)e_{\pi_1}(x_1)P(\pi_1)\\
&= \sum_{\pi_L} f_{\pi_L}(L)\\
&= \sum_{\pi_L} e_{\pi_L}(x_L)\sum_{\pi_{L-1}} P(\pi_L \mid \pi_{L-1})f_{\pi_{L-1}}(L-1)
\end{aligned}$$

therefore $f_k(i) = e_k(x_i)\sum_z P(k \mid z)f_z(i-1)$ this is equivalent to equation 3.11 in the book. Finally we see that

$$f_k(i) = \sum_{\pi_{i-1}}\sum_{\pi_{i-2}}\cdots\sum_{\pi_2}\sum_{\pi_1} e_k(x_i)P(k \mid \pi_{i-1})e_{\pi_{i-1}}(x_{i-1})P(\pi_{i-1} \mid \pi_{i-2})\ldots e_{\pi_2}(x_2)P(\pi_2 \mid \pi_1)e_{\pi_1}(x_1)P(\pi_1)$$

Part II: derivation of backward algorithm

$$\begin{aligned}
&\sum_\pi P(x,\pi)\\
&= \sum_{\pi_L}\sum_{\pi_{L-1}}\cdots\sum_{\pi_2}\sum_{\pi_1} P(\pi_1)e_{\pi_1}(x_1)P(\pi_2 \mid \pi_1)e_{\pi_2}(x_2)\ldots P(\pi_{L-1} \mid \pi_{L-2})e_{\pi_{L-1}}(x_{L-1})P(\pi_L \mid \pi_{L-1})e_{\pi_L}(x_L)\\
&= \sum_{\pi_1} P(\pi_1)e_{\pi_1}(x_1)\sum_{\pi_2} P(\pi_2 \mid \pi_1)e_{\pi_2}(x_2)\sum_{\pi_3} P(\pi_3 \mid \pi_2)e_{\pi_3}(x_3)\ldots\\
&\qquad \sum_{\pi_{L-2}} P(\pi_{L-2} \mid \pi_{L-3})e_{\pi_{L-2}}(x_{L-2})\sum_{\pi_{L-1}} P(\pi_{L-1} \mid \pi_{L-2})e_{\pi_{L-1}}(x_{L-1})\sum_{\pi_L} P(\pi_L \mid \pi_{L-1})e_{\pi_L}(x_L)\\
&= \sum_{\pi_1} P(\pi_1)e_{\pi_1}(x_1)b_{\pi_1}(1)\\
&= \sum_{\pi_1} P(\pi_1)e_{\pi_1}(x_1)\sum_{\pi_2} P(\pi_2 \mid \pi_1)e_{\pi_2}(x_2)b_{\pi_2}(2)
\end{aligned}$$

$$= \sum_{\pi_1} P(\pi_1) e_{\pi_1}(x_1) \sum_{\pi_2} P(\pi_2 \mid \pi_1) e_{\pi_2}(x_2) \sum_{\pi_3} P(\pi_3 \mid \pi_2) e_{\pi_3}(x_3) b_{\pi_3}(3)$$

therefore $b_k(i) = \sum_z P(z \mid k) e_z(x_{i+1}) b_z(i+1)$ this is equivalent to the recursive equation in the book for the backward algorithm. Finally we see that:

$$b_k(i) = \sum_{\pi_L} \sum_{\pi_{L-1}} \cdots \sum_{\pi_{i+2}} \sum_{\pi_{i+1}} P(\pi_{i+1} \mid k) e_{\pi_{i+2}}(x_{i+1}) \ldots P(\pi_{L-1} \mid \pi_{L-2}) e_{\pi_{L-1}}(x_{L-1}) P(\pi_L \mid \pi_{L-1}) e_{\pi_L}(x_L)$$

Part III: derivation when π_i is known

We are looking for $\sum_{\pi} P(x, \pi, \pi_i = k)$ Begin with the expansion used in parts I and II:

$$= \sum_{\pi_L} \sum_{\pi_{L-1}} \cdots \sum_{\pi_2} \sum_{\pi_1} e_{\pi_L}(x_L) P(\pi_L \mid \pi_{L-1}) e_{\pi_{L-1}}(x_{L-1}) P(\pi_{L-1} \mid \pi_{L-2}) \ldots$$
$$e_{\pi_{i+1}}(x_{i+1}) P(\pi_{i+1} \mid \pi_i) e_{\pi_i}(x_i) P(\pi_i \mid \pi_{i-1}) \ldots e_{\pi_2}(x_2) P(\pi_2 \mid \pi_1) e_{\pi_1}(x_1) P(\pi_1)$$

insert the information that $\pi_i = k$:

$$\sum_{\pi} P(x, \pi, \pi_i = k) = \sum_{\pi_L} \sum_{\pi_{L-1}} \cdots \sum_{\pi_2} \sum_{\pi_1} e_{\pi_L}(x_L) P(\pi_L \mid \pi_{L-1}) e_{\pi_{L-1}}(x_{L-1}) P(\pi_{L-1} \mid \pi_{L-2}) \ldots$$
$$e_{\pi_{i+1}}(x_{i+1}) P(\pi_{i+1} \mid k) e_k(x_i) P(k \mid \pi_{i-1}) \ldots e_{\pi_2}(x_2) P(\pi_2 \mid \pi_1) e_{\pi_1}(x_1) P(\pi_1)$$

Now we can see that:

$$\sum_{\pi} P(x, \pi, \pi_i = k) = f_k(i) b_k(i)$$

and that

$$\sum_{\pi} P(x, \pi, \pi_i = k, \pi_{i+1} = j) = f_k(i) e_j(x_{i+1}) P(k \mid j) b_k(i+1)$$

which is the same as equation 3.19 in the book.

3.4 Predicting Region Classification

We now have enough technique to apply answer where the CpG islands are in a variety of ways with a variety of success. Here are four techniques:

- **Varying Window**. Repeatedly ask if specific strings are in the class of interest. This can be exponentially slow if you are probing to discover the classification of subsequences in a longer one. In short don't do this for subsequence discovery.
- **Best fit path to sequence (Viterbi)**. Finds the best path π^*. Generally this assumes the best path is greatly superior to the second best path which might be quite different in its

state sequence. Really this algorithm works quite well given that it assumes an underlying Markov model.

- **Posterior Decoding** In this approach we compute $P(\pi_i = k \mid x)$ by dynamic programming above computing, an f and b matrix and combining it with $P(x)$.

$$\hat{\pi}_i = \operatorname*{argmax}_k P(\pi_i = k \mid x)$$

  This gives us a sequence that of $\pi_1, \pi_2, \ldots \pi_L$ where at each point the most likely state is chosen. This may not create anything that looks like the Viterbi path. That is probably $\pi^* \neq \hat{\pi}$.

- **Weighted Posterior Decoding**

$$G_i(x) = \sum_k P(\pi_i = k \mid x) g(k)$$

Any required interactions between elements that are widely separated in the sequence are not detectable by this method except by complex machines with a large number of states. Adjustments to remove states and transitions between states are need to keep the HMMs manageable.

3.5 Parameter Estimation

Given a training set of sequences $x^1, x^2, x^3, \ldots x^n$ (superscripts for numbering sequences and subscripted to represent individual characters in the sequence) and assuming a topology of states we want to find the most likely transition and emission probabilities. We will refer to the set of parameters as θ. We do this maximizing the log probabilities:

$$\log P(x^1, x^2, x^3, \ldots x^n \mid \theta) = \sum_j \log P(x^j \mid \theta)$$

3.5.1 ...if we know π

If we know π^i for each x^i then we know all the transitions and emissions that took place and so we can use the sample as an estimator for the true probabilities. We do this by taking a count of the the number of transitions of each pair of states and the number each kind of symbol emission for each state. “Simples”

Let $A_{k,j}$ be the number of observed transitions k to j.

Let $E_k(j)$ be the number of observed cases when j is emitted in state k.

$$P(k \mid j) \approx \frac{A_{k,j}}{\sum_z A_{k,z}}$$

```
Baum-Welsh()
1   Guess some values for $P(k \mid j)$ and $e_k(j)$.
2   repeat
3       for $z \leftarrow 1$ to $n$
4       do
5           Add to $A_{k,j}$ and $E_k(j)$ based on $x^z$ using $P(k \mid j)$ and $e_k(j)$.
6
7       Estimate new $P(k \mid j) \approx \frac{A_{k,j}}{\sum_z A_{k,z}}$
8       Estimate new $e_k(j) \approx \frac{E_k(j)}{\sum_z E_k(z)}$
9
10  until
11
```

Figure 3.1: The Baum-Welsh Algorithm. Given examples strings, $x^i \quad i \in 1, 2, ...n$, and a HMM topology estimate the parameters for the model.

$$e_k(j) \approx \frac{E_k(j)}{\sum_z E_k(z)}$$

We can even bias the estimations by initializing the counts to ratios that we believe are in proportion to the true probabilities. This also makes all counts nonzero so we can avoid division by zero should the sample not contain an emission of a specific state or any transitions from a specific state.

3.5.2 ...if we don't know π

If π is not known then the problem becomes one of optimizing a likelihood. We follow this plan called the **Baum-Welch algorithm**:

We know that an estimation can be made for $A_{k,j}$ by

$$A_{k,j} = \sum_{x \in \text{trainingdata}} \sum_{\pi} P(x, \pi, \pi_i = k, \pi_{i+1} = j)$$

$$P(x, \pi, \pi_i = k, \pi_{i+1} = j) = \sum_{\pi} P(x, \pi, \pi_i = k, \pi_{i+1} = j) = f_k(i) e_j(x_{i+1}) P(j \mid k) b_k(i+1)$$

Using Bayes' Rule:

$$P(\pi, \pi_i = k, \pi_{i+1} = j | x) = \frac{P(x, \pi, \pi_i = k, \pi_{i+1} = j)}{P(x)} = \frac{f_k(i) e_j(x_{i+1}) P(j \mid k) b_k(i+1)}{P(x)}$$

Therefore we can use this as an estimator for $A_{k,j}$:

$$\begin{aligned} A_{k,j} &\approx \sum_z \sum_i P(\pi, \pi_i = k, \pi_{i+1} = j | x^z) \quad \text{where} \quad x^z \quad \text{is the } z^{\text{th}} \text{ example sequence} \\ &= \sum_z \sum_i \frac{f_k(i) e_j(x^z_{i+1}) P(j \mid k) b_k(i+1)}{P(x^z)} \\ &= \sum_z \frac{1}{P(x^z)} \sum_i f_k(i) e_j(x^z_{i+1}) P(j \mid k) b_k(i+1) \end{aligned}$$

which is equation 3.20 in the book. Similarly:

$$E_k(b) \approx \sum_z \frac{1}{P(x^z)} \sum_{i, x^z_k = b} f^z_k(i) b^z_k(i)$$

which is equation 3.21 in the book.

3.6 Mathematical Stability of Dynamic Programming

Doing arithmatic in log space. Assume $a = \log(a')$ where a' is the probabilities we are actually calculating and similarly for b. For multiply it is easy: $\log(e^a e^b) = a + b$. For addition it is a little bit more complicated.

Assume $b > a$:

$$\begin{aligned} \ln(e^a + e^b) &= \ln(e^a + e^{a+(b-a)}) \\ &= \ln(e^a + e^a e^{b-a}) \\ &= \ln(e^a(1 + e^{b-a})) \\ &= \ln(e^a) + \ln(1 + e^{b-a}) \\ &= a + \ln(1 + e^{b-a}) \end{aligned}$$

If $b \gg a$ then $a + \ln(1 + e^{b-a}) \approx a + \ln(e^{b-a}) = a + b - a = b$. The approximate equality becomes equality when the computer thinks that $1 + e^{b-a} = 1$ which for standard IEEE double precision arithmatic (with a 52 bit mantissa) is about when $b - a \approx \ln(2^{52}) \approx 36.04$. So if $|a - b| > 36.04$ then $\ln(e^a + e^b) = \max(a, b)$.

If $b = a$ then $a + \ln(1 + e^0) = a + \ln(2) \approx a + 0.6931471806$.

A side note is that most computer languages provide access to functions that are numerically tricky to computer such as:

$$\text{expm1} = \exp(x) - 1$$

and

$$\text{log1p} = \ln(1 + x)$$

This is kind of work is where these corner case functions might come in handy.

Chapter 4

Profile HMMs

We want to use a biologist verified multiple sequence alignment of a class of sequences to do a better job of finding similar sequences.

Example: testing if the customer has a dog or not.

You want to create the essense of a dog and test that against the creature brought in rather than do individual comparisons against sample dogs and trying to merge the results. How do we create a computational model of an essense?

4.1 Conscensus Modeling

A **conscensus sequence** is a manufactured sequence from a multiple alignment in which the value of any string position is assigned the most frequently appearing symbol in that column. When there is no clear winner a marking character is generally used such as ".".

4.2 Global Matching with Profile HMM

Build an HMM from repetative structures where each structure is intended to be semi-position specific. This will be profile for a global match called a **profile HMM**.

4.2.1 Linear Model

Let's begin with a simple position dependent model. The straight probability given model M:

[linear model here]

$$P(x \mid M) = \prod_j e_{M_j}(x_i) P(M_j \mid M_{j-1})$$

where j is position of anticipated match (the M_j node) and i is the position in the sequence. Here each node corresponds to each position in the string. So we can assume each transition $M_{j-1} \longrightarrow M_j$ has a probablility of 1.

$$P(x \mid M) = \prod_j e_{M_j}(x_i)$$

log odds ratio against the random model R is now:

$$\frac{P(x \mid M)}{P(x \mid R)} = \log(e_{M_j}(x_i)/q_{x_i})$$

4.2.2 Insertion Model

Here we model the ability to insert many copies of a state with an emission distribution for the insert state.

[insertion model here]

4.2.3 Deletion Model

Here we add a way to skip match nodes.

[full deletion model here]

4.3 Full Viterbi Equations

The Viterbi Equations allow us to compute $P(x, \pi \mid M)$ We often do this so we can really compute:

$$\frac{P(x, \pi \mid M)}{P(x, \pi \mid R)}$$

Here are the equations that make that possible where M are the match nodes, I are the insert nodes, and D the delete nodes. $V_j^M(i)$ is the log odds of the best path matching subsequence $x_1, x_2, x_3, ...x_i$ and emitting x_i and ending in state M_j. Similarly for $V_j^I(i)$ and $V_j^D(i)$ except that D nodes don't emit a symbol. NOTE: that often there is no transition between delete and insert nodes and so those parts of the equation can be ignored in those cases.

$$V_j^M(i) = \log \frac{e_{M_j}(x_i)}{q_{x_i}} + \max \begin{cases} V_{j-1}^M(i-1) + \log P(M_j \mid M_{j-1}) \\ V_{j-1}^I(i-1) + \log P(M_j \mid I_{j-1}) \\ V_{j-1}^D(i-1) + \log P(M_j \mid D_{j-1}) \end{cases}$$

$$V_j^I(i) = \log \frac{e_{I_j}(x_i)}{q_{x_i}} + \max \begin{cases} V_j^M(i-1) + \log P(I_j \mid M_j) \\ V_j^I(i-1) + \log P(I_j \mid I_j) \\ V_j^D(i-1) + \log P(I_j \mid D_j) \end{cases}$$

$$V_j^D(i) = \text{zero} + \max \begin{cases} V_{j-1}^M(i) + \log P(D_j \mid M_{j-1}) \\ V_{j-1}^I(i) + \log P(D_j \mid I_{j-1}) \\ V_{j-1}^D(i) + \log P(D_j \mid D_{j-1}) \end{cases}$$

The following are a product form of the forward equations (these are not the nice log forms we see in the book but just provided to make a point about why log doesn't always make our lives easy.)

$$F_j^M(i) = \left(\frac{e_{M_j}(x_i)}{q_{x_i}}\right) \quad [F_{j-1}^M(i-1)P(M_j \mid M_{j-1}) + F_{j-1}^I(i-1)P(M_j \mid I_{j-1}) + F_{j-1}^D(i-1)P(M_j \mid D_{j-1})]$$

$$F_j^I(i) = \left(\frac{e_{I_j}(x_i)}{q_{x_i}}\right) \quad [F_j^M(i-1)P(I_j \mid M_j) + F_j^I(i-1)P(I_j \mid I_j) + F_j^D(i-1)P(I_j \mid D_j)]$$

$$F_j^D(i) = [F_{j-1}^M(i)P(D_j \mid M_{j-1}) + F_{j-1}^I(i)P(D_j \mid I_{j-1}) + F_{j-1}^D(i)P(D_j \mid D_{j-1})]$$

Taking the logs of these forward equations to produce the log odds ratio is not as easy as for the Viterbi equations, since Viterbi equations are strictly product and max functions. Here we have addition and that imposses a problem. For example:

$$F_j^M(i) = \log\left(\frac{e_{M_j}(x_i)}{q_{x_i}}\right) + \underbrace{\log[F_{j-1}^M(i-1)P(M_j \mid M_{j-1}) + F_{j-1}^I(i-1)P(M_j \mid I_{j-1}) + F_{j-1}^D(i-1)P(M_j \mid D_{j-1})]}_{\text{OOPS!}}$$

4.4 Parameter Estimation

What we want to find is given a **topology** of a model, that is an set of nodes and interconnections, we would like to do **parameter estimation** of the transition probabilities and emission probabilities from a vetted multiple sequence alignment that represents exemplars from a class of interest. The sequences in the multisequence alignment would be the **training set** for deriving the parameters and hence completing the HMM specification for the class.

Let $c_{j,a}$ be the observed frequency in training set of character a at position j in the alignment. We can estimate

$$e_{M_j}(a) \approx \frac{c_{j,a}}{\sum_z c_{j,z}}$$

It is not likely that an amino acid is totally not allowed in any one position and yet for most samples not all symbols will be found in every column.

4.4.1 Pseudocounts

We could initialize all $c_{j,a} = 1$ for all j and a. This is like assuming an even initial distribution. A better idea is to add a count proportional to the expected overall frequency of a symbol:

$$e_{M_j}(a) \approx \frac{c_{j,a} + Aq_a}{\sum_z (c_{j,z} + Aq_z)} = \frac{c_{j,a} + Aq_a}{\sum_z c_{j,z} + A} = \frac{c_{j,a} + \alpha_a}{\sum_z c_{j,z} + A}$$

where $\alpha_a = Aq_a$ may be thought of as a prior for symbol a.

4.4.2 Multinomials and Dirichlet

Let c_j be the observed distribution of symbols at position j. Let α^i be one of K sets of pseudopriors. It may be that we can weight the the pseudocount models by the expectation of the pseudopriors given the data, c_j like this:

$$e_{M_j}(a) \approx \sum_y P(y \,|\, c_j) \left(\frac{c_{j,a} + \alpha_a^y}{\sum_z c_{j,z} + \alpha_z^y} \right)$$

this, in a sense, gives each column of the alignment a chance to emphsize a different set of pseudocounts. The difficulty is now in choosing these **mixture coefficients** $P(y \,|\, c_j)$. The book suggests first computing $P(c_j \,|\, y)$ and using a prior of p_y with Bayes' Rule. This they claim this is a direct analogy with the proportional counts above. And that both can be considered to be transformations via Bayes' Rule from a Dirichlet distribution.

DIRICHLET DISTRIBUTION

Factoid: Like Hercule Porot, Dirichlet is not French, he is Belgian. He was a great mathematician in number theory and proved Fermat's Last Theorem for $n = 5$.
A **multinomial distribution** for K different possible outcomes:

$$P(n \,|\, \theta) = \frac{(\sum_z n_z)!}{\prod_z n_z!} \prod_{i=1}^{K} \theta_i^{n_i}$$

where n and θ are vectors of size K. n_i is the number of observed occurrences of outcome i and θ_i is the probability of outcome i with $\sum_z \theta_z = 1$. The multinomial distribution lets us answer questions like what are the odds we see a certain set of outcomes given a set of odds for those outcomes.
The **Dirichlet distribution** is analogous in form:

$$\mathcal{D}(\theta \,|\, \alpha) = \frac{\Gamma(\sum_z \alpha_z)}{\prod_z \Gamma(\alpha_z)} \prod_{z=1}^{K} \theta_z^{\alpha_z - 1}$$

The mean of the various outcome k is the normalized value of α_k. The distribution is this form is useful for modeling a distribution of parameter sets θ based on a set of parameters α. So the result is a distribution on a parameter set.

4.4.3 Substitution Matrix Mixtures

This heuristic uses the substitution matrix S to compute conditional probabilities $P(b \,|\, a)$.

$$\lambda s(a,b) = \log\left(\frac{P(a,b)}{q_a q_b}\right) = \log\left(\frac{P(b \,|\, a)P(a)}{P(a)P(b)}\right) = \log\left(\frac{P(b \,|\, a)}{P(b)}\right) =$$

therefore

$$P(b \,|\, a) = P(b)e^{\lambda s(a,b)} = p_b e^{\lambda s(a,b)}$$

where λ is the λ of the substitution matrix.

If you start with estimate of the true distribution of a in column j

$$q_{j,a} = \frac{c_{j,a}}{\sum_z c_{j,z}}$$

then we apply this constraining calculation:

$$\alpha_{j,a} = A \sum_z q_{j,z} P(a|z)$$

rather than Aq_a which we used before. Then we can use the $\alpha_{j,a}$ as before:

$$e_{M_j}(a) \approx \frac{c_{j,a} + \alpha_{j,a}}{\sum_z c_{j,z} + \alpha_{j,z}}$$

In this case we have combined information found in the substitution matrix and general symbol distribution, q, by using $P(a|b)$ and information from indiviual columns to customize the $e_{M_j}(a)$ help customize to a column. The constant A lets a balance these two forces of choice. An excellent choice for A is thought to be $A = 5R_j$ where R_j is the number of different symbols in column j.

Compare that with the previous pseudocount formulation:

$$e_{M_j}(a) \approx \frac{c_{j,a} + Aq_a}{\sum_z (c_{j,z} + Aq_z)} = \frac{c_{j,a} + Aq_a}{\sum_z c_{j,z} + A} = \frac{c_{j,a} + \alpha_a}{\sum_z c_{j,z} + A}$$

Here general symbol distribution q is used but substitution information is missing.

4.4.4 Estimation from Ancestor

Assume you have $P_t(b \,|\, a)$ where t is some time measure. We have an estimate for this from say Blossum or PAM matrices (given the score and the λ we can get the probability see above.)

What is $P_t(y_i = a \,|\, X^1, X^2, ..., X^k)$ where Y is the hypothetical ancestor and the X are the k samples of the family.

$$P_t(y_i = a \,|\, X^1, X^2 ...) = \frac{q_a \prod_z P_t(x_j^z \,|\, a)}{\sum_{a'} q_{a'} \prod_z P_t(x_j^z \,|\, a')}$$

Now summing over all possible ancestors and assuming a time t since when we saw that ancestor:

$$e_{M_j}(a) = \sum_z P_t(a \,|\, z) P(y_i = z \,|\, X_1, X_2 ...)$$

Consider again our K training sequences X^k AND their alignment, so that in column j we find symbol x_j^k or a gap. Imagine that x's are derived via evolution from some ancestral sequence y. Let's begin by computing

$$P(y_j = a \,|\, \text{aligned}(x)) = \frac{q_a \prod_k P(s_j^k \,|\, a)}{\sum_z q_z \prod_k P(s_j^k \,|\, z)}$$

So if we can guess the general symbol distribution for y which we denote q, then we have a probablity of the ancestor amino acid. (We'll see this again when we do phylogenetics).

Now we can estimate the emissions probabilities

$$e_{M_j}(a) \approx \sum_z P(a \,|\, z) P(y_j = z \,|\, \text{aligned}(x))$$

Note that $P(a \,|\, z)$ comes from the substitution matrix, a global measure, and $P(y_j = z \,|\, \text{aligned}(x))$ from common ancestor inference using substitution matrix and general symbol distribution assumed for the ancestor.

Part of our assumption is that we have strongly conserved regions in the alignment. That means we should be using different substitution matrices for each column sort of S_j for column j. The book points to some references for how to vary the substitution matrix per column.

4.4.5 Model Construction

Chapter 5

Phylogenetics

So we have 3+ billion years of evolution and all we can see, for the most part, are today's DNA sequences and a few bones. Bummer. We suspect that all creatures come from primordial creatures in a **tree of life**. This inheritance forms a true tree like a family tree for a human but for all life on Earth. **Problem 1:** We would like to recover a good guess for an evolutionary tree given a set of creatures for which we have DNA.

The process of building a tree could be done for a whole genome but generally is done for only a gene or part of gene. Since it is easier to do, we have genes, but perhaps not whole genomes and other reasons. **Problem 2:** We would like to recover the evolutionary history in tree form of a set of genes. The tree we create for a set of organisms/genes is called a **phylogenetic tree** and the relationships between the organisms/genes is a **phylogeny**. And the study of making these trees is called **phylogenetics**.

In the rest of these notes I will try to switch to using the term species in the phylogenetic context to mean species or sequences or genes.

A word of caution: just because we can compute a tree doesn't make it the right tree. If there is one thing we can be sure about it is that over the vast stretch of history events of incredibly low probability happen and can influence the outcome of our analysis today. (see the Ray Bradbury short story "Sound of Thunder")

Let's look a the history of some genes:

[insert picture here]

Homologues are genes that share a common evolutionary history. **Orthologues** are genes are homologues that retain their function through evolutionary history. **Paralogues** are genes that have diverge not through speciation but through gene duplication. A BLAST search will detect similar sequences. We often assume these are homologues but they may not be because they don't share a evolutionary ancestor. If they are homologues they might be paralogues relative to one another. One may retain a function and be much more conserved. The other may have drifted from its function and be evolving at a much higher rate giving the impression it is much older than it is.

5.1 Trees in General

Since we can only look at the DNA of today, we only know the "distance" between the sampled sequences. We can only guess what the root is of the evolutionary tree implied by the sequences. We have the option of using techniques to root the tree or leave it unrooted.

Leaves are **nodes** with only one **edge**. Nodes that are not leaves are called **internal nodes**. The total shape or arrangement of nodes and edges is called the **tree topology**. The leaves represent the sequences, species, or genes.

First consider an unrooted tree. It has the property that every internal node has three unordered edges attached to it. With this definition an unrooted tree of three leaves can be labeled with the three species in any order and be considered the same tree.

- There is only one labeled unrooted tree of 2 leaves. It has 1 edge.
- There is only one labeled unrooted tree of 3 leaves. It has 3 edges.
- There are 3 labeled unrooted trees of 4 leaves. If we begin with a tree of 3 leaves labeled A, B, and C. The 4 leaf trees can be generated by taking the unrooted tree of 3 leaves and allowing a branch to come off of either the edge to A, B, or C. Giving 3 unique trees. This operation adds 2 edges. These trees have 5 edges.
- By induction if we have a unrooted tree with n leaves there are $2n - 3$ edges.

The construction above allows us to specify any tree by building it from 3 leaves up. Since at each step we have a 2 more choices of edges than at the previous step a recursive formula can be set up where there are

$$\prod_{k=2}^{n-1}(2k-3)$$

n leaf labeled trees. A table and a comparison with $n!$ is provided in Table 5.1. The table shows that there is an amazing explosion of trees even if you have only a small number of leaves. In fact:

$$\begin{aligned}\prod_{k=2}^{n-1}(2k-3) &= 1 \cdot 3 \cdot 5 \cdot \ldots \cdot 2n-5 \\ &< (1*2) \cdot (2*2) \cdot (3*2) \cdot \ldots \cdot ((n-2)*2) \\ &< (n-2)!2^{(n-2)}\end{aligned}$$

$$
\begin{aligned}
\prod_{k=2}^{n-1}(2k-3) &= 1 \cdot 3 \cdot 5 \cdot \ldots \cdot 2n-5 \\
&> 1 \cdot (1*2) \cdot (2*2) \cdot (3*2) \cdot \ldots \cdot ((n-3)*2) \\
&> (n-3)!2^{(n-3)}
\end{aligned}
$$

Adding a root requires only adding a fake species called root. More about this later. Therefore the number of labeled unrooted and rooted trees grow exponentially.

n	num unrooted trees	n!	trees/n!
2	1	2	0.50000
3	1	6	0.16667
4	3	24	0.12500
5	15	120	0.12500
6	105	720	0.14583
7	945	5040	0.18750
8	10395	40320	0.25781
9	135135	362880	0.37240
10	2027025	3628800	0.55859
11	34459425	39916800	0.86328
12	654729075	479001600	1.36686
13	13749310575	6227020800	2.20801
14	316234143225	87178291200	3.62744
15	7905853580625	1307674368000	6.04574
16	213458046676875	20922789888000	10.2022
17	6190283353629375	355687428096000	17.4037
18	191898783962510625	6402373705728000	29.9731
19	6332659870762850625	121645100408832000	52.0585
20	221643095476699771875	2432902008176640000	91.1024

Table 5.1: A Comparison of the Number of Unrooted Trees

A second property of a phylogenetic tree is that the edges will be labeled with a useful length. So the task ahead is not only to select the right tree but the right lengths.

5.1.1 Newick Format

An unrooted tree is hierarchical grouping of leaves with an edge associated with each node both leaves and internal nodes. Let each leaf be a group. Let a pair of parens group a set of groups $g_1, g_2, g_3, \ldots$ as follows: $(g_1 : l_1, g_2 : l_2, g_3 : l_3, \ldots)$ where the l_i is the distance of the group from the rest of the tree. This notation is very general. In an unrooted tree each internal node accept one connects 2 nodes the remaining internal node connects 3 nodes. For example: Assume you have a

Table 5.2: The number of nodes and edges in a tree

	Unrooted	Rooted
Number of leaves	n	n
Number of internal nodes	$n-2$	$n-1$
Number of nodes	$2n-2$	$2n-1$
Number of edges	$2n-3$	$2n-2$

tree with 4 leaves constructed by making a tree of 3 nodes A, B, and C. Then attaching node D on the edge leading to C.

$$(A : a,\ B : b,\ (C : c,\ D : d) : e)$$

The lengths of the edges connecting to each leaf follow the leaf in the list. The length e is between the two internal nodes. Unfortunately this notation is not unique. For example:

$$((A : a,\ B : b) : e,\ C : c,\ D : d)$$

represents the same tree as above. It all depends on where the list is that has three groups in it. Draw them out and see for yourself. How many representations are their for a given unrooted tree? Are there always the same for an n leaf tree?

5.1.2 Tree Space

Since the space of trees is so huge we will have to search some small portion of the space looking for the tree with the best edge length assignment we can find. Ignoring for a moment how we search for good edge lengths given a topology. How do we search tree space? We want to take small steps to move about the tree space assuming that quality of trees is correlated over small steps. That way we can perform **hill climbing** to find local optima. The step function defines a neighborhood and the trees that you can "step" to are called **neighbors**.

Tree Mutation

Tree mutation or the process of moving to a neighboring tree can mean a terrific speed up in the computation of any quality measure of the tree since most the tree doesn't change.

In **nearest neighbor interchange** (**NNI**) we define the idea of neighbor by exchanging subtrees. Exchanging subtrees joined at an interior node doesn't work. Exchanging the subtrees about an interior edge does. For n leaves there are $n-3$ **internal edges** and n **external edges**. If you remove an internal edge the four subtrees around it can be rearranged in 2 new ways. So if you have a tree with n leaves there are $2(n-3)$ neighbors.

In **subtree pruning and regrafting** (**SPR**) we split a tree at an edge giving a tree of n_1 and n_2 leaves. We reattach tree 2 to tree 1 at any remaining edge of which there are $2n_1-3-1$ if we ignore the place we got the tree from. We can also attach tree 1 to tree 2 in $2n_2-4$ ways for a total of $2n - 8$ or $2n - 6$ if edge is an external edge. So with n external and $n - 3$ internal edges there are $4(n - 3)(n - 2)$ neighbors which is quadratic in n.

In **tree bisection and reconnection** (**TBR**) we remove an interior edge and reconnect both halfs by adding a branch between any edge in one tree to any edge in the other. There are at most $(2n_1 - 3)(2n_2 - 3) - 1$ neighbors.

In **windowing** we take a connected subsection of the tree and look at all rearrangements in just that section.

5.1.3 Tree Construction

How do we get the trees in the first place that we can then mutate?

Sequential Addition

Add a species at a time in the same way we counted the trees. But this time we keep only the best tree after having made each addition. What if a tree ties with another in score? How many do we keep?

What order do we add species? Perhaps we should add the species in the order of the most reliable data. This way the most reliable tree is constructed for the less reliable data. Another approach may be to include the species that are closest matches first.

Star Decomposition

Start with a **star graph** which is a graph with one interior node and all leaves attached directly to that node. Add an edge at a time splitting the graph first into two joined star graphs and so on. This is not a unique process to reach any one final graph.

5.1.4 Comments on Construction and Mutation

Both construction and mutation involve a way to search limited amounts of the search space. Nothing guarantees that we have landed within a hill climb of the globally optimum value. In both cases we make choices between widening the search to cover more space and speed of computation. There is a rich and deep collection of literature on how to do this but no universal answers.

5.2 Edge Length: Distance Matrix Methods

This class of algorithms uses only pairwise distances to align trees. This leaves out any higher order interactions so one might think it would not work well, but on the contrary, they seem to do quite well.

The basic idea is we compute a table of distances $D_{i,j}$ which is a distance measure between sequence i and j. We now want to find the tree that "best fits" this data.

5.2.1 Least Squares Methods

If $d_{i,j}$ is the distances in our "optimum" tree then for Least Squares Methods we want to minimize an equation like:

$$Q = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j}(D_{i,j} - d_{i,j})^2$$

where $w_{i,j}$ is a weight such as 1, $1/D_{i,j}$, or $1/D_{i,j}^2$ but

$$d_{i,j} = \sum_{k} \mathrm{inpath}_{i,j}(k) v_k$$

now

$$Q = \sum_{i=1}^{n} \sum_{j \neq i} w_{i,j}(D_{i,j} - \sum_{k} \mathrm{inpath}_{i,j}(k) v_k)^2$$

Inflection points in this curve can be found by taking the derivative of both sides.

$$\frac{dQ}{dv_k} = -2 \sum_{i=1}^{n} \sum_{j \neq i} w_{i,j}(D_{i,j} - \sum_{k} \mathrm{inpath}_{i,j}(k) v_k) = 0$$

hmmm. something funny about this derivative. Fix later.

[More about matrix approach later.]

5.2.2 Cluster Methods

Assume we have a distance between individual species $d_{i,j}$. In this approach we are given a matrix of these distances and we want to **cluster** the species based on this.

A tree is said to **satisfy a molecular clock** if the tree is rooted and the height from any species to the root is the same. This is a very powerful restriction and these trees are called **ultrametric**. Distances are ultrametric if for all triplets i, j, k either:

$$d_{i,j} = d_{j,k} = d_{k,i}$$

or

$$d_{i,j} = d_{j,k} > d_{k,i} \text{ for some arrangement of } i, j, \text{ and } k$$

Another property of trees is that they can be **additive**. Trees are additive if the distance between i and j is the sum of the edge lengths traversed in going from i to j. Since it is a tree there is only one way to get from any i to j.

UPGMA

UPGMA is Unweighted Pair Group Method with Arithmetic Mean and is a method that clusters from nearest related pair and moves up the tree. **It assumes the tree will satisfy a molecular clock.**

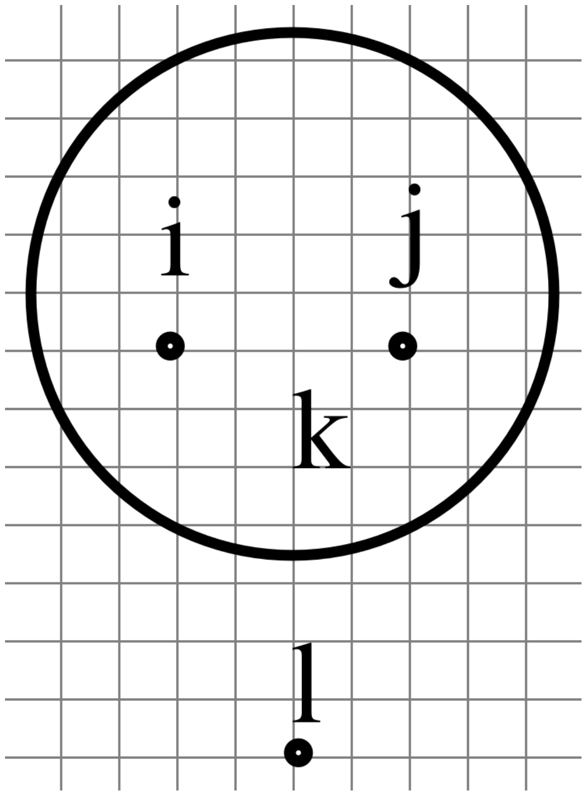
We can talk about a cluster of of species as being a set of species. In our algorithm two clusters C_i and C_j can have defined a distance between them as the average of all the distances between the species in the sets.

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

To merge two clusters, $C_i \cup C_j \rightarrow C_k$, we can derive the following formula for the set of distances from other clusters to form C_k. Consider the distance to some other leaf node l:

$$\begin{aligned}
d_{k,l} &= \frac{d_{i,l}|C_i| + d_{j,l}|C_j|}{|C_i| + |C_j|} \\
&= \frac{\frac{1}{|C_l|} \sum_{p \in C_i, q \in C_l} d_{p,q} + \frac{1}{|C_l|} \sum_{p \in C_j, q \in C_l} d_{p,q}}{|C_i| + |C_j|} \\
&= \frac{\frac{1}{|C_l|} \left(\sum_{p \in C_i, q \in C_l} d_{p,q} + \sum_{p \in C_j, q \in C_l} d_{p,q} \right)}{|C_k|} \\
&= \frac{1}{|C_k||C_l|} \sum_{p \in C_k, q \in C_l} d_{p,q} \\
&= \text{ distance between } k \text{ and } l
\end{aligned}$$

The UPGMA Algorithm



1. Let the set of clusters be called L and initially $i \to C_i \; \forall i$ that is $|C_i| = 1$ and $L = C_1, C_2, \ldots C_N$.

2. $d_{i,j}$ is the distance from the initial distance matrix.

3. For each C_i and give it an initial height: $h_i = 0$.

4. Find $(i,j) = \underset{C_i, C_j \in L}{\operatorname{argmin}} d_{i,j}$

5. Merge $C_i \cup C_j \to C_k$ where k is a new cluster number.

6. $L \leftarrow L - C_i - C_j$

7. Compute $d_{k,z} \forall z \in L$ as described above

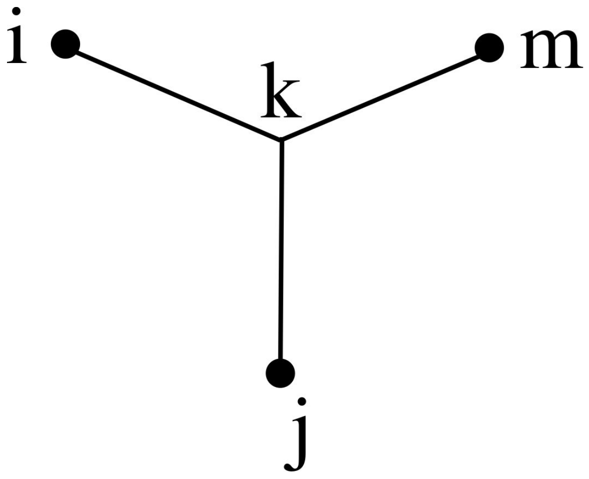
8. Define height $h_k = d_{i,j}/2$ where h_k is the height of node that is the ancestor to all in C_k. When drawing the tree h_k is the height above the baseline (where all the leaves are).

9. $L \leftarrow L \cup C_k$

10. While $|L| > 1$ go to step 4

Neighbor Joining

Neighbor-joining replaces two clusters with one cluster just as UPGMA but **it works with trees where the molecular clock is not constant**, but it requires that the trees be additive. This algorithm **creates an unrooted tree** since the lack of molecular clock eliminates the idea of oldest ancestor node.



First note that if k is first common node on the paths between species i, and m and the path between j and m (that is it is the fork in the road between i and j from m) then

$$d_{k,m} = \frac{1}{2}(d_{i,m} + d_{j,m} - d_{i,j})$$

and

$$d_{i,k} = \frac{1}{2}(d_{i,m} - d_{j,m} + d_{i,j})$$

In the Neighbor-joining Algorithm two distances are used. $d_{i,j}$ is the estimated distance between i and j and $D_{i,j}$ is the distance used in the algorithm for clustering. This second distance is computed from $d_{i,j}$ and so must be computed at each iteration. This slows the algorithm a bit to as slow as $\mathcal{O}(n^3)$.

The Neighbor-joining Algorithm

Given a distance matrix d compute an unrooted tree topology complete with edge lengths that tries to preserve the additive property: $d_{i,m} + d_{j,m} - d_{i,j} = 2d_{k,m}$ where k is the first node on both routes from i and j to m.

1. Let the set of clusters be called L and initially $i \to C_i$ $\forall i$ that is $|C_i| = 1$ and $L = C_1, C_2, \ldots C_N$.

2. $d_{i,j}$ is the distance from the initial distance matrix.

3. Compute "normalized distance matrix" $D_{i,j}$ for all i, j such that

$$D_{i,j} = d_{i,j} - (r_i + r_j) \text{ where } r_i = \frac{1}{|L| - 2} \sum_{z \in L} d_{i,z}$$

   This subtracts the average distance to all other nodes than the pair involved. Note: this is **not** where we use the distance identity.

4. Use normalized distance to find $(i, j) = \operatorname*{argmin}_{C_i, C_j \in L} D_{i,j}$

5. Merge $C_i \cup C_j \rightarrow C_k$ where k is a new cluster number.

6. Mark old clusters as used so that effectively: $L \leftarrow L - C_i - C_j$

7. Compute a new unnormalized distance matrix including the new cluster k and excluding i, j.

$$d_{k,z} = d_{z,k} = \frac{1}{2}(d_{i,z} + d_{j,z} - d_{i,j}) \text{ for all } z \in L$$

   This uses the additivity of the distances to compute the distance to the new cluster from each other node.

8. Compute the length of the edges from k to i and j. Even though C_k has assumed the role of both C_i and C_j you still need the edge length to i and j from k in order to "draw" the tree.

$$edge_{i,k} = \frac{1}{2}(d_{i,j} + r_i - r_j), \qquad edge_{j,k} = \frac{1}{2}(d_{i,j} + r_j - r_i)$$

9. Define height $h_k = d_{i,j}/2$ where h_k is the height of node that is the ancestor to all in C_k. When drawing the tree h_k is the height above the baseline (where all the leaves are).

10. $L \leftarrow L \cup C_k$

11. While there is more than two clusters left go to step 3

12. Finally, join the remaining two clusters with:

$$edge_{j,k} = d_{i,j}$$

Implementation Notes

Consider this part of the computation:

$$D_{i,j} = d_{i,j} - (r_i + r_j) \text{ where } r_i = \frac{1}{|L| - 2} \sum_{z \in L} d_{i,z}$$

The values of r_z can be computed once each time we want to compute matrix D. This saves a vast amount of time. Furthermore, since $D_{i,j}$ is only used to find the argmin of $D_{i,j}$ we actually don't have to save array D; we only need to find the argmin of it. So first compute all the r and then combine the argmin step with the computation of $D_{i,j}$.

Comparison of the Approaches

Neighbor-joining is slower but is solves a problem illustrated as follows:

[more, picture]

Also rooting of a tree can be a problem with the NJ algorithm. So adding an **outgroup**, a species guaranteed to have evolutionarily diverged first is a common approach. e.g. If you want to build a tree for a collection of mammals you might want to add a reptile as an outgroup. It should end up at the root of the tree.

5.3 Recursive Tree Evaluation Algorithms

In these algorithms we will propose a topology, T and evaluate its cost: eval(T) and then use a minimization scheme to find the lowest cost (best) topology. For the next few algorithms it is important to note that we will assume we already have an alignment. This will in most cases let us treat each column of the alignment separately.

The Search

1. propose a tree T
2. evaluate the cost of T retaining best seen so far
3. go back to 1.

5.3.1 Parsimony

Parsimony finds the tree that explains the data with the minimum number of substitutions.

See example in book page 173 under section 7.4. Let's see how we get an evaluation for a given topology. For n leaves we will assume a bifurcating tree with hypothetical values at $n-1$ internal nodes. We will construct these values and the cost from leaves up by using recursion starting at the root and going down.

Notationally: at each leaf k there is a sequence x^k with the character at site u denoted x^k_u. Let's take one site at a time so the value of u is assumed in a sense to be global. Let's focus on what we do at this one site. $S^k(a)$ is the minimum cost of assigning a to node k. For a leaf $S^k(a) = 0$ if $a = x^k_u$, that is, it is no cost. For a leaf $S^k(a) = \infty$ if $a \neq x^k_u$ since that is simply not going to happen. Another way to treat this is S^k is a vector indexed by all possible values in the sequences:

$$\{S^k(c_1), S^k(c_2), S^k(c_3), \dots\} \text{ for characters } c_i$$

Finally, let $S(a, b)$ the cost of substituting a for b in the evolutionary tree.

Let α be the alphabet of the sequence. Then what we want to compute is $\min_{a \in \alpha} S^{\text{root}}(a)$.

```
Cost of Tree at Site $u$ via Weighted Parsimony
This computes $S^T(a)$ for all $a$ given $T$
procedure cost($T$, $u$) :
    if $T$ is a leaf :
        for $a \in \alpha$ :
            if $a = x_u^k$ :
                $S^T(a) = 0$
            else :
                $S^T(a) = \infty$
    else :
        for $a \in \alpha$ :
            cost(left($T$), u)
            cost(right($T$), u)
            $S^T(a) = \min_{b \in \alpha} \left( S^{\text{left}(T)}(b) + S(a,b) \right) + \min_{c \in \alpha} \left( S^{\text{right}(T)}(c) + S(a,c) \right)$
            $x_u^{\text{left}(T)} = b$
            $x_u^{\text{right}(T)} = c$
    return $S^T$
```

with the condition that for the root:

$$\min_{a \in \alpha} S^{\text{root}}(a)$$

and

$$x_u^{\text{root}} = a$$

5.4 Traditional Parsimony

This is the algorithm of Fitch.

```
procedure cost($T$) :
    if $T$ is a leaf :
        if $a = x_u^k$ :
            $R_k = a$
            $cost = 0$
    else :
        $cost = cost(left(T)) + cost(right(T))$
        $R_T = R_{left(T)} \cap R_{right(T)}$
        if $R_T = \varnothing$ :
            $R_T = R_{left(T)} \cup R_{right(T)}$
            $cost += 1$
    return $cost$
```

5.5 Bootstrapping

If we have done a stochastic search resulting in a guess for a best tree and we want to in someway reassure ourselves that we have the right tree then we might try bootstrapping. **Bootstrapping** is the

Given a set of n points $\widehat{F} = \{x_1, x_2, \ldots, x_n\}$ that are drawn **independently** from a distribution $F(\theta)$. We might use this sample to estimate θ.

Even if the distribution F is unknown but n is large enough we can consider our sample, $\widehat{F}$ to "stand-in" for F and the variation we see in sampling from $\widehat{F}$ should be typical of the variation we see in any large sample from F. So we calculate $\widehat{\theta}$ based on $\widehat{F}$. We wanted to know the variation we see in $\widehat{\theta}$ based on creating a new n point dataset from F. If we go back to F and draw a new sample of points we could get more information however, we may not be able to in that. For example, we might not be able to fly back to New Guinea and collect more toads or there may not be more amino acids in the aligned sequences. So we do the next best thing, we draw from $\widehat{F}$ **with replacement** letting $\widehat{F}$ stand-in for F. Weird, eh? So we can compute the variation $\widehat{\theta}$ in estimating θ by resampling n samples of $\widehat{F}$.

$\widehat{F}$ is called the **bootstrap replicate**. If we look at the variation in $\widehat{\theta}_i$ computed from many replicates $\widehat{F}_i$ we can understand the variation in $\widehat{\theta}$.

So how is this useful with phylogenies? Given a set of sequences representing species. The reason we can construct a tree is that a column of an aligned sequence from one species to the next is **not** independent. But we often make the assumption of independence from one column to the next. So let's bootstrap the columns and generate new trees!

Assume we have k species with n characters in each sequence. We construct a new set of k species with n characters in each sequence by drawing columns with replacement from the data we already have. The result is not a collection of numeric estimates $\widehat{\theta}_i$ but rather a set of trees. Not as easy to deal with.

5.5.1 Combining Trees

A combined set of trees is called a **consensus tree**. It is hoped that the consensus tree is a summary of the good qualities of all of the trees in the set. There are many kinds of consensus trees we will look at three specific kinds.

For unrooted trees we must first understand the idea of a partition in the tree. If a branch divides a 5 taxa tree into two **groups** of taxa $\{A, B\}$ and $\{C, D, E\}$ then that can be expressed as a **partition**:

$$\{AB|CDE\}$$

A partition consists of a joining of two groups of related nodes in the tree. Of course, every branch divides the tree into two groups of taxa so there is a mapping from branches to partitions. n leaf tree produces $n-1$ **nontrivial partitions** in that each group in the partition has more than one taxon. When we combine trees we may want to preserve certain divisions or partitions

Majority Rule Consensus

In **majority rule consensus** we construct a tree that has every group of nodes that appears in more than 50% of the trees. Suppose we construct such a tree and pick a partition from it say $\{\alpha|\beta\}$. Then since the groups α and β occur in more than 50% of the trees we know there is a tree in our sample that has both α and β in it so they must not conflict so the tree is constructable from the set of all such majority groups.

We can label the internal arcs (nontrivial partitions) by the percentage of trees that have that partition. This is called the **P-value** for the edge.

Adams Consensus

A problem with Majority Rule is that if you add an extra copy of one of the trees in the set to your set it could change the consensus. This is not true of the Adams consensus since it doesn't use voting. It tries to prevent a conflict. A **three taxon statement** is a relationship between three taxa where two are more closely related than the third. For example, a three taxon statement might be expressed: $((A, B), C)$. An Adams consensus tree contains all three taxon statements that are not contradicted by any tree in the set. Felsenstein [?] describes the three taxon constraint as going to all trees in the set and removing all but the three nodes in the three taxon statement. All the trees should now be identical and display the same grouping of closeness. If $((A, B), C)$ is in some of the trees in the set but $((A, C), B)$ is in another tree in the set then a trifurcation is displayed.

5.5.2 Problems with P-values

P-values can have a **multiple tests problem**. If you take look for **bootstrap support** for an edge by selecting edges with a $P > x$ then you run the risk of finding edges that by accident are supported.

P-values are not actually probabilities. If an edge occurs 70% of the time then it might actually have a 95% chance of being true. See Hillis and Bull [?].

5.5.3 Robinson Foulds Distance Metric

The number of branches that are different between the trees. This is done by counting the number of unique paritions generated by the trees. Very sensitive to small changes in tree shape.

5.6 Evolutionary Models

Consider a substitution matrix that is dependent on time t. $S(t)$ for nucleotides is a 4×4 matrix of functions dependent on t.

A property of such matrices is that they are multiplicative:

$$S(t)S(u) = S(t+u)$$

for each element of the matrix this is the same as saying:

$$P(a \,|\, b, t+u) = \sum_z P(a \,|\, z, t) P(z \,|\, b, u)$$

If this is going to work for any t and u then there is a continuity about the probability that it is computable directly from time. The instantaneous probability can't change over time.

Here is the proposed rate matrix used by Jukes-Cantor:

$$R = \begin{pmatrix} -3\alpha & \alpha & \alpha & \alpha \\ \alpha & -3\alpha & \alpha & \alpha \\ \alpha & \alpha & -3\alpha & \alpha \\ \alpha & \alpha & \alpha & -3\alpha \end{pmatrix}$$

Consider the substitution over a small time step ϵ:

$$S(\epsilon) \approx I + R\epsilon$$

then

$$S(t+\epsilon) = S(t)S(\epsilon) \approx S(t)(I + R\epsilon) = S(t) + S(t)R\epsilon$$

This means that

$$\lim_{\epsilon \to 0} \frac{S(t+\epsilon) - S(t)}{\epsilon} = \lim_{\epsilon \to 0} S(t)R$$

From basic calculus this means that

$$S'(t) = S(t)R$$

this suggests a matrix like:

$$S(t) = \begin{pmatrix} g(t) & f(t) & f(t) & f(t) \\ f(t) & g(t) & f(t) & f(t) \\ f(t) & f(t) & g(t) & f(t) \\ f(t) & f(t) & f(t) & g(t) \end{pmatrix}$$

therefore

$$S'(t) = S(t)R = \begin{pmatrix} g(t) & f(t) & f(t) & f(t) \\ f(t) & g(t) & f(t) & f(t) \\ f(t) & f(t) & g(t) & f(t) \\ f(t) & f(t) & f(t) & g(t) \end{pmatrix} \begin{pmatrix} -3\alpha & \alpha & \alpha & \alpha \\ \alpha & -3\alpha & \alpha & \alpha \\ \alpha & \alpha & -3\alpha & \alpha \\ \alpha & \alpha & \alpha & -3\alpha \end{pmatrix}$$

this gives

$$S'(t) = \begin{pmatrix} -3\alpha g(t) + 3\alpha f(t) & -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) \\ -\alpha g(t) + \alpha f(t) & -3\alpha g(t) + 3\alpha f(t) & -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) \\ -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) & -3\alpha g(t) + 3\alpha f(t) & -\alpha g(t) + \alpha f(t) \\ -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) & -\alpha g(t) + \alpha f(t) & -3\alpha g(t) + 3\alpha f(t) \end{pmatrix}$$

this means that

$$g(t)' = -3\alpha g(t) + 3\alpha f(t)$$
$$f(t)' = -\alpha g(t) + \alpha f(t)$$

The solution for this is:

$$g(t) = \frac{1 + 3e^{-4\alpha t}}{4}$$
$$f(t) = \frac{1 - e^{-4\alpha t}}{4}$$

Note

$$\lim_{t\to\infty} g(t) = 1/4 \lim_{t\to\infty} f(t) = 1/4$$

expected number of substitutions over the maximum likelihood distance

f is the fraction of sites that differ. $3\alpha d_{ML} = -3/4 \ln(1-4f/3)$ where $d_{ML} = -1/4\alpha \ln(1-4f/3)$ where d_{ML} is the maximum likelihood distance

Chapter 6

Multiple Sequence Alignment

How do we do a **multiple alignment**. The multiple alignment can improve the quality of participating pairwise alignments and provide a source of data for protein family profiling.

6.1 Scoring Schemes

Multiple alignment has two parts **scoring** and **alignment**. Let's first look at scoring.

1. Clearly from inspecting multiple alignments we must use a position specific scoring scheme of some kind since there is a high degree of defitconservation in some columns i.e. limited variation in the substitution. and low levels of conservation in other columns.

2. Evolutionary history can be very valuable in matching variation between pairs of sequences. Unfortunately this approach is computationally intensive.

Let m be a multiple sequence alignment.
Let m_i be the symbols in column i.
Let $c_{i,a}$ be the count of the number of symbols a in column i.
Let $p_{i,a}$ be the probability of symbols a in column i.

If all of the columns are independent then

$$S(m) = G + \sum_i S(m_i)$$

where S is a scoring function and G is a function to score the gaps. This is usually an affine measure and so not column independent and modeled with G separately. However if the gap penalty is linear then the gap can be treated as an extra symbol e.g. a 21^{st} amino acid.

6.2 Sum of Pairs Scoring

A fair approximation of $S(m_i)$ might be based on summing up a scoring scheme for all pairs of symbols in the list m_i. This is certainly not as perfect as a score that takes in all dependencies between any subset of the symbols but it sure is easier.

$$S(m_i) = \sum_{k<j} S(m_i^k, m_i^j)$$

Gaps can be handled by $S(x, -)$ and $S(-, x)$ or by an affine gap penalty function.

A true multiple alignment score would be more like this three sequence example:

$$\log\left(\frac{P_{abc}}{p_a p_b p_c}\right) \neq \log\left(\frac{P_{ab}}{p_a p_b}\right) + \log\left(\frac{P_{ac}}{p_a p_c}\right) + \log\left(\frac{P_{bc}}{p_b p_c}\right)$$

6.3 Multiple Dimension Dynamic Programming

This generalization suffers from the **curse of dimensionality**. Dynamic programming in N dimensions requires inputs from $2^N - 1$ inputs to each position (street corner) in the matrix. Plus if the sequences being compared are about L symbols, there are about $\mathcal{O}(L^N)$ positions in the matrix. So the algorithm takes $\mathcal{O}(L^N 2^N)$ time to execute. If S also takes about $\mathcal{O}(N)$ to calculate but worse yet if it is stored as a matrix then it takes K^N space. If there are $K = 20$ amino acids and $N = 10$ then that is $10,000,000,000,000$ values in the substitution matrix alone. No one would ever do it this way but it just shows how quickly the number of options explodes. This also shows how difficult it will be to contain the explosion.

6.3.1 Containing the Explosion

If we can create severe bounding of function values we can contain the search space and only consider those areas that might have a chance of producing a minimum score.

Let $S(a)$ be the score for a given multiple sequence alignment a.

$$S(a) = \sum_{k<j} S(a^{kj})$$

where $S(a^{kj})$ is the score for the pairwise alignment of m_k and m_j. Assume we have an optimal alignment for k and j and its score is $S(\widehat{a}^{kj})$. We know these things:

$$S(a) = \sum_{k<j} S(a^{kj})$$

Where $S(a^{kj})$ is the alignment score of the pair in the multiple sequence alignment. This is not known to begin with.

$$S(a^{kj}) \leq S(\widehat{a}^{kj})$$

Where $S(\widehat{a}^{kj})$ is the pairwise alignment score. This is computed by something like dynamic programming for each pair and so must be at least as good as the pair score that has the restriction that it must fit with $N-2$ extra sequences as well as the j^{th} one.

$$\sigma(a) \leq S(a)$$

where $\sigma(a)$ is a lower bound on the multiple sequence alignment score. In particular it is calculated by some cheap and fast multiple sequence alignment algorithm that gives an adequate lower bound.

For any j and k:

$$\sigma(a) \leq \sum_{y<z} S(a^{yz}) \leq S(a^{kj}) - S(\widehat{a}^{kj}) + \sum_{y<z} S(\widehat{a}^{yz})$$

therefore from

$$\sigma(a) \leq S(a^{kj}) - S(\widehat{a}^{kj}) + \sum_{y<z} S(\widehat{a}^{yz})$$

we get:

$$S(a^{kj}) \geq \sigma(a) + S(\widehat{a}^{kj}) - \sum_{y<z} S(\widehat{a}^{yz}) = \beta^{kj}$$

Each β^{kj} is a lower bound on the pairwise alignment we must get for k abd j. The higher these β^{kj} are the smaller the search space.

For each β^{kj} fine a complete set of coordinate pairs, B^{kj} such that the best alignment through each point in B^{kj} is better than β^{kj}. But we know how to compute this in two dimensions by DP using Viterbi equations in both a forward and backward form. This is inexpensive: $\mathcal{O}(L^2)$. In the original N sequence problem the points in N-space, $(i_1, i_2, i_3, \ldots, i_N)$ must contain only pairs from the various set B^{kj}.

The book points to Gupta et al. at this point for details of how that is done.

6.3.2 Details of MSA

- MSA does allow for affine gap penalties.
- A **guide tree** is constructed to associate weights with each pair of sequences m_k, m_j.
- MSA begins with an overestimated lowerbound which may yield no answers and incrementally reduces the lower bound until answers are discovered. It is too expensive to start with an underestimated lowerbound... the algorithm might not finish in our lifetime.

6.4 A^*, Dijkstra's Algorithm and Other Network/Tree Optimizations

There are many useful optimization algorithms like dynamic programming that can be useful in sequence analysis. The classic A^* algorithm is an example of an optimization algorithm that prunes it search to just the areas that stand a chance of containing optimal values. When trying to reduce the vast size of the search space of the optimizations techniques in this family are very useful.

6.5 Progressive Alignment

Start with two sequences aligned. Add another and another progressively. Questions are about order of progressive construction and how to merge. More specifically:

- what order to align the sequences
- whether the order is sequentially adding to a single group or collecting smaller multiple alignments and merging
- how to you merge alignments or simply add the next sequence to an existing alignment

Align most similar sequences first. These are reliable and give a strong signal about what can be expected to be conserved and what cannot in less similar sequences. In order to do this a **guide tree** is constructed that groups the sequences together in order of similarity. These cannot be phylogenetic trees exactly because we need the multiple alignment to do a good phylogenetic tree. But we need a good phylogenetic tree to get a good alignment. It is a chicken and egg problem. There exist methods that do both together.

6.5.1 Feng-Doolittle Progressive Alignment

1. Create a matrix of all pairwise distances using a normalization of the pairwise alignment score as a distance measure
2. Construct a guide tree from the distance matrix using the clustering algorithm of Fitch and Margoliash (we will do this in the next chapter)
3. Start merging sequences by following the groupings dictated by the tree.

A normalized distance measure is used

$$D = -\log S_{\text{eff}} = -\log \frac{S_{\text{obs}} - S_{\text{rand}}}{S_{\text{max}} - S_{\text{rand}}} \Leftrightarrow -\log \frac{\text{observed} - \text{worst}}{\text{best} - \text{worst}}$$

where:

- S_{rand} is the score of aligning two sequences of same length and residue composition
- S_{obs} is the observed score of the align of interest
- S_{best} is the score of the aligning either of the sequences with itself

The guide tree constructed is now used to select the order of merge.

A sequence is merged into a group by finding the highest pairwise scored individual in the group with the sequence to be merged. Then use that pairwise alignment to gap the new sequence and insert into the group.

Similarly, in merging two groups is to find the highest scoring alignment between the pairs where one sequence is in one group and one sequence is in another.

Once a gap has been inserted into a sequence it is never uninserted. This assumes that the pairwise alignment contained the most information and so ungapping to accommodate a less similar sequence would be bad.

6.5.2 Profile Alignments

Progressive alignments that only consider pairwise scores ignore information we wanted to generate and use in multiple sequence alignment in the first place namely knowledge of conserved regions. So position specific information accumulated during the multiple alignment should be used. We can do this by using any position specific scoring matrix or producing an HMM.

In this section **profile** means a set of sequences that contains information gathered as a whole and not strictly gathered as isolated pairwise alignments. This is not to be confused with the HMM profile.

6.5.3 Example: CLUSTALW

CLUSTALW uses a profile-based progressive multiple alignment algorithm much as described: creating a distance matrix, constructing a guide tree and then performing a progressive alignment. There are many enhancements and heuristics:

- Use a weighted sum of pairs scoring for sequence comparison. This allows us to discount biases such as nearly identical sequences.

- Choose substitution matrix so that it most nearly matches the degree of differences in the sequences to be compared e.g. BLOSUM50 vs BLOSUM80.

- Position dependent gap-open penalties. By **gap-open** we mean the beginning of a gap as opposed to in the middle of a series of gaps. **gap-extend** penalties are the costs to extend an already opened gap.

- gap-open penalties are decreased if the position is spanned by a consecutive sequence of 5 or more hydrophilic residues since they are known to be more susceptible to gaps.

- gap penalties of any kind are decreased in areas of low gaps.

- if during multiple sequence alignment the score from an alignment comes in significantly lower than anticipated by pairwise scoring that built the guide tree then do not merge that sequence if a better choice exists.

6.6 Iterative Alignment

A problem with progressive alignment is that once a sequence has been merged into an aligned subset of sequences the alignment is frozen regardless of what new information comes to light.

Local optima only has meaning relative to an operator.

This step is an attempt to get better optimum by local search. This is done by for all sequences unplug a sequence and align it with the rest. Repeat until converged. This is really an example of **stochastic optimization**.

6.7 Multiple Alignment by HMM Profiles

HMM profiles are based on firm probabilistic grounds. Let's begin by assuming we have a HMM model, topology of states and probabilities, and perform a multiple sequence alignment.

This can arise when we have an HMM profile of a family of sequences and we want to expand on that family by aligning a set of new examples.

The most probable path for a new sequence though an HMM model can be found via the Vertirbi algorithm. This associates each residue with a match state or an insert state. A match state maps to a column in the final alignment. Insert states group inserted residues but do not attempt to align them. See Figures 6.4 through 6.6 in the book. This takes the rather severe view that the inserted sequences are unalignable.

Now multiple alignment can be thought of as coming in two parts. First build model, then align as above. To build the model:

- Choose a number of match states for the model you want to build. This essentially says select a topology.
- Estimate initial values for parameters. We discussed several ways to do this.
- Use Baum-Welch to estimate parameters

Baum-Welch is just one kind of optimization that finds good parameters.

If we consider optimization as trying to optimize a function in high dimensional space where each dimension is another parameter to optimize. Then any stochastic optimization algorithm can be applied but some may work better than others.

A popular stochastic optimization algorithm is **simulated annealing**. Figure 6.1 is an outline of the simulated annealing algorithm. Here we are trying to optimize the function f which, in our case, would be the quality of the parameters for the HMM for the sequences we are trying to match. The goodness of the move from the current point to the next randomly chosen point is in Δ. Notice that if Δ is positive the current move is set to the next move because it has a higher value of the function f. If Δ is negative then moving to that point makes f smaller or worse. If always take the better move we will quickly become stuck in a local optimum (if mutate makes small random changes in the parameters). But the current move could still set to the next move for negative Δ with some small probability to allow us to escape the local optimum. The probability is based on how "bad" the Δ is and what the value of temperature T. A high temperature means $\Delta/T \to 0$ so the probability is near 1. A low temperature ($T \to 0, T > 0$) means $\Delta/T \to -\infty$ since $\Delta < 0$ so the probability is near 0.

```
for t=0 to ∞ {
    T = Schedule(t, T)
    if T<lowtemp then return
    next = mutate(current)
    $\Delta$ = f(next)-f(current)
    if $\Delta > 0$ current = next                      # case of positive $\Delta$
    else current = next with probability $e^{\Delta/T}$     # case of negative $\Delta$
}
```

Figure 6.1: Simulated Annealing

A brute force approach mentioned is noise injection which is a lot like changing the mutation function as time passes. There are many variations of stochastic search that can be applied.

Chapter 7

Probabilistic Phylogenetics

$f(x) = P(x|y)$ is the **probability** of x given y. $f(y) = P(x|y)$ is the **likelihood** that y yielded x.

What we want to do is ask the related questions:

$$P(x^\bullet \,|\, T, t_\bullet)$$

which asks what the probability of seeing sequences $x^\bullet$ given tree topology T and edge lengths $t_\bullet$. and

$$P(T, t_\bullet \,|\, x^\bullet)$$

Which is the probability of the tree given the data.

$$\frac{P(H_1 \,|\, D)}{P(H_2 \,|\, D)} = \frac{P(D \,|\, H_1)}{P(D \,|\, H2)} \frac{P(H_1)}{P(H2)}$$

$\frac{P(H_1)}{P(H2)}$ is ratio of the probabilities of our hypotheses. This is our prior information before we observe more data. So we say this is the ratio of **prior probabilities**. $\frac{P(D \,|\, H_1)}{P(D \,|\, H2)}$ gives us information about the data we observed. It asks the question: do each of the hypotheses support the data we see? This provides us a quantitative answer that we can use to modify the prior to get the ratio of **posterior probabilities** $\frac{P(H_1 \,|\, D)}{P(H_2 \,|\, D)}$.

7.1 Maximum Likelihood

Let's assume we can answer the following localized question: $P(x \,|\, y, t)$ which is the probability of seeing an intermediate sequence x given parent sequence y after an edge length or "time" t. If we assume that columns are independent then this localized question becomes $P(a \,|\, b, t)$ which is the probability of seeing character a as the child of character b in the sequence after length t.

The values $P(a \,|\, b, t)$ are a model of the evolution that takes place in the tree. There are many standard models but we'll come back to this.

If we have these values the $P(x \,|\, y, t)$ is simply $\prod_u P(x_u \,|\, y_u, t)$.

For the tree:

we have

$$P(x^1, x^2, x^3, x^4, x^5 | T, t_\bullet) = P(x^1|x^4, t_1)P(x^2|x^4, t_2)P(x^4|x^5, t_4)P(x^3|x^5, t_3)P(x^5)$$

So if we want $P(x^1, x^2, x^3 | T, t_\bullet)$ we will have to:

$$P(x^1, x^2, x^3 | T, t_\bullet) = \sum_{x^4, x^5} P(x^1, x^2, x^3, x^4, x^5 | T, t_\bullet)$$

To find the maximum likelihood tree we will need to do this over all possible T and $t_\bullet$.

There exists a recursive way to solve this. Consider a tree with two leaves:

$$P(x^1, x^2, a \,|\, T, t_1, t_2) = P(x^1 \,|\, a, t_1)P(x^2 \,|\, a, t_2)P(a)$$

therefore,

$$P(x^1, x^2 \,|\, T, t_1, t_2) = \sum_a P(x^1, x^2, a \,|\, T, t_1, t_2) = \sum_a P(x^1 \,|\, a, t_1)P(x^2 \,|\, a, t_2)P(a)$$

To make this recursive:

[needs to be filled in from handwritten notes.]

7.1.1 Felsenstein's Algorithm

Let the leaf notes of the tree be numbered 1 through N. Let the internal nodes of the tree be numbered $N + 1$ through $2N - 1$ with the root as $2N - 1$. Let T_k denote the subtree starting at node k and going down to tree to the leaves. Finally, let t_k be the edge length from node k to its parent node. This is the evolutionary distance. Assume independence of columns of data and denote the columns by u. Consider the tree derived for each column u separately. The likelihood for the whole tree using column u data is:

$$P(T_{2N-1}) = \sum_z P(T_{2N-1} \,|\, z)P(z)$$

where $P(z)$ is the probability seeing nucleotide (residue) z at the ancestral root.

For any nonleaf node k with children i and j:

$$P(T_k \,|\, z) = \sum_{i,j} P(i \,|\, z, t_i) P(T_i|i) P(j \,|\, z, t_j) P(T_j|j)$$

$$P(T_k \,|\, z) = \left(\sum_{i} P(i \,|\, z, t_i) P(T_i|i)\right) \left(\sum_{j} P(j \,|\, z, t_j) P(T_j|j)\right)$$

For any leaf node k:

$$P(T_k \,|\, z) = 1 \;\text{ if }\; z = x_u^k \;\text{ and }\; 0 \;\text{ otherwise}$$

To compute this we need the value of $P(T_k \,|\, z)|_z$. This is a vector in $\mathbb{R}^4$ for each column of the sequence.

Time Reversible

Since Felsenstein Algorithm assumes we are building a rootless tree how is it possible that the algorithm itself refers to a root? It turns out that any node can be the root. To see this assume we have a tree of two leaves X and Y and an internal node W. This miniature tree can stand in for any fork in the rooted tree used in the Felsenstein's Algorithm. Assume that node W has character z We see that:

$$\sum_{z} P(T_W \,|\, z) P(z) = \sum_{z,i,j} P(i \,|\, z, t_i) P(T_i|i) P(j \,|\, z, t_j) P(T_j|j) P(z)$$

Efficiencies

exponentiation precomputing

preserving all the subtree data

leaves are half the number of nodes

making small changes

edge lengths computed by standard gradient methods

You want to compute in log space which requires computing $\log(x+y)$ from $\log(x)$ and $log(y)$. You don't want to do

$$\log(x+y) = \log(e^{\log(x)} + e^{\log(y)})$$

But because the values of $e^{\log(x)}$ and $e^{\log(y)}$ may differ a lot for large areas of the domain the function is easy to compute as simply the larger of the two values. For similar values, tricks can be performed to preserve accuracy and speed computation.

7.1.2 Ambiguities

The Felsenstein algorithm allows us to take various ambiguities into account. Suppose for the sequence at a leaf we get an "N" rather than a specific nucleotide? The "N" means that any nucleotide will work. Then what are the values of $P(L_{\text{leaf}} \,|\, a)$? For example, in $P(L_{\text{leaf}} \,|\, G)$ we are asking the question what is the probability of seeing "N" given a G. The answer is 1 and NOT $\frac{1}{4}$!

Suppose we want to account for some reader error, ϵ at the leaf stage? We can assign $P(L_{\text{leaf}} \,|\, G)$ the values $(\epsilon/3, \epsilon/3, 1-\epsilon, \epsilon/3)$ to the values returned for each of the (A, C, T, G) cases in the leaf node. It answers the question: what is the probability that we saw the nucleotide in the leaf sequence given that the leaf node is "really" G?

7.2 Bayesian

In the Maximum Likelihood section we wanted to compute the likelihood of the observed sequences $x^\bullet$ given a tree topology, T; lengths of edges, $t_\bullet$; and an evolutionary model and its parameters (e.g. Kimura 2 Parameter Model), θ. In the book, the model detail was assumed and we had:

$$P(x^\bullet \,|\, T, t_\bullet)$$

In the Bayesian approach what we really want to compute is:

$$P(T, t_\bullet \,|\, x^\bullet)$$

with some carefully accounted for assumptions. This seems like the more natural question anyway. We can begin by using Bayes law:

$$P(T, t_\bullet \,|\, x^\bullet) = \frac{P(x^\bullet \,|\, T, t_\bullet)P(T, t_\bullet)}{P(x^\bullet)} = \frac{P(x^\bullet \,|\, T, t_\bullet)P(T, t_\bullet)}{\displaystyle\sum_{\tau \in \text{ all}(T, t_\bullet)} P(x^\bullet \,|\, \tau)P(\tau)} \tag{7.1}$$

So we can get the posterior probability $P(T, t_\bullet \,|\, x^\bullet)$ by using the likelihoods and $P(x^\bullet)$ or more specifically by computing the likelihoods $P(x^\bullet \,|\, T, t_\bullet)$ over ALL trees and by using their priors $P(T, t_\bullet)$. The priors is the natural place we want to merge in our understanding of evolutionary model and any other preconceived ideas about the evolution.

So let's back up and rethink this from the beginning of this section. We can think of $P(T, t_\bullet \,|\, x^\bullet)$ as generating a distribution of trees given the fixed data $x^\bullet$. For instance one tree topology and lengths might be twice as likely as another giving a distribution. So we have to ask ourselves why we want to know $P(T, t_\bullet \,|\, x^\bullet)$. It is not to get the precise probability of a specific tree but probably to infer **the expected value of some property of the trees given the data**!. This would could be served by finding the whole distribution and computing for property ψ:

$$E[\psi(\tau)|x^\bullet] = \sum_\tau \psi(\tau)P(\tau \,|\, x^\bullet)$$

It could also be served by finding a large sample of examples from the distribution of trees implied by the data. By the Law of Large Numbers if we take a large number of i.i.d. samples from our distribution $P(\tau \,|\, x^\bullet)$ we can get as close as we want to $E[\psi(\tau)|x^\bullet]$. So if τ_i has twice the probability of τ_j in our distribution then τ_i occurs twice as often in our sample. Crude, but effective.

So, if we sample a large number of times from the distribution $P(\tau \,|\, x^\bullet)$ then

$$E[\psi(\tau)|x^\bullet] \approx \sum_{k=1}^{\text{large}} \psi(\tau_k)$$

So to get the real answer we want, we need to find a way to sample from the distribution $P(\tau \,|\, x^\bullet)$ and decide how many samples is enough. Since the distribution is nearly impossible to actually compute, we will have to find a trick. The key to this will lie in understanding Acceptance and Rejection Sampling.

7.2.1 Acceptance and Rejection Sampling

Suppose you have a **target pdf** $\pi(x) = f(x)/K$ for some constant K, $x \in \mathbb{R}^d$. Let $h(x)$ be a density that can be simulated such that $f(x) \le ch(x) \forall x$ Now we want a random sample from π.

```
Loop {
    Generate $Y$ from $h$
    Generate $u$ from $U(0,1)$
    if ($u \le f(Y)/(ch(Y))$) return Y
}
```

Now Y will be the sampled from the distribution π but it has various degrees of efficiency depending on c.

$$c = max_x f(x)/h(x)$$

is a good choice.

7.2.2 Metropolis-Hastings Algorithm

The next trick is we won't try to generate Y from a distribution h but rather we will run a Markov chain whose stationary distribution is the same as the desired distribution. Then wait until we get close to the stationary distribution before taking samples. So even though the samples are not i.i.d. we can use them because over a LARGE NUMBER OF SAMPLES they have the same distribution as the one we want.

Changes we will make to the Acceptance and Rejection Algorithm: rather than simply generate from a distribution h, we will use a Markov model that we will construct to have certain properties. We want to generate a set whose distribution is the same as the distribution over all of the trees.

```
Loop {
    Generate $Y$ based on $X_t$
    Generate $u$ from $U(0,1)$
    If $(u \leq \alpha(X_t, Y))$ $X_{t+1} = Y$
    Else $X_{t+1} = X_t$
    k++
}
```

Note that when we reject a sample in this case don't ignore it we take $X_{t+1} = X_t$. Lot's of rejection implies lots of same values but we will be summing over these. In the case of trees τ:

$$E[\psi(\tau)|x^\bullet] \approx \sum_{k=m}^{n} \psi(\tau_{t+1})$$

where m marks the end of the **burn-in period** where we were waiting for the Markov model to get close to its stationary distribution and n is large.

Else $\tau_{t+1} = \tau_t$

Where does the **acceptance function** α come from and what are the transition probabilities for our Markov model?

Let's look to the equation for α which is the probability of accepting the new sample Y. It is based on the desired distribution $\pi(x)$ and the Markov model's transition probability $q(X_{t+1}|X_t)$:

$$\alpha(X, Y) = \min\left(1, \frac{\pi(Y)q(X|Y)}{\pi(X)q(Y|X)}\right)$$

$q(X|Y)$ is called the **proposal distribution** and can be absolutely anything (much like $h(x)$)! Well, almost anything... One thing to note is that if $\alpha(X, Y) < 1$ then the quotient in the min function is less than 1 so swapping X and Y makes the quotient greater than 1. Since the min function is then applied: $\alpha(Y, X) = 1$. The above algorithm can now be written:

```
Loop {
    Generate $Y$ based on $q(. \mid X_t)$
    Generate $u$ from $U(0,1)$
    If $(u \leq \alpha(X_t, Y))$ $X_{t+1} = Y$
    Else $X_{t+1} = X_t$
    k++
}
```

We can observe several things about the algorithm:

- $q(X_{t+1}|X_t)$ is the probability of generating X_{t+1} for the **if** statement.
- $\alpha(X_t, X_{t+1})$ is the probability of accepting X_{t+1} given that the previous generated value was X_t.
- $1 - \alpha(X_t, X_{t+1})$ is the probability of passing to the **else** statement if the last two values generated were X_t and X_{t+1}

Therefore we can compute $P(X_{t+1}|X_t)$ for both halves of the **if** statement starting with the probability that X_{t+1} was generated and then accepted:

$$P(X_{t+1}|X_t) = \begin{cases} q(X_{t+1}|X_t)\alpha(X_t, X_{t+1}) & \text{if } \; X_t \neq X_{t+1} \\ q(X_{t+1}|X_t)\alpha(X_t, X_{t+1}) + \sum_Y (1 - q(Y|X_t)\alpha(X_t, Y)) & \text{if } \; X_t = X_{t+1} \end{cases}$$

The resulting transition matrix, also known as **transition kernel**, can be compactly written:

$$P(X_{t+1}|X_t) = q(X_{t+1}|X_t)\alpha(X_t, X_{t+1}) + Test(X_{t+1} == X_t)\sum_Y (1 - q(Y|X_t)\alpha(X_t, Y))$$

where $Test(X_{t+1} == X_t)$ returns 1 if the two values are equal and 0 otherwise.

What about the stationary distribution? From the definition of $\alpha(.,.)$ let's assume the case the fraction is less than 1, i.e. $\alpha(X, Y) < 1$, then we know:

$$\begin{aligned} \alpha(X,Y)\pi(X)q(Y|X) &= \pi(Y)q(X|Y) \;\text{ and }\; \alpha(Y,X) = 1 \\ \alpha(X,Y)\pi(X)q(Y|X) &= \pi(Y)q(X|Y)\alpha(Y,X) \\ \pi(X)\alpha(X,Y)q(Y|X) &= \pi(Y)\alpha(Y,X)q(X|Y) \\ \pi(X)P(Y|X) &= \pi(Y)P(X|Y) \end{aligned}$$

Over all X:

$$\sum_X \pi(X)P(Y|X) = \sum_X \pi(Y)P(X|Y)$$

$$\sum_X \pi(X)P(Y|X) = \pi(Y)$$

This says that

$$\pi(X) = \pi(Y)$$

so we are going to generate Y's using the same distribution as X. If X is from π so will Y! Holy cow! So the practical side of this is as the Markov model approaches its stationary distribution it approaches generating samples from the desired distribution!

7.2.3 Some Practical Points about MCMC

So we have built a Markov model that generates a set of samples from any distribution π and a proposal distribution q. But how many samples do you need?

convergence

burn-in

mixing

7.2.4 MCMC for trees

In doing MCMC for trees we need to be able to generate samples of trees from a proposal distribution $q(.|\tau)$ of the probability of getting the next tree given the last tree and we need our target distribution π. This is seen in the equation for α:

$$\alpha(X, Y) = \min\left(1, \frac{\pi(Y)q(X|Y)}{\pi(X)q(Y|X)}\right)$$

So all we need is π. But wait! We want the distribution of trees τ which is $P(T, t_\bullet \,|\, x^\bullet)$. This would allow us to take some function of each tree times the probability of that tree occurring and get the expected value! So we can use this MCMC to compute the expected value. That is expected value of some function over samples, X, that come in a distribution π can be estimated by generating sample X_i with distribution π and averaging the function values over those:

$$E[f(X)] \approx \frac{1}{n}\sum_{i=1}^{n} f(X_i) \text{ assuming } X_i \text{ are generated with distribution } \pi$$

In the case of trees:

$$P(T, t_\bullet \,|\, x^\bullet) = \frac{P(x^\bullet \,|\, T, t_\bullet)P(T, t_\bullet)}{P(x^\bullet)} = \frac{P(x^\bullet \,|\, T, t_\bullet)P(T, t_\bullet)}{\sum\limits_{\tau \in \,\text{all}(T, t_\bullet)} P(x^\bullet \,|\, \tau)P(\tau)}$$

The final trick is that when we use this in the α equation (allowing me to mix my X/Y model with my $\tau/T/t_\bullet$ model:

$$\begin{aligned}
\alpha(X,Y) &= \min\left(1, \frac{\pi(Y)q(X|Y)}{\pi(X)q(Y|X)}\right) \\
&= \min\left(1, \frac{P(Y\,|\,x^\bullet)q(X|Y)}{P(X\,|\,x^\bullet)q(Y|X)}\right) \\
&= \min\left(1, \frac{P(x^\bullet\,|\,Y)P(Y)\left[\sum_{\tau\in\,\mathrm{all}(T,t_\bullet)} P(x^\bullet\,|\,\tau)P(\tau)\right] q(X|Y)}{P(x^\bullet\,|\,X)P(X)\left[\sum_{\tau\in\,\mathrm{all}(T,t_\bullet)} P(x^\bullet\,|\,\tau)P(\tau)\right] q(Y|X)}\right) \\
&= \min\left(1, \frac{P(x^\bullet\,|\,Y)q(X|Y)}{P(x^\bullet\,|\,X)q(Y|X)}\right)
\end{aligned}$$

The intractable sum cancels as does the priors on X and Y! Check and mate! So if we can compute the likelihood and the $q(.|.)$ we can efficiently sample the posterior distribution by Markov model. “Believe it, or not”.

In fact, if $q(X|Y) = q(Y|X)$ the proposal is symmetric. This is the original Metropolis algorithm from 1953... yes... 1953.

$$\alpha(X,Y) = \min\left(1, \frac{\pi(Y)}{\pi(X)}\right)$$

or

$$\alpha(X,Y) = \min\left(1, \frac{P(x^\bullet\,|\,Y)}{P(x^\bullet\,|\,X)}\right)$$

7.2.5 Incorporating an Evolutionary Model

The individual likelihoods for the data given a tree can be expanded to include the extra parameters but this will require that we introduce integration over these parameters.

$$P(x^\bullet\,|\,T,t_\bullet) = \int_\theta P(T,t_\bullet,\theta)\mathrm{pdf}(\theta)\,d\theta \tag{7.2}$$

where $\mathrm{pdf}(\theta)$ is the probability density function of the θ which are parameters such as the evolutionary model parameters. In this expression $P(T,t_\bullet,\theta)$ represents the likelihood as a density function over θ.

In Equation 7.1 every occurrence of a likelihood is now replaced with an integral from Equation 7.2. The result, although a more refined model, appears to be much harder to compute.

7.2.6 Mr. Bayes, MCMC, and Bootstrapping

If you'll remember we talked about bootstrapping your ML procedure to get a reliability estimate. It is when you run optimize the ML tree on many samples (typically a 1000 or more) and build a consensus tree from that the resulting optimized trees. Compare that with MCMC. Once you have passed burn-in it generates one tree per ML calculation! You can build a consensus tree from the resulting sample distribution. That is much faster!

Mr Bayes uses MCMC to generate such a sample and the parameters will now look familiar to us plus some new twists.

- **nst** is a parameter that lets us set the evolutionary model assumed. e.g. `nst=6`
- **rates** assumed distribution on model parameters e.g. `rates=gamma`
- **ngen** number of generations (evaluations) e.g. `ngen=1000000`
- **samplefreq** frequency of tree sampling e.g. `samplefreq=100`
- **nchains** number of chains running simultaneously e.g. `nchains=4`
- **burnin** burnin time in generations e.g. `burnin=100000`

The trees generated by Mr. Bayes can be fed to PAUP* to find the consensus tree using the consensus tree tool which uses the same algorithms we discussed in here.

Chapter 8

Grammatical Analysis of Sequences

The Chomsky Hierarchy

- Unrestricted Grammars - equivalent to Turing machines and so encompasses all computation
- Context Sensitive Grammars - substitutions can be sensitive to context, equivalent to linear bounded automata
- Context Free Grammars - has hierarchical structure, can do nesting of structures, equivalent to pushdown automata
- Regular Grammars - very linear structure, can't match nested structures, equivalent to finite state automata and HMMs

A **language** is a set of legal strings. If M is a model of a language then $L(M)$ is the set of strings **accepted** by the language. In a **stochastic grammar** there is a probability that a string S is accepted. This is denoted: $P(S \mid M)$

A **Context Free Grammar** (**CFG**) has a model that can be represented as

$$M = (T, N, S, \pi)$$

Where

T	- a set of **terminal symbols** such as $\{a, b, c, d\}$
N	- a set of **nonterminal symbols** such as $\{A, B, C, D\}$
S	- a **start symbol** chosen from $T \cup N$
π	- a set of productions

Let α^* denote the set of strings each of which is zero or more elements from the set α. Let α^+ denote the set of strings each of which is one or more elements from the set α. Let ϵ denote the string of no symbols.

Productions are rules of the form:

$$A \rightarrow \quad \lambda$$

where $A \in N$ and $\lambda \in (T \cup N)^*$

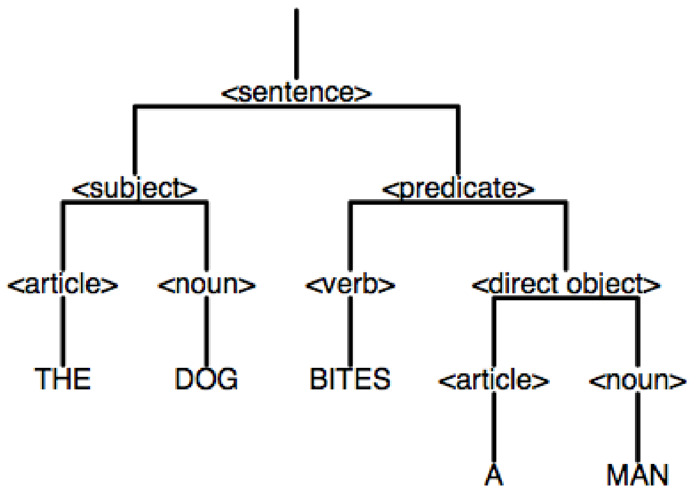
An example set of productions for English. The start symbol is $< sentence >$.

$$\begin{array}{lll}
< sentence > & \rightarrow & < subject >< predicate > \\
< subject > & \rightarrow & < article >< noun > \\
< predicate > & \rightarrow & < verb >< direct - object > \\
< direct - object > & \rightarrow & < article >< noun > \\
< article > & \rightarrow & THE \ | \ A \\
< noun > & \rightarrow & MAN \ | \ DOG \\
< verb > & \rightarrow & BITES \ | \ PETS
\end{array}$$

A **statement** in the language is a string $\vec{x} \in x^*$ where $x \in T$

Parse is the process of showing how a sentence could be built from a grammar.

A **parse tree** is a tree based notation showing how a specific sentence is parsed by a set of productions.



A **derivation** is the ordered list of steps used in construction of a specific parse tree for a sentence from a grammar.

Left most derivation is a derivation in which the left most nonterminal is always replaced first.

A **derivation** can be represented as a list of transformations of strings from $T \cup N$ via productions:

$$[\vec{s_0}, \vec{s_1}, \vec{s_2}, \vec{s_3} \ldots, \vec{s_k}]$$

where $\vec{s_0} = S$ and $\vec{s_k} = \vec{x}$ and each step from $\vec{s}_i$ to $\vec{s}_{i+1}$ is governed by a production from π such that the **left hand side** of the production is replaced by the **right hand side** of the production in one place in string $\vec{s}_i$ to yield $\vec{s}_{i+1}$. This is list a called a **derivation**.

CFGs allow arbitrary nesting of elements which a Regular Grammar cannot do. For example:

$$\begin{array}{lcl} X & \rightarrow & (X) \\ X & \rightarrow & \epsilon \end{array}$$

defines a language of matched pairs of parentheses.

Productions for a list of X's

$$\begin{array}{lcl} < sentence > & \rightarrow & < sentence > X \\ < sentence > & \rightarrow & X \end{array}$$

A language does not necessarily have a unique grammar:

$$\begin{array}{lcl} < sentence > & \rightarrow & X < sentence > \\ < sentence > & \rightarrow & X \end{array}$$

This is also a list of X's but allows the empty list.

$$\begin{array}{lcl} < sentence > & \rightarrow & < sentence > X \\ < sentence > & \rightarrow & \epsilon \end{array}$$

8.1 RNA Folding

8.2 Stochastic Context Free Grammar

A **Stochastic Context Free Grammar** (**SCFG**) also known as **Probablistic Context Free Grammar** (**PCFG**) is a model that include a probability with each production.

$$M = (T, N, S, \pi, P)$$

Where $P : \pi_k \rightarrow \mathbb{R} \ \ \forall \ \ \pi_k \in \pi$ and the probabilities are normalize such that $\ \ \forall \ \ X \in N$:

$$\sum_{X \ \rightarrow \ \lambda \in \pi} P(X \ \rightarrow \ \lambda) = 1$$

That is, the sum of the probabilities for each left hand side nonterminal is 1.

If a series of productions $[\pi_1, \pi_2, \pi_3, \ldots \pi_k]$ is a derivation of $\vec{x}$ using model M then the probability of that derivation happening is:

$$P(S \ \rightarrow \ \vec{x} \,|\, \pi) = \prod_{i=1}^{k} P(\pi_i)$$

and given all **left most derivations** of $\vec{x}$:

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

Example CFG:

$$\begin{aligned}
S &\ \rightarrow\ aXu \\
X &\ \rightarrow\ aYu \\
Y &\ \rightarrow\ cZg \\
Z &\ \rightarrow\ BBBB \\
B &\ \rightarrow\ a \mid c \mid g \mid u
\end{aligned}$$

A derivation would be

$$\begin{gathered}
S \\
aXu \\
aaYuu \\
aacZbuu \\
aacBBBBbuu \\
aacgBBBbuu \\
aacggBBbuu \\
aacgggBbuu \\
aacggggbuu
\end{gathered}$$

An example SCFG:

$$\begin{aligned}
A &\ \rightarrow\ aBu, P = .5 \\
B &\ \rightarrow\ uCa, P = .5 \\
C &\ \rightarrow\ cDg, P = .5 \\
D &\ \rightarrow\ cEg, P = .5 \\
Z &\ \rightarrow\ BB, P = 1.0 \\
B &\ \rightarrow\ a \mid c \mid g \mid u, P = .25
\end{aligned}$$

$$\begin{gathered} S \\ aSu \\ aaSuu \\ aaaSuuu \\ aaauZauuu \\ aaauBBauuu \\ aaaugBauuu \\ aaauguauuu \end{gathered}$$

$$P(aaauguauuu) = .5.5.5.51.0.25.25 = 0.00390625$$

8.2.1 Chomsky Normal Form (CNF)

Any CFG can be converted into a grammar whose productions are either:

$$\begin{aligned} X &\rightarrow YZ \\ X &\rightarrow a \end{aligned}$$

- Eliminating useless nonterminals (nonterminals that only derive ϵ)
- Eliminating **null productions** ($X \rightarrow \epsilon$)
- Eliminating **unit productions** ($X \rightarrow Y$)
- Factoring series ($X \rightarrow abc$ becomes $X \rightarrow aY, Y \rightarrow bZ, Z \rightarrow c$)
- Factoring out terminals ($X \rightarrow aX$ becomes $X \rightarrow ZX, Z \rightarrow a$

8.3 CYK Parsing

The following version of the CKY algorithm is taken from *Seven Lectures on Statistical Parsing* by Christopher Manning, Stanford.

```
// returns most probable parse/prob
function CKY(words, grammar)
    score = new double[#(words)+1][#(words)+][#(nonterms)]
    back = new Pair[#(words)+1][#(words)+1][#nonterms]]

    for (i=0; i<#(words); i++)
        for A in nonterms
            if A -> words[i] in grammar
                score[i][i+1][A] = P(A -> words[i])

    // handle unaries
    added = true
    while added
        added = false
        for A, B in nonterms
            if score[i][i+1][B] > 0 && A->B in grammar
                prob = P(A->B)*score[i][i+1][B]
                if prob > score[i][i+1][A]
                    score[i][i+1][A] = prob
                    back[i][i+1] [A] = B
                    added = true

    for span = 2 to #(words)
        for begin = 0 to #(words) - span
            end = begin + span
            for split = begin+1 to end-1
                for A,B,C in nonterms
                    prob = score[begin][split][B] * score[split][end][C] * P(A->BC)
                    if prob > score[begin][end][A]
                        score[begin][end][A] = prob
                        back[begin][end][A] = new Triple(split, B, C)

                // handle unaries
                added = true
                while added
                    added = false
                    for A, B in nonterms
                        prob = P(A->B) * score[begin][end][B]
                        if prob > score[begin][end] [A]
                            score[begin][end][A] = prob
                            back[begin][end][A] = B
                            added = true
```

```
  return buildTree(score, back)
```

In class example:

$$\begin{aligned} N_1 &\rightarrow N_1 + N_1 \\ N_1 &\rightarrow N_2 * N_2 \\ N_1 &\rightarrow X \\ N_2 &\rightarrow X \end{aligned}$$

Convert to Chomsky Normal Form by adding nonterminals Z_x:

Production	*Probability*	*Unary*
$N_1 \rightarrow N_1\ Z_1$	P_1	
$Z_1 \rightarrow Z_3\ Z_2$	P_2	
$Z_2 \rightarrow N_1$	P_3	✓
$Z_3 \rightarrow +$	P_4	✓
$N_1 \rightarrow N_2\ Z_4$	P_5	
$Z_4 \rightarrow Z_6\ Z_5$	P_6	
$Z_5 \rightarrow N_2$	P_7	✓
$Z_6 \rightarrow *$	P_8	✓
$N_1 \rightarrow X$	P_9	✓
$N_2 \rightarrow X$	P_{10}	✓

Given string $X + X * X$ what is the most likely parse tree?

Chapter 9

Sequence Assembly

9.1 Shotgun Sequencing

Shotgun sequencing is the inference of a large sequence by breaking up the sequence into many random small parts and then performing **sequence assembly**.

These notes are from chapter 7 of Waterman's book: "Introduction to Computational Biology: Maps, Sequences and Genomes".

9.1.1 The Shortest Common Superstring Problem

Assume you have a sequence: $a = a_1a_2...a_L$ and you have N fragments $\mathcal{F} = \{f_1, f_2, ...f_N\}$. We assume $f_i \neq f_j, i \neq j$ and no f_i is a substring of f_j. We make no claims that $|f_i| = |f_j|$.

The **shortest common superstring** or **SSP** problem. Given set $\mathcal{F} = \{f_1, f_2, ...f_N\}$ find the string S such that f_i is a substring of S for all i. Turns out this problem is NP complete. By saying a problem is **NP complete** we are classifying it as a problem that grows exponentially more difficult with respect to some measure of problem size. (See a theory computation book for a much longer and more detailed explanation. e.g. Introduction to Languages and the Theory of Computation by Matin) But we can do a practical job assembling a good enough sequence.

Here is a sample set of six fragments we will use in the examples to follow. These fragments just happen to be the same length but in general fragments are of various lengths.

$$\begin{aligned} f_1 &= \text{ATAT} \\ f_2 &= \text{TATT} \\ f_3 &= \text{TTAT} \\ f_4 &= \text{TATA} \\ f_5 &= \text{TAAT} \\ f_6 &= \text{AATA} \end{aligned}$$

The shortest string for the above fragment set is:

TAATATTATA

which coresponds to the overlapping sequences:

$$(f_5, f_6, f_1, f_2, f_3, f_4)$$

How do we discover a sequence like this? Consider ordered pair (i, j) then let v be the longest substring such that $f_i = uv, f_j = vw$. Note that $v = \varnothing$ is possible. This is the longest overlapping string. Note that $u \neq \varnothing$. If either $u = \varnothing$ or $w = \varnothing$ then one string is inside the other which is not allowed by our assumptions. Define **overlap** to be: $ov(i, j) = |v|$ and **prefix** to be: $pf(i, j) = |u|$. Note: $ov(i, j)$ does not necessarily equal $ov(j, i)$.

Consider:

$$f_1 = \text{TATAGCG}$$

$$f_2 = \text{GCGTA}$$

Then

$$ov(1,2) = 3 \qquad \text{and} \qquad pf(1,2) = 4$$

but also

$$ov(2,1) = 2 \qquad \text{and} \qquad pf(2,1) = 3$$

and

$$ov(1,1) = 0 \ \text{ since } \ u \neq \varnothing$$

if

$$f_3 = \text{TATATA}$$

then

$$ov(3,3) = 4 \ \text{ since } \ u \neq \varnothing$$

We define a **prefix graph** to be:

- Edge weighted directed graph
- N verticies that are from $\mathcal{F}$
- N^2 edges labeled with $pf(i, j)$. These are the edge weights. Note that both a edge from i to j exists and from j to i with possibly different weights.

What we are doing in solving the SSP is trying to find a cycle in the complete weighted graph where the weights of the edges are the values of $pf(i, j)$. To see this imagine that we have k strings that form a list of vertices in our graph. If we sum the edges that sums up all the extra nonoverlapping parts until you get to that last sequence which overlaps with the first. This would be great if our original sequence was a circular piece of DNA like a plasmid. But most of the time it isn't so we must add the full length of the final string. This problem is very similar to the famous **traveling salesperson problem** (TSP). This is a classic example of a problem in computer science that is provably incredibly difficult.

One way to solve this problem is to use a **greedy algorithm** to find an adequate solution. A greedy algorithm is one that uses local information only to solve a global problem. That is, it makes the "greedy" decision based on a short term goal. This is often a useful **heuristic**. Here is the greedy algorithm:

```
$S \leftarrow \varnothing$
$F \leftarrow \{f_1, f_2, ...f_N\}$
while $F \neq \varnothing$ {
    select edge $(a, b) \in F$ and maximizes $ov(a, b)$
    if $(a, b)$ completes a cycle then {
        remove $a$ from $F$
        add $a$ to $S$
    }
    else {
        merge $a$ and $b$
    }
}
```

If we form a matrix of the overlap function for all pairs of fragments we can process the matrix similarly to the way we did distance methods for phylogenetic trees. Below we labeled boxes show the order in which the fragments are assembled.

A Table of $ov(i, j)$ with first sequence i and following sequence j.

$i\backslash j$	1	2	3	4	5	6
1	2	3	1	$\boxed{3}_1$	1	0
2	0	1	$\boxed{2}_6$	1	1	0
3	2	$\boxed{3}_4$	1	3	1	0
4	3	2	0	2	$\boxed{2}_5$	1
5	2	1	1	1	1	$\boxed{3}_3$
6	$\boxed{3}_2$	2	0	2	2	1

This generates the following assembly first a four fragment cycle and then a second two fragment assembly.

$$\begin{array}{cc} & f_1f_4 \\ & f_6f_1f_4 \\ & f_5f_6f_1f_4 \\ f_3f_2 & f_5f_6f_1f_4 \end{array}$$

We can now assemble these cycles. Note that it matters in what order we assemble the sequences.

$$f_3 f_2 = \text{TTATT}$$
$$f_5 f_6 f_1 f_4 = \text{TAATATA}$$

which can be merged to

TTATTAATATA

which is 11 characters or one more than the optimal sequence. If we assembled the cycles in a different order we would have gotten a 12 character sequence.

TAATATATTATT

Our greedy algorithm only uses the orginal overlap values. Perhaps we could look for new overlaps that might be formed. In fact that will do us no good. To see that the question: can f_i be a substring of $f_1 \cup f_2 = f = uvw$? If it could then v must be contained in f_i or f_i would have been contained in either of f_1 or f_2. But if v is in f_i and not contained then it must have some part of u in it as well. But then $ov(i, 2) > ov(1, 2)$ but that can't be because of our algorithm.

9.1.2 Real World Shotgun Sequencing

But, of course, things are never so nice. The fragments we get generally:

1. Some fragments are substrings of others
2. We don't know which direction to read the string. The shotgun approach to shattering the sequence doesn't perserve only only one direction of the string. We now need to consider that either f_i or its reverse f_i^r could be used.
3. And worst of all our sequencers are not perfect. We have to deal with mismatches, insertions and deletions just like we have seen in sequence alignment.

Here is a rough outline of how practical shotgun sequencing is done.

1. Compute all pairwise alignments both between f_i and f_j and between f_i and f_j^r. This can be done by using dynamic programming as we did at the beginning of class. For any pair of sequences it could be that one sequence is contained inside the other. We setup the DP matrix. We detect contained sequences by looking for matches that terminate in the middle of the DP matrix. We remove duplicates. Then we then detect if there is an overlap. This can be done by looking for matches along the lower edge or the right edge of the DP matrix.
2. choose an orientation by
$$\max(A(f_i, f_j), A(f_i, f_j^r))$$
where A is the alignment score.

3. ignore alignments that score less than some C i.e. $A(f_i, f_j) < C$.
4. Perform greedy algorithm comparing all competing alternatives.

Each pairwise alignment is local and as such suffers from not knowing about other attempted alignments.

9.2 Other Techniques

9.2.1 Other Approaches

A ***k*-tuple approach** uses a fixed k long fragment that is either derived from longer fragments or by using microarrays and hybridizing. The later approach is called **sequencing by hybridizing** or **SBH**.

Index