1 Sequence Alignment

As mutation and selection drive evolution, DNA and genes change over time. Due to these forces, 2 species with a common ancestor will have similar, but non-identical genes in terms of base-pair sequence. This basic fact can be used for several purposes. For example, suppose you know the DNA sequence for a human gene but are unsure of the function of the gene. If you can find a gene with a similar sequence in a closely related species, then a reasonable conjecture is that the functions of the 2 genes are the same. Another application of this basic idea is in constructing phylogenetic trees, i.e. how closely related a set of species are to one another. Such considerations often have medical applications: for example, by knowing how similar different strains of human immunodeficiency virus (HIV) are to one another we may have some idea about how effective a vaccine will be that has been designed using a certain strain. (Most HIV research works with strain B, the predominant form in the US, whereas most HIV patients are infected with a non-strain B virus.) Yet another common application is in the recognition of common sequence patterns in DNA (the motif recognition problem). Recognizing common patterns in DNA or proteins can help determine where genes are in a sequence of DNA or proteins. We can also try to find locations where certain proteins bind to DNA if we suppose there is some sequence of DNA that is recognized by a protein.

These are some of the basic problems that drive the question: how similar are 2 bio-polymers. There are many other applications of these techniques. Previously we noted that while thinking of DNA as a linear molecule is often a useful abstraction, this is not a useful way to think about proteins. Hence comparison methods for proteins should consider not just the sequence of amino acids, but also the structure of the protein. Similarly, RNA can exhibit complicated 3 dimensional structures, although the structures are not as varied as protein structures.

Methods for comparing 2 or more biomolecules are typically referred to as methods of *sequence alignment*. The name comes from the basic idea that to see how similar 2 sequences are, we should write one down, then try to line up the other sequence with what we have written down. What we mean by "line up" is really the heart of the matter. Mutation of DNA or protein sequences is thought to occur through just 3 basic processes:

1. changes in a single nucleotide or amino acid
2. deletion of a single nucleotide or amino acid
3. addition of a single nucleotide or amino acid.

Differences we see in DNA across species are thought to arise through the combination of many such basic mutations and subsequent selection. Hence, if we are trying to line up sequences thought to have a common evolutionary origin, we should allow mismatches and gaps in the 2 sequences. Some mutations are selected against, for example, if a hydrophobic residue in a protein takes the place of a hydrophilic residue the protein may lose the ability to perform its function, hence we don't expect such mismatches when we line up the sequences. Other mutations may not affect the function of the biomolecule, and as such, may be consistent with a common progenitor. For these reasons, when we line up the sequences, we should consider some mismatches more severe than others, and we should also rate how severely gaps may lead to a non-functional biomolecule in terms of these mismatch penalties.

Sequences that share a certain degree of similarity are referred to as *homologous*, and we say the 2 sequences are *homologs*. *Orthologs* are homologs that are derived from a common ancestor, but have diverged due to evolutionary divergence of the organisms. *Paralogs* are homologs that arose due to gene duplication in an organism. After this gene duplication, one of the copies diverged from the common ancestor gene.

Xenologs are homologs that resulted from horizontal gene transfer, i.e. a gene entered the genome of some organism from another organism. Orthologs tend to have common functions, whereas paralogs tend to have different functions. Xenologs may or may not have the same function. A gene is typically inferred to be a xenolog if the gene seems out of place in a genome. For example, the G-C content of a gene that is horizontally transferred into a species may depart radically from the overall G-C content of the genome into which the gene has been transferred.

2 Dot plots

The idea of the dot plot is quite simple: put 2 sequences along the margins of a table and enter a dot in the table if the 2 sequences have the same letter at that location. One can then graphically inspect the sequences to detect if there are long shared regions. Such regions will show up as a long diagonal string of dots in the plots. While this simple method would indicate stretches of shared sequence elements, it would not reflect the fact that some mutations affect the functionality of the resulting protein more than others.

A heuristic technique for comparing 2 sequences would be to count the longest diagonal stretch in the plot. One could then compare this value with what we expect if the agreement is just chance agreement. This simple idea is the basis for methods used to look for homology of a given sequence with all sequences in a database, as we discuss later.

3 Finding the most likely alignment

In order to adopt a likelihood based method for aligning 2 biosequences, we must specify the likelihood for pairs of sequences, which we will denote R_1 and R_2, with elements r_{ij} for $i = 1, 2$ and $j = 1, \ldots, n_i$. One simple approach is to suppose that the elements r_{ij} are independent and identically distributed observations from a multinomial distribution over the possible element values (e.g. amino acids). While the iid assumption is not realistic in that we certainly don't think all possible sequences of amino acids of a fixed length are equally likely to be observed in living organisms (since such a sequence performs some function), and given evidence for serial dependence in sequences of DNA, these assumptions greatly simplify problems in alignment. In this case $L(R_i) = \prod_j^{n_i} P(r_{ij})$ and the only parameters entering the likelihood are the frequencies of observing the different sequence elements. But if we consider the joint likelihood $L(R_1, R_2) = L(R_1|R_2) \prod_j^{n_2} P(r_{2j})$ we still must consider the conditional distribution of R_1 given R_2. Given our previous assumption that the elements of the 2 sequences arise in an iid fashion, it seems natural to assume that if f_1 and f_2 are the 2 functions that map the sequences to each other for a given alignment then (i.e. the 2 functions that stick the minimal number of gaps in the sequences so that we can then line them up), $L(R_1|R_2) = \prod_j^m P(f_1(r_{1j})|f_2(r_{2j}))$ (where m is a parameter that depends on the number of gaps used in the alignment). Note that this assumes that mutations at adjacent positions occur independently of each other, and this is also likely violated in reality but again, greatly simplifies matters. Note that we can express the joint log-likelihood of our sequences as

$$\log L(R_1, R_2) = \sum_j^m \log(P(f_1(r_{1j})|f_2(r_{2j}))P(f_2(r_{2j}))).$$

If we tried to use the resulting likelihood, we would find that we need to estimate $P(r_j = k)$ for all possible values of k, f_1 and f_2. Clearly these parameters are not identified with the given data since many combinations of these lead to the same value of the likelihood. From a more intuitive perspective, we don't

even have any information on functional similarity of biosequences that we are using to estimate parameters. A pragmatic approach to these problems is to determine estimates of $s(j,k) = \log\left(\frac{P(r_{1i}=j,r_{2i}=k)}{P(r_{1i}=j)P(r_{2i}=k)}\right)$ for all possible j and k using a dataset with information about protein functionality as it relates to sequence similarity, then use these estimates to find f_1 and f_2 for a given pair of sequences. This is the usual approach that is adopted in searching databases for homologous sequences.

Collections of $s(j,k)$ for amino acids are referred to as substitution matrices. A number of these have been derived using expert knowledge. There are 2 basic varieties in common practice: PAM and BLOSUM. Each variety has a number of substitution matrices indexed by a value that indicates the extent of divergence one expects between the 2 sequences. For closely related sequences, one should use a PAM matrix with a low value, such as PAM30, or a BLOSUM matrix with a high value, such as BLOSUM80. Note that neither of these types of substitution matrices indicate what gap penalty should be used. Recent empirical evidence for gap penalties when using the BLOSUM50 or PAM200 substitution matrices suggests a value of about 5 works well (Reese and Pearson (2002)). BLOSUM substitution matrices are thought to outperform the PAM matrices for distantly related sequences due to the manner in which they are constructed.

4 Dynamic Programming

Dynamic programming is a general technique that can be used to solve many sequential optimization problems, even with complicated constraints on the solutions. The technique is fundamental in several areas of probability and statistics: stochastic control, optimal stopping and sequential testing. In fact the technique was used by Abraham Wald in the 1940's to solve some optimal stopping problems before the technique was codified and named dynamic programming by Richard Bellman in the 1950's. While many problems can be addressed with the technique, it has some fundamental limitations. To fix ideas, it is best to discuss the technique with an example in mind.

Suppose we want to traverse a regular square lattice from the upper left corner to the lower right corner in a highest scoring fashion. We require that we can only move to an adjacent location on each move and each move has a certain score associated with it. Furthermore, we constrain the solution so that we only move down, right or diagonally down and to the right. We could find the highest scoring path by writing down every possible path, computing the score for each path and then selecting the highest scoring path. While this technique would find the optimal path, it could take a very long time if the lattice is large.

The principle of dynamic programming can be used to find the optimal path much more quickly. The principle of dynamic programming is as follows: if a path is an optimal path, then whatever the best initial first move is in this path, the rest of the path is optimal with respect to this first move. In other words, we don't know what the optimal path is, but whatever the optimal path is, if a location, x, is visited by that path, the rest of the optimal path is the best path of those paths that start at location x. The proof of the principle is by contradiction: suppose the principle is not true, then an optimal path has the property that it optimizes the total score of traversing the lattice, yet such a path is suboptimal if we start at the second location of the optimal path for traversing the entire lattice.

Returning to our example, this principle means that we don't have to check every path. Instead we go to each location in the lattice and ask: if this location is visited by the optimal path, what was the highest scoring way to get here and what is the score? We ask this question of every location in the lattice, starting at the upper left corner and working to the right and down. If we store all this information (i.e. the optimal way to get to each location and the score for doing so), then when we have asked the question of the location in the bottom right corner, we will have the score of the highest scoring path, and we will know what is the optimal move to get to the bottom right corner from the previous location in the optimal path. We can then

go to this previous location in the optimal path and find out what was the optimal way to get there (since we stored all that information). We then continue the process of tracing back the previous optimal location until we reach the upper left corner. In the end, we have the highest scoring path and its score.

Although we have discussed dynamic programming for only this simple example, the basic idea carries over to a more general setting. Most problems amenable to treatment by dynamic programming can be fit into this "traversing a lattice" or network type of problem. A simple extension of the above network problem is to suppose we want a path to go from the left side of the lattice to the right in an optimal fashion and not requiring only moves down and to the right. This setup can be used to solve many interesting problems. For example, suppose we want to find a continuous function that minimizes an integral that depends on the first derivative of the function (i.e. we want to find $f(x)$ to minimize $\int F(f', f)\, dx$ for some given F). We can approximate the solution by discretizing f, f' and the integral, and then finding the optimal path from the left side of the lattice to the right side (the scores for moves are determined by the function F). Such problems are frequently difficult to solve numerically.

The size of the lattice largely determines how quickly the method will work (this is less true for calculus of variations type problems than it is for our simple example above). Recall the point of dynamic programming is that it is faster than enumerating all paths because we just query each location. Typically the number of paths grows exponentially with the dimension of the grid, whereas the number of locations only grows like the square of the dimension of the grid. But for large grid sizes, a computation depending on the square of the dimension can even become too computationally intensive, hence dynamic programming can become infeasible in some situations. On the other hand, it is very easy to program and is guaranteed to find the *global optimum* (or all of the global optima if there is more than one), something the techniques of differential calculus can't even guarantee.

5 Using dynamic programming to find the optimal alignment

If we are given 2 sequences and the penalties for mismatches and gaps, we can use dynamic programming to find the optimal alignment. To this end, suppose sequence 1 is of length n_1 and sequence 2 is of length n_2. Consider the $n_1 + 1$ by $n_2 + 1$ lattice with each sequence written along one of the margins. An alignment can be represented as a path that only goes down and to the right from the upper left corner to the lower right corner. If the path moves to the right in the i^{th} row then the i^{th} residue in sequence 1 is aligned to a gap, if the path goes down in the i^{th} column then the i^{th} residue in sequence 2 is aligned to a gap. If the path goes diagonally, then the residues in the rows that are at the endpoint of the path are aligned to each other.

Every allowable alignment can be represented as one of the paths we have described above, hence since we are given the scores for mismatches we can determine the score for each path. Notice that the score for each path is just the sum of the scores from all of the alignments. Hence every connection between 2 adjacent locations has a score associated with it. This is exactly the sort of problem we introduced when we discussed dynamic programming, and so we can use dynamic programming to find the optimal alignment. This method is referred to as the Needleman-Wunsch algorithm. The score for aligning 2 sequences given the mismatch scores and the gap penalties is referred to as the *alignment score*.

5.1 Some variations

There are a number of simple refinements that can be made to increase the realism of the alignment method discussed above. For example, opening a gap may be the primary obstacle preventing a mutation from

occurring, but once a gap has been opened it may be easy to extend it since having a gap there doesn't affect biological function. Hence we may want to allow different penalties for gaps depending on whether the gap is opened or extended. This is easy to accommodate within the context of dynamic programming because when we find the highest scoring of reaching each location we just need to consider if there was a gap leading to each of the previous locations. In contrast, if we had a penalty that depended on if there was a gap 2 residues ago, we would encounter much more bookkeeping and the dynamic programming solution becomes more computationally intensive. We can also have position dependent scores, for example, we may want to say to sequences are more similar if they have more residues in common in some region of the molecule. For example, a ligand binding site may be known for some protein, hence we only consider proteins to be matches if they have a lot of sequence in common in this region.

References

Reese JT, Pearson WR (2002), "Empirical determination of effective gap penalties for sequence comparison", *Bioinformatics*, 11, 1500-1507.

Exercises

1. Find the optimal alignment of the 2 protein sequences (ALTN) and (AFSDG) using the BLOSOM50 score matrix and a gap penalty of 5.

2. If one compares 2 completely unrelated sequences of the same length, n, what proportion of the squares in a dot plot would be 1 if the sequences have elements from an alphabet of length a? Do you think it is easier to detect homology between proteins or DNA segments (of the same length)?