Sequence Alignment Part I: Motivation, dynamic programming, global alignment

Lecture notes for October 3rd, 2007, Adam Waite

Sequence alignment is the process by which several strings are compared and the best matches are chosen. Alignment has a prominent place in biology due to the fact that similar sequences typically share function or are otherwise related. This makes alignment one of the most useful and most used computational tools in all of molecular biology.

One example of sequence similarity is the protein exportin, which is recognizably similar enough across all eukaryotes to suggest that it has been performing the same function for hundreds of millions of years. Zymogenetics, a Seattle biotech company, got its start around the time that the human genome was being published. Every day, the latest sequencing results were downloaded and aligned with the genomes of other species on record to check for similarities. Available sequences typically come from extant species, but fragments of DNA from fossils have been recovered.

The terminology of alignment contains words that might seem familiar to biologists, but have their own meanings. The most important ones (string, prefix, suffix, substring, and subsequence) can be found on the lecture slide. Importantly, the empty string is a prefix.

Alignment is defined as the process by which two strings are made to be the same length without reordering. Thus, the means by which a match can be improved usually involves adding spaces to one or more of the strings. A means of scoring a match versus a mismatch versus a space is chosen, and the "optimal alignment" is the one which maximizes this score. A very simple scoring system would involve a positive score for every match, a small negative score for a space, and a larger negative score for a mismatch.

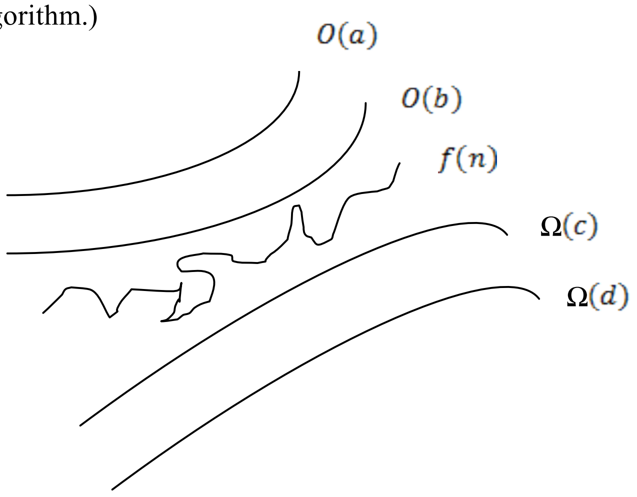
For proteins, one of the most important things an alignment can do is emphasize the structure/function relationships between sequences. To do this most effectively, one needs (and needs to fully understand) the 3D structures associated with the sequence. Since this is typically very hard to do, the simplification "similar sequence = similar structure/function" is used. However, the simple scoring system mentioned above will miss important things such as aligning two otherwise similar proteins due to dissimilar regions that may either be functionally similar (e.g., two dissimilar sequences both make an alpha-helix or a beta-sheet) *or* functionally unimportant. Thus, it is necessary to use a more complex scoring system that can consider, for example, how common certain amino acids are or their context within a larger portion of the sequence.

The other element of alignment is the particular algorithm used. Naïvely, we could just compare every subsequence to every other. This would produce the best alignment…eventually. Simple calculations (see slide 13) show that this process requires 2^{2n} operations for every n letters of sequence. As typical query lengths are in the hundreds for proteins and can be in the millions for DNA, this is not a realistic solution. On top of this, the amount of available biological data continues to increase exponentially.

The process by which the upper-bound of run time is calculated is called *asymptotic analysis* and is used to determine the "worst case scenario" for a given algorithm *independent* of any particular computer's speed, operating system, available memory, etc. Importantly, the algorithm can be *greater than* this upper bound as long as it is by no more than a constant factor (see slide 15).

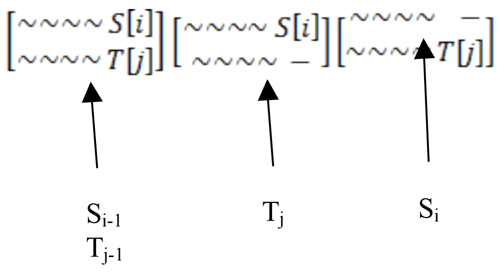
It is important to understand that, while *dynamic programming* is an important concept in computer science, it is something of a misnomer. The “dynamic” part was mostly a PR move on the part of its inventor and the “programming” part comes from an older meaning of the term. Simply put, dynamic programming uses tables and other types of stored information to use as a reference as it proceeds. (In this sense, it is more “static” than the techniques it replaced.) This way, each piece of information can be calculated (the expensive part) once and then looked up (cheap) when necessary.

(The ‘Ω’ symbol on slide 18 refers to the lower bound of a particular algorithm. Just as one tries to pick the lowest upper bound to describe a particular algorithm, one can also pick the highest lower bound to describe that algorithm.)



Alignment problems are ideally suited for dynamic programming, since many substrings and local comparisons are made, all of which can be saved and used later to generate the overall best alignment. Using dynamic programming, it is possible to bring the run time down to the order of n^2.

Note for slide 24:


$\begin{bmatrix} \sim\sim\sim\sim S[i] \\ \sim\sim\sim\sim T[j] \end{bmatrix}$
$\begin{bmatrix} \sim\sim\sim\sim S[i] \\ \sim\sim\sim\sim - \end{bmatrix}$
$\begin{bmatrix} \sim\sim\sim\sim - \\ \sim\sim\sim\sim T[j] \end{bmatrix}$
S_{i-1} T_{j-1}
T_j
S_i

The matrix on slide 25 can be filled out by using slides 23-25. The important thing is that it is done in an orderly way (row-wise or column-wise).

After the matrix has been filled out, the *best score* will be in the lower-right hand corner. To find the *best alignment*, it is necessary to step backwards through the matrix, picking the previous neighbor square (from left, top, or left-diagonal) that led to the current square. If there is a tie, there are two best alignments.