6.047/6.878 Lecture 3 - Rapid Sequence Alignment and Database Search

September 11, 2008

Last lecture we saw how to use dynamic programming to compute sequence alignments in $O(n^2)$ time. In particular, we considered performing so-called *global alignment*, in which we want to match one entire sequence with another entire sequence. The goal of such alignments is to be able to infer evolutionary events such as point mutations, insertions, deletions, etc. In this lecture, we shall look at why and how we might want to do a *local alignment* rather than *global alignment* and how to transform Needleman-Wunsch algorithm for global alignment into Smith-Waterman algorithm for local alignments. We then look at $O(n)$ algorithms for exact string matching followed by the $BLAST$ algorithm and inexact matching.

1 Global alignment vs. Local alignment

When finding a global alignment, we try to align two entire sequences. For example, we might match one gene against another gene. However, this may not always be what we want. Perhaps we want to align a gene against a whole cluster of genes, thus finding a similar gene within the cluster. Alternatively, a long sequence might be rearranged or otherwise only partly conserved. In practice, we shall be looking for regions of homology and similarity within the genes. Therefore, local alignment methods become handy.

Formally, a *local alignment* of strings s and t is an alignment of a substring of s with a substring of t.

1.1 Using Dynamic Programming for local alignments

In this section we will see how to find local alignments with a minor modification of Needleman-Wunsch algorithm that we discussed last time for finding global alignments. To find global alignments:

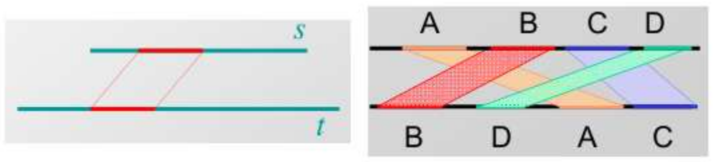


Figure 1: Local Alignments

$$
\begin{aligned}
\textit{Initialization} &: \quad F(i,0) = 0 \\
\textit{Iteration} &: \quad F(i,j) = \max \begin{cases} 0 \\ F(i-1,j) - d \\ F(i,j-1) - d \\ F(i-1,j-1) + s(x_i,y_j) \end{cases} \\
\textit{Termination} &: \quad \textit{Bottom right}
\end{aligned}
$$

To find local alignments, we should be able to skip nucleotides just to get to the segment of interest. Thus, we allow the algorithm to start and stop anywhere with no penalty:

$$
\begin{aligned}
\textit{Initialization} &: \quad F(i,0) = 0 \\
& \quad\;\; F(0,j) = 0 \\
\textit{Iteration} &: \quad F(i,j) = \max \begin{cases} 0 \\ F(i-1,j) - d \\ F(i,j-1) - d \\ F(i-1,j-1) + s(x_i,y_j) \end{cases} \\
\textit{Termination} &: \quad \textit{Anywhere}
\end{aligned}
$$

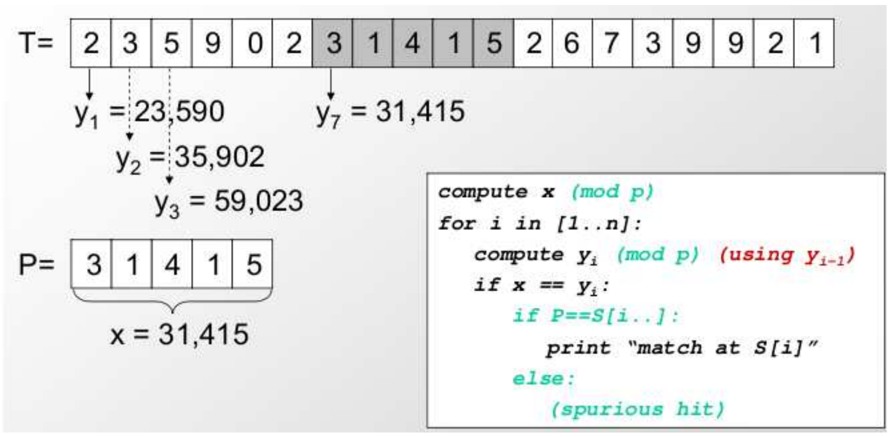
Another variation of alignments is semi-global alignment. The algorithm is as following:

$$
\begin{aligned}
\textit{Initialization} &: \quad F(i,0) = 0 \\
\textit{Iteration} &: \quad F(i,j) = \max \begin{cases} 0 \\ F(i-1,j) - d \\ F(i,j-1) - d \\ F(i-1,j-1) + s(x_i,y_j) \end{cases} \\
\textit{Termination} &: \quad \textit{Right Column}
\end{aligned}
$$

Sometimes it can be costly in both time and space to run these alignment algorithms. Therefore, there exist some algorithmic variations to save time/space that work well in practice. However, the details are outside of the scope of this lecture.

1.2 Generalized gap penalties

Gap penalties determine the score calculated for a subsequence and thus which match is selected. Depending on the model, it could be a good idea to penalize differently for, say, the



```
compute x (mod p)
for i in [1..n]:
   compute $y_i$ (mod p) (using $y_{i-1}$)
   if x == $y_i$:
      if P==S[i..]:
         print "match at S[i]"
      else:
         (spurious hit)
```

Figure 2: Karp Rabin Algorithm

gaps of different lengths. However, the is also cost associated with using more complex gap penalty functions such as substantial slower running time. There exist nice approximations to the gap penalty functions.

2 Linear-time exact string matching

When looking for exact matches of a pattern, there is a set of algorithms and data structures that work well in some applications and another set that work well in other applications. Some of them that are good to know are Z-algorithm, suffix trees, suffix arrays, etc. In this lecture we will talk about Karp-Rabin algorithm.

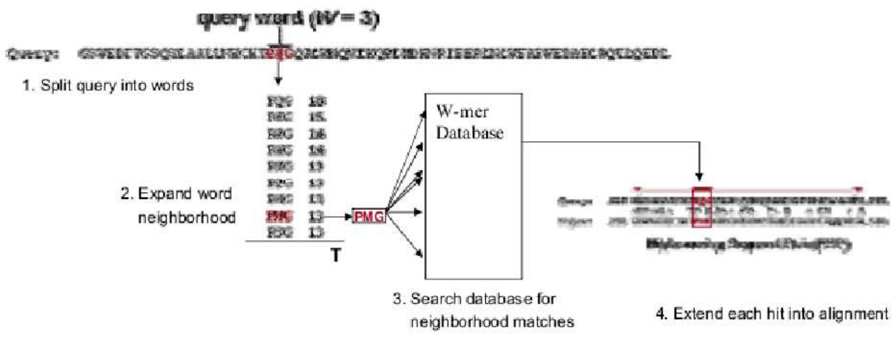
2.1 Karp-Rabin Algorithm

The problem is as follows: in text T of length n we are looking for pattern P of length m. The key idea behind this algorithm is to interpret strings as numbers that can be compared in constant time. Translate the string P and substrings of T of length m into numbers (x and y, respectively), slide x along T at every offset until there is a match. This is O(n).

One might raise the objection that converting a substring of T of length m into a number y would take $O(m)$ and therefore the total running time is $O(nm)$. However, we only need to do this only once since the number at the subsequent offset can be computed based on the number at the previous offset using some bit operations: a subtraction to remove the high-order bit, a multiplication to shift the characters left, and an addition to append the low-order digit (a diagram will be attached). Therefore, it is only $O(n+m)$.

These ideas can be used to rapidly index consecutive W-mers in a lookup table or hash table, using the index of the previous string to compute the index of the previous string in constant time, without having to read the entire string.

Also, there is always the risk that the numerical representations of x and y become large enough to exceed the processor word size in which case there is no guarantee that the



query word (W = 3)
Query:
1. Split query into words
2. Expand word neighborhood
T
PMG
W-mer Database
3. Search database for neighborhood matches
4. Extend each hit into alignment

Figure 3: The Blast Algorithm

comparisons are constant time. To eliminate this, we use hash (e.g., mod q) to map x and y to a smaller space. As a tradeoff, this is susceptible to collisions, which translate into false positive matches. However, there is a fix for that too.

3 The BLAST algorithm (Basic Local Alignment Search Tool)

While Dynamic Programming (DP) is a nice way to find alignments, it will often be too slow. Since the DP is $O(n^2)$, matching two $3,000,000,000$ length sequences would take about $9 \cdot 10^{18}$ operations. What we'll do is present BLAST, an alignment algorithm which runs in $O(n)$ time. For sequences of length $3,000,000,000$, this will be around $3,000,000,000$ times faster. The key to $BLAST$ is that we only actually care about alignments that are very close to perfect. A match of 70% is worthless; we want something that matches 95% or 99% or more. What this means is that correct (near perfect) alignments will have long substrings of nucleotides that match perfectly. As a simple example, if we have a 1000 length sequence A and a second 1000 length sequence B which is identical to A except in 10 point mutations, then by the Pigeonhole Principle[1], there is a perfectly matching subsequence of length at least 100. If we suppose that the mutations happen at random, the expected length of the longest perfectly matching subsequence will actually be considerably longer. In biology, the mutations that we find will not actually be distributed randomly, but will be clustered in nonfunctional regions of DNA while leaving untouched long stretches of functional DNA.

The other aspect of BLAST which allows us to speed up repeated queries is the ability to preprocess a large database of DNA off-line. After preprocessing, searching for a sequence of length m in a database of length n will take only $O(m)$ time.

[1]The Pigeonhole Principle states that if we have n pigeons stuffed into k holes, then some hole must have at least $\lceil n/k \rceil$ pigeons.

3.1 The BLAST algorithm

The steps are as follows:

1. Split query into overlapping words of length W (the W*-mers*)
2. Find a "neighborhood" of similar words for each word (see below)
3. Lookup each word in the neighborhood in a hash table to find where in the database each word occurs. Call these the *seeds*, and let S be the collection of seeds.
4. Extend the seeds in S until the score of the alignment drops off below some threshold X.
5. Report matches with overall highest scores

Pre-processing step of *BLAST* is to make sure that all substrings of W consecutive nucleotides will be included in our database (or in a hash table[2]). These are called the W*-mers* of the database.

As in step 1, we first split the query by looking at all substrings of W consecutive nucleotides in the query. For protein BLAST, we then modify these to generate a "neighborhood" of similar words based on amino-acid similarity scores from the similarity matrix. We take progressively more dissimilar words in our neighborhood until our similarity measure drops below some threshold T. This allows us to still find matches that don't actually have W matching characters in a row, but which do have W very similar characters in a row.

We then lookup all of these words in our hash table to find *seeds* of W consecutive matching nucleotides. We then extend these seeds to find our alignment. A very simple method for doing this which works well in practice is to simply greedily scan forward and stop as soon as the first discrepancy is found. Alternatively, one could use the DP alignment algorithm on just the DNA around the seed to find an optimal alignment in the region of the seed. Since this is a much shorter segment, this will not be as slow as running the DP algorithm on the entire DNA database.

Note that if the W-mers are small (6-10 nucleotides or 4-6 amino-acids), or if all W-mers are well-populated (that is, $4^W <$ database size for nucleotides, or $20^W <$ database size for amino-acids), then a simple table lookup can be more efficient than using a hash table (i.e. directly look up the address by interpreting the string as a number). A hash table becomes necessary when 4^W (or 20^W) is much greater than the genome/proteome, in which case the table is typically too large to fit in memory, and even if it did, most of the entries in a direct lookup table would be empty, wasting resources.

[2]Hash tables allow one to index data in (expected) constant time even when the space of possible data is too large to explicitly store in an array. The key to a hash table is a good hash function, which maps one's data to a smaller range. One can then explicitly index this smaller range with an array. For a good hash table, one needs a fast, deterministic hash function which has a reasonably uniformly distributed output regardless of the distribution of the input. One also needs policies for handling hash collisions. For more information on hashing, refer to the notes from 6.046.

3.2 Extensions to BLAST

- **Filtering** — Low complexity regions can cause spurious hits. For instance, if our query has a string of copies of the same nucleotide and the database has a long stretch of the same nucleotide, then there will be many many useless hits. To prevent this, we can either try to filter out low complexity portions of the query or we can ignore unreasonably over-represented portions of the database.

- **Two-hit BLAST** — If there's a mutation in the middle of a long sequence, then we may not have a match with a long W-mer when there is still a lot of matching DNA. Thus, we might look for a pair of nearby matching, but smaller, W-mers. Getting two smaller W-mers to match is more likely than one longer W-mer. This allows us to get a higher sensitivity with a smaller W, while still pruning out spurious hits. This means that we'll spend less time trying to extend matches that don't actually match. Thus, this allows us to improve speed while maintaining sensitivity.

- **Combs** — Recall from your biology classes that the third nucleotide in a triplet usually doesn't actually have an effect on which amino acid is represented. This means that each third nucleotide in a sequence is less likely to be preserved by evolution, since it often doesn't matter. Thus, we might want to look for W-mers that look similar except in every third codon. This is a particular example of a *comb*. A comb is simply a bit mask which represents which nucleotides we care about when trying to find matches. We explained above why 110110110... might be a good comb, and it turns out to be. However, other combs are also useful. One way to choose a comb is to just pick some nucleotides at random. This is called *random projection*.

4 The Statistics of Alignments

As described above, the BLAST algorithm uses a scoring (substitution) matrix to expand the list of W-mers to look for and to determine an approximately matching sequence during seed extension. But how do we construct this matrix in the first place? How do you determine the value of $s(x_i, y_j)$?

The most basic way to do this is to first do some simple(r) alignments by hand (alternately, we can ask an algorithm to do this with some randomized parameters and then refine those parameters iteratively). Once we have some sequences aligned, we can then look at the regions that are the most conserved. From these regions, we can derive some probabilities of a residue's replacement with another, i.e., a substitution matrix. Let's develop a more formal probabilistic model. (See slides on Page 5)

First, we have to step back and look at a probabilistic model of an alignment of two sequences. For this discussion, we're going to focus on protein sequences (although the reasoning is similar for DNA sequences).

Suppose we have aligned two sequences x and y. How likely are x and y related through evolution? We have to look at the likelihood ratio of the alignment score:

$\frac{P(score|related)}{P(score|unrelated)}$

More formally, for unrelated sequences, the probability of having an n-residue alignment between $\mathbf{x}$ and $\mathbf{y}$ is a simple product of the probabilities of the individual sequences since the residue pairings are independent. That is,

$$\begin{aligned}
\mathbf{x} &= \{x_1 \ldots x_n\} \\
\mathbf{y} &= \{y_1 \ldots x_n\} \\
q_a &= P(\text{amino acid } a) \\
P(\mathbf{x}, \mathbf{y}|U) &= \prod_{i=1}^{n} q_{x_i} \prod_{i=1}^{n} q_{y_i}
\end{aligned}$$

For related sequences, the residue pairings are no longer independent so we must use a different joint probability:

$$\begin{aligned}
p_{ab} &= P(\text{evolution gave rise to } a \text{ in } \mathbf{x} \text{ and } b \text{ in } \mathbf{y}) \\
P(\mathbf{x}, \mathbf{y}|R) &= \prod_{i=1}^{n} p_{x_i y_i}
\end{aligned}$$

Do some math and we get our desired expression for the likelihood ratio:

$$\begin{aligned}
\frac{P(\mathbf{x}, \mathbf{y}|R)}{P(\mathbf{x}, \mathbf{y}|U)} &= \frac{\prod_{i=1}^{n} p_{x_i y_i}}{\prod_{i=1}^{n} q_{x_i} \prod_{i=1}^{n} q_{y_i}} \\
&= \frac{\prod_{i=1}^{n} p_{x_i y_i}}{\prod_{i=1}^{n} q_{x_i} q_{y_i}}
\end{aligned}$$

Since we eventually want to compute a sum of scores and probabilities require add products, we take the log of the product to get a handy summation:

$$\begin{aligned}
S &\equiv \log \frac{P(\mathbf{x}, \mathbf{y}|R)}{P(\mathbf{x}, \mathbf{y}|U)} \\
&= \sum_i \log \left(\frac{p_{x_i y_i}}{q_{x_i} q_{y_i}} \right) \\
&\equiv \sum_i s(x_i, y_i) \\
s(x_i, y_i) &= \log \left(\frac{p_{x_i y_i}}{q_{x_i} q_{y_i}} \right)
\end{aligned}$$

The above expression for $s(x_i, y_i)$ is then used to crank out a substitution matrix like BLOSUM62 (Page 6, Slide 2). Incidentally, the BLOSUM62 matrix captures both chemical similarity and evolutionary similarities, which gives more information than a single expert system can. For example (in the slide), we see a positive score for matches between aspartate (D) and glutamate (E), which are chemically similar. Additionally, the matrix shows tryptophan (W) matching with itself has a higher score than isoleucine (I) matching with itself, i.e., $s(I, I) < s(W, W)$. This is primarily because tryptophan occurs in protein sequences less frequently than isoleucine, meaning $s(W, W)$ has a smaller denominator, but it may also mean that tryptophan is more highly conserved in general.

So now that we're given two sequences and an alignment score, how do we determine whether the score is statistically significant? That is, how can we determine whether the two sequences are related by evolution and not simply by chance? In particular, large database size increases the probability of finding spurious matches. In 1990, Karlin and Altschul determined that for two random sequences, the expected number of alignments with a score of at least S is $E(S) = Kmne^{\lambda S}$, which we can use to correct for false positives.