Lecture Notes

CMSC 858W - Algorithms for Biosequence Analysis

March 30th, 2010

Announcements

- Project criterion is available on website

Wrapup of (inexact) alignment of two sequences, MUMmer

> **Question -** "How do you align two sequences globally that are both very long?" This would correspond to, say, globally aligning two genomes to each other.

Previously we had a short pattern to be matched against a long text, i.e.

$$n = |P| \ll |T| = m$$

This would correspond to matching a single gene of one genome to an entire genome.

Now, when aligning two genomes (both of which are going to be large in length), one wouldn't expect that a fully-global alignment matches well or has any intuitive (or otherwise biological) meaning. Regions in genome $\mathcal{G}_1$ could be completely absent, or strongly mutated versions of a corresponding region in genome $\mathcal{G}_2$. Chromosomal inversion can also create problems in alignments, as some segments of $\mathcal{G}_1$ match segments of $\mathcal{G}_2$ in the forward direction, while other segments match in the reverse manner.

> **Main Idea -** We could employ our standard local alignment approach from last lecture, and then pick the best-scoring local regions.

A current system that does this is MUMmer, developed originally at TIGR and now maintained by UMD's CBCB department. MUMmer has existed in three versions:

(a) MUMmer - "Alignment of whole genomes." 1999, Nucleic Acids Research.

(b) MUMmer 2 - "Fast Algorithms for Large-Scale Genome Alignment and Comparison." 2002, Nucleic Acids Research.

(c) MUMmer 3 - "Versatile and open software for comparing large genomes." 2004, Genome Biology.

Each of these versions offers speed up or other improvements over the earlier versions. Underlying all of them, however, are several concepts. Of primary importance is the notion of a "MUM," or Maximally Unique Match.

For a pair of sequences (we think of these sequences as genomes), a MUM is a substring of both genomes, and further occurs only once in each genome (due to uniqueness requirement). Because each MUM is maximal, any extension of the MUM in either direction results in a mismatch between the two genomes. For example, in Figure 1, `ATCGTGCC` is a MUM provided it does not occur anywhere else in the genome. Observe that extending the string in either direction results in a mismatch, so the match is maximal.

```
. . .A A T C G T G C C T . . .
       | | | | | | | |
. . .C A T C G T G C C C . . .
```

Figure 1: Example of a MUM found between two sequences (genomes).

MUMmer takes two genomes as input, and acts as follows:

1. Find all MUMs that occur between the two genomes. Intuitively, we suspect that long matching sequences (i.e. MUMs) should be present as part of the global alignment.
2. Extract the longest possible sequence of MUMs in increasing order by genome position according to genome $\mathcal{G}_1$, using an adaptation of an algorithm that solves the Longest Increasing Subsequence (LIS) problem. This gives a partial construction of the entire alignment with high-confidence regions.
3. Fill in intra-MUM regions of the alignment table using more expensive (dynamic programming) approaches

This approach avoids using a fully-dynamic programming approach which takes $O\left(n^2\right)$ time where n is very large, we only perform it on a number of smaller regions. To quickly convince yourself of this, observe that even if the chosen MUMs (say k MUMs overall were chosen) were taken to be all be single characters, then the resulting cost of the alignment becomes

$$\sum_{i=0}^{k} O\left(n_i^2\right) \ll O\left(n^2\right)$$

for $\sum_{i=0}^{k} n_i = n$, where n_i is the length of unaligned region between the MUMs at index i and $i+1$. Figure 2 illustrates the benefit.

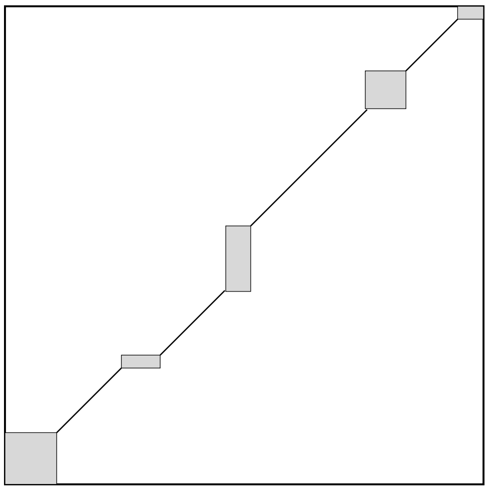

Figure 2: After choosing a set of MUMs using LIS-based approach, only the shaded regions of the overall alignment matrix must be computed. The diagonal line segments represent the selected set of MUMs that go into the final alignment.

Finding MUMs can be performed using a suffix tree (or suffix array), which can efficiently solve substring queries. We first build a suffix tree $\mathcal{T}$ over genome $\mathcal{G}_1$, and then check each suffix of $\mathcal{G}_2$ against $\mathcal{T}$. Let m be the number of characters that are matched in $\mathcal{T}$ by suffix s_i of $\mathcal{G}_2$. We can immediately jump to checking suffix s_{i+m+1} next, omitting all suffixes between. Verifying the uniqueness property of a MUM is also easy given the suffix tree. We simply check the subtree below the locatino where the m characters matched. If more than one leaf node exists, then the stretch of m characters was *not* unique and is thus not a MUM. In this way, we only check a total of $O(n)$ characters of $\mathcal{G}_2$ to determine which suffixs contain MUMs as prefixes. Using Ukkonen's suffix tree construction, the construction time for $\mathcal{T}$ is also $O(n)$, and overall it takes $O(n)$ time to determine the set of MUMs. It should be noted that the reverse complementary strands of each genome should also be considered in order to properly align the two genomes.

Finding the sequence of MUMs to use in the eventual global alignment is perforemd by an adaptation of LIS as discussed above. The MUMs are first sorted by starting position in genome $\mathcal{G}_1$ and then assigned values corresponding to their starting position in genome $\mathcal{G}_2$, at which point the longest increasing subsequence of this sequence is computed. Further details are given in the first MUMmer paper listed above. A known algorithm for LIS allows this step to be done in $O(k \log k)$ time. Because k is typically much smaller than n, this time is about $O(n)$ time (not formally shown).

The above description summarizes the work presented in the first paper regarding MUMmer (1999), as well as improvements made in MUMmer 2 (2002). MUMmer 3 (2004) open-sourced the code, as well as making several perfromance boosts. The open sourcing also makes readily available a suffix tree implementation.

Dot plots can be used to visualize the relationships between sequences on a grid. A black dot occurs at location (i, j) if the character at index i of string 1 matches character j of string 2. When considering MUMs in a region, these will manifest as long diagonal lines in the plot. If a reverse complement strand matches, the diagonal line will move in the opposite direction, as shown in Figure 3. Dotplots are computable in $O\left(n^2\right)$ time using the straight-forward approach.

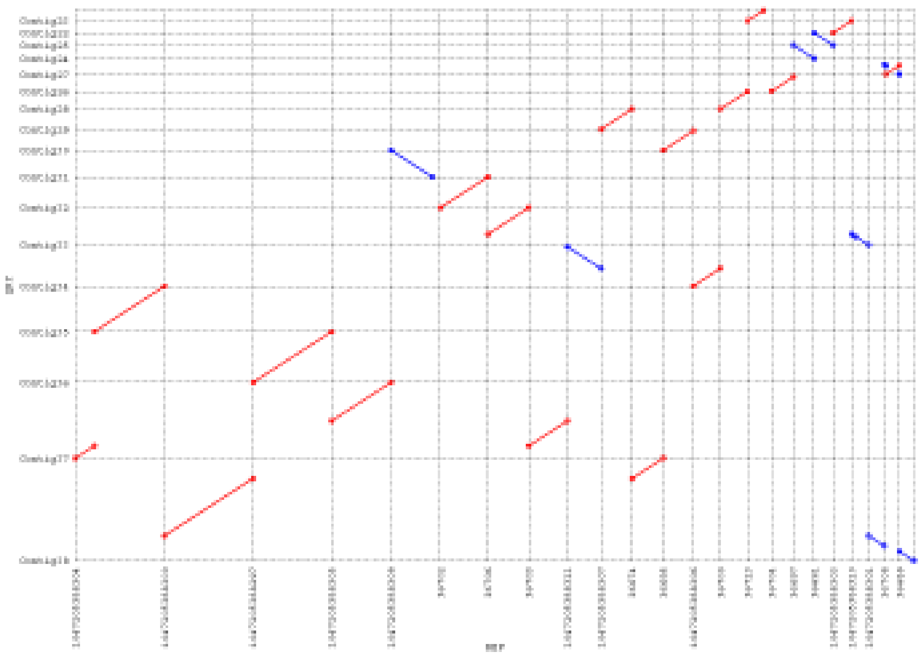

Figure 3: Dot plot. Red lines match forward direction, blue lines match reverse compliment.

Chaining local alignments

(this is paraphrased from Gusfield's section on "chaining diverse local alignments")

A related question becomes how to choose a "good" chain of MUMs that can altogether be combined to constitute the start of the global alignment. A method related to LIS was discussed above, here we detail an alternative that was presented in class that is similar to a *plane sweep* algorithm, a well-characterized algorithm in computational geometry. A simpler, 1 dimensional version of the problem is presented here to provide intuition. In this case, we can think of MUMs as intervals on a line. We are given a function $v(i)$ which assigns a value (a positive number) to each interval i. The goal is to select a subset of the intervals such that they do not overlap and whose total value $\sum_i v(i)$ is maximized. This subset is called a "chain" of intervals.

The algorithm for solving the chaining is presented here. We take advantage of dynamic programming which allows us to "greedily" scan through the set of intervals and avoid backtracking.

Algorithm 1 One dimensional chaining

```
Input:  A set $I$ of $r$ total intervals along the number line.
Output:  Maximal value of a non-overlapping chain chosen from $I$.
 1: Insert both endpoints of all $r$ intervals into a list $L$, sorted in increasing order. Keep track of
    the original interval each number in $L$ comes from.
 2: Let $V$ be an array of size $r$, initialized to all 0's
 3: Set $best \leftarrow 0$
 4: for $e$ from 1 to $2r$ do                     ▷ Step through each element of $L$ in turn.
 5:     if $L[e]$ is the left side of an interval then
 6:         Let $i$ be the interval associated with $e$.
 7:         $V[i] \leftarrow v(i) + best$
 8:     end if
 9:     if $L[e]$ is the right side of an interval then
10:         Let $i$ be the interval associated with $e$.
11:         Set $best \leftarrow \max(best, V[i])$
12:     end if
13: end for
14: Output $best$
```

As in the case with other dynamic programming algorithms, a simple modification can be made to recover the actual set of intervals that constitute the maximized-value subset rather than the value itself. *best* is the current maximized total value seen as of the current iteration of the algorithm (which corresponds to the solution for the subset of intervals whose both endpoints have been seen by the current iteration). One of two events may occur at a given iteration.

- If the left side of a new interval i is encountered at the next iteration, then that interval can potentially be used to extend the current chain. The value of the chain would then be $v(i) + best$. We are not certain if we actually want to include this interval, however, as there may be better intervals not compatible with i that lead to higher scoring chains. So, we store the value of this chain into $V[i]$ so that later it may be considered when the interval closes (with a right endpoint of i is encountered).
- If instead the right side of some interval i is encountered, then we check if the current value of *best*. If the value stored in $V[i]$ is higher, we update *best*. This effectively drops all intervals added to the running chain since encountering the left endpoint, and adds i to the end of the chain instead.

The algorithm runs in $O(r \log r)$ time, as the bottleneck is the initial sorting of the points.

Figure 4 shows a small example of the chaining algorithm being performed. During execution, the algorithm iterates as follows, with *best* starting off as 0:

1. The left endpoint of interval 1 is visited. $V[1]$ is set to 3, *best* remains 0.

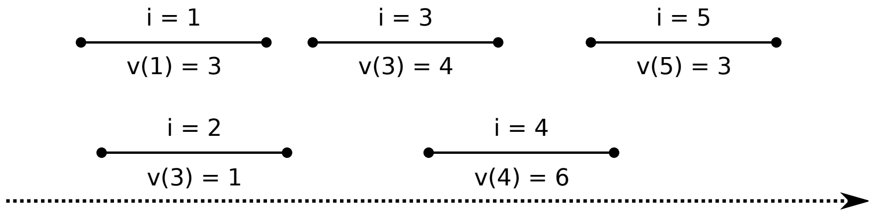

Figure 4: Sample chaining. Interval indices and corresponding values are shown. The algorithm outputs a value of 10, which can be accomplished by selecting intervals $i = 1$, $i = 3$, and $i = 5$.

2. The left endpoint of interval 2 is visited. $V[2]$ is set to 1, *best* remains 0.
3. The right endpoint of interval 1 is visited. $V[1] = 3 > \textit{best} = 0$, so *best* is updated to 3.
4. The right endpoint of interval 2 is visited. $V[2] = 1 < \textit{best} = 3$, so *best* is not updated.
5. The left endpoint of interval 3 is visited. $V[3]$ is set to 7, *best* remains 3.
6. The left endpoint of interval 4 is visited. $V[4]$ is set to 9, *best* remains 3.
7. The right endpoint of interval 3 is visited. $V[3] = 7 > \textit{best} = 3$, so *best* is updated to 7.
8. The left endpoint of interval 5 is visited. $V[5]$ is set to 10, *best* remains 7.
9. The right endpoint of interval 4 is visited. $V[4] = 9 > \textit{best} = 7$, so *best* is updated to 9.
10. The right endpoint of interval 5 is visited. $V[5] = 10 > \textit{best} = 9$, so *best* is updated to 10.

Gusfield also presents a 2-dimensional version of this algorithm, although it is *suspected* to be in error, and as such will not be repeated here. The general intuition is that here we have boxed-off regions of the two dimensional space that correspond to high similarity content between the two strings (MUMs). The goal is to use as many of the boxed off regions (which were intervals in the 1D case) as possible. One could think of the boxes as nodes in a directed acyclic graph, and the task would be to find a path from source to sink that would maximize some requirement.