CS 262 Lecture 3 Scribe Notes
Linear Space Alignment

Lecture By: Serafim Batzoglou
Scribe: Thomas Lau

January 12, 2016

1 Introduction

One of the most basic genetic sequence analysis tasks is to ask if two sequences are related. In the previous lecture, we explored several algorithms for finding optimal sequence alignments, including dynamic programming variants. One of the major computational resources that can limit dynamic programming algorithms is memory usage. In this lecture, we will explore variants of the previously presented sequence alignment algorithms that take $O(N + M)$ space. Then, we will explore heuristical alignment algorithms such as $BLAST$ and $FASTA$ that are under the dynamic programming $O(NM)$ time complexity.

2 Hirschberg's Algorithm

We will introduce an algorithm that finds the common sub-sequence of all strings in a given set. To follow the theme of this lecture, this algorithm will solve this problem in linear space. First, a few preliminaries:

Definition: A string x' is a *substring* of a string x if $x = ux'v$ for some prefix string u and suffix string v. In other words, $x' = x_i...x_j$, for some $1 \leq i \leq j \leq \mid x \mid$.

Definition: A string x' is a *subsequence* of a string x if x' can be obtained from x by deleting 0 or more characters from the string. In other words, $x' = x_{i1}...x_{ik}$, for some $1 \leq i_1 \leq ... \leq i_k \leq \mid x \mid$.

Examples:

$x = abracadabra$	
$y = cadabr$	*subsequence*
$z = brcdbr$	*subsequence, not substring*

Now, if we want to find the longest common subsequence, what should we set our alignment scores (match = x, gap = y, mismatch = z) to? It is recommended that the reader pause here as an exercise.

We will now present Hirschberg's Algorithm:

Given a set of strings $x, y, ...$, we will output the longest *common subsequence*, a string u that is a subsequence of all strings $x, y, ...$

Longest Common Subsequence:

- Given strings $x = x_1, x_2, ..., x_m$ and $y = y_1, y_2, ..., y_n$
- Find the longest common subsequence $u = u_1, ...u_k$

Algorithm:

- $F(i,j) = \max \begin{cases} F(i-1,j) \\ F(i,j-1) \\ F(i-1,j-1) + [1, \text{ if } x_i = y_j, 0 \text{ otherwise}] \end{cases}$
- $Ptr(i,j) = \begin{cases} DIAG, & \text{if [case 1]} \\ LEFT, & \text{if [case 2]} \\ UP, & \text{if [case 3]} \end{cases}$

Termination

- Trace back from Ptr(M,N) and prepend a letter to u whenever $Ptr(i,j) = DIAG$ **and** $F(i-1, j-1) < F(i,j)$

In case it was not clear from the given algorithm, to find the longest common subsequence, we simply set $match = 1$, $gap = 0$, and $mismatch = -\infty$. Intuitively, this does not let us have mismatches but allows us to have as many gaps as we want. Another variation of this algorithm not presented in lecture to find the longest common subsequence is setting ($match = 1$, $gap = 0$, and $mismatch = 0$) and then simply taking the characters that align with the given string.

3 Computing Alignment Scores in Linear Space

One of the major limitations of dynamic programming algorithms is memory usage. In the algorithms presented in the previous lecture, making the M by N dynamic programming matrix took $O(NM)$ time. To put this in relatable terms, if we are comparing two 100,000 long strings, our DP array will be 10 GB large. This may be too large to fit in the main memory of some computers. However, by examining the algorithm, we can see that it is easy to compute the optimal sequence score, $F(M,N)$, in linear space since we only need to use two columns at a time. Consider the following algorithm:

Algorithm 1 Finding F(M,N) in linear space

```
1:  procedure LinearScoreAlgorithm
2:      Allocate(column[1])
3:      Allocate(column[2])
4:      for i in 1...M do
5:          if i > 1 then
6:              Free(column[i − 2])
7:              Allocate(column[i])
8:          end if
9:          for i in 1...N do
10:             F(i,j) = max(F(i − 1, j),F(i, j − 1),F(i − 1, j − 1)+1
11:             Ptr(i,j) = DIAG (case 1), LEFT (case 2), UP (case 3)
12:         end for
13:     end for
14: end procedure
```

It is also possible for the algorithm above to be run column-by-column, row-by-row, and diagonal-by-diagonal. It is left up to the reader to prove that these additional algorithms produce the correct output. However, with the algorithm above, we are unable to find the actual optimal alignment sequence since we are not saving the back-pointers.

To compute **both** the optimal score and the optimal alignment, we will introduce Hirschberg's Linear-Space Algorithm.

4 Hirschberg's Linear-Space Algorithm

In this algorithm, we will use a divide and conquer approach to computing the optimal score and the optimal alignment in $O(N+M)$ space and $O(MN)$ time.

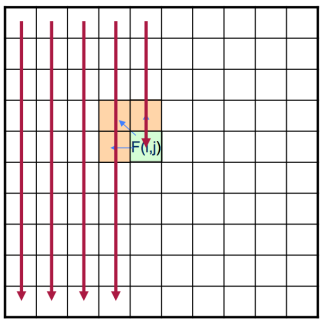

Figure 1: The "Compute Optimal Score" Algorithm allocates and deallocates columns. Only the current and the previous columns (1 x N) need to be stored, requiring O(N) space.

We now present Hirschberg's Linear-Space Algorithm:

Lemma 1: Assume that M is even. If we take the midpoint of x, $\frac{M}{2}$, we can find the index of y, k^* that $x[\frac{M}{2}]$ aligns to. (See Figure 2) Since the optimal score of a sequence is the sum of its subsequences:

$$F(M,N) = \max_{k=0\ldots N}(F(\frac{M}{2},k) + F^r(\frac{M}{2},N-k)) \tag{1}$$

Where F^r is the reverse of the string.

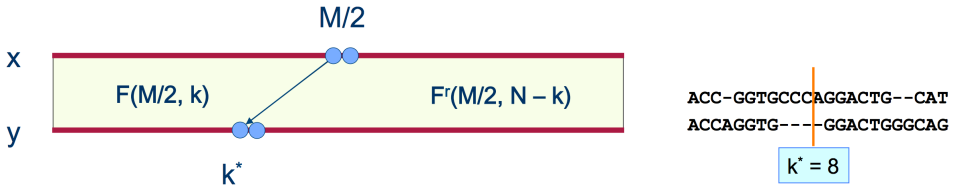

Figure 2: From this image, it is apparent that the midpoint of string x will always align to a character (match) or a gap.

Since the score is additive across subsequences, we can see that *Lemma 1* must be true. This gives us a hint as to how we could use a recursive divide and conquer approach to the optimal alignment problem.

Proof: Let $F(i,j)$ and $F^r(i,j)$ denote the minimum cost of a path from $(0,0)$ to (i,j) for string N and N^r, respectively. Let k be a number in $[0,N]$. Let q^* be an index that minimizes $F(q^*,k) + F^r(q^*,k)$. There will be a complete path of minimum length that passes through (q^*,k).
Let l denote the optimal alignment. Now let's pick a column k in our score array. Since complete alignments must start at the top left corner and end at the bottom right corner of the array, our alignment must pass through column k. We will represent the array index that is apart of the optimal alignment as (p,k).

$$l = F(p,k) + F^r(p,N-k) \geq \min_q F(q,k) + F^r(q,N-k) \tag{2}$$

$$l \geq F(q,k) + F^r(q,N-k) \tag{3}$$

Where q is the optimal row position for a given column k. Since l is the minimum length of *any* corner-to-corner path, we also have that:

$$l \leq F(q,k) + F^r(q,N-k) \tag{4}$$

It then follows that:

$$l = F(q,k) + F^r(q,N-k) \tag{5}$$

This proves that for any optimal row position chosen in a given row, the optimal corner-to-corner path goes through this position. ■

Therefore, by finding k^* that maximizes $F(M/2, k) + F^r(M/2, N-k)$, we can find portions of the optimum sequence and stitch together the optimum sequence using two columns of space, plus space for the back pointers. We will now present the divide and conquer algorithm:

Algorithm 2 Hirschberg's Linear Space Algorithm

```
1: procedure MEMALIGN(l, l', r, r')
2:     h = ceil((l'-l)/2)
3:     L_h = OPTIMALPATH()      ▷ Find the optimal path in Time O((l'-l)(r'-r)), Space (r'-r); entering column h-1, exiting column h
4:     k_1 = position at column h − 2 where L_h enters
5:     k_1 = position at column h + 1 where L_h exits
6:     MEMALIGN(l, h-2, r, k_1)
7:     Output L_h
8:     MEMALIGN(h_1, l', k_2, r')
9: end procedure
```

Where this algorithm is executed by calling MEMALIGN(1, M, 1, N).

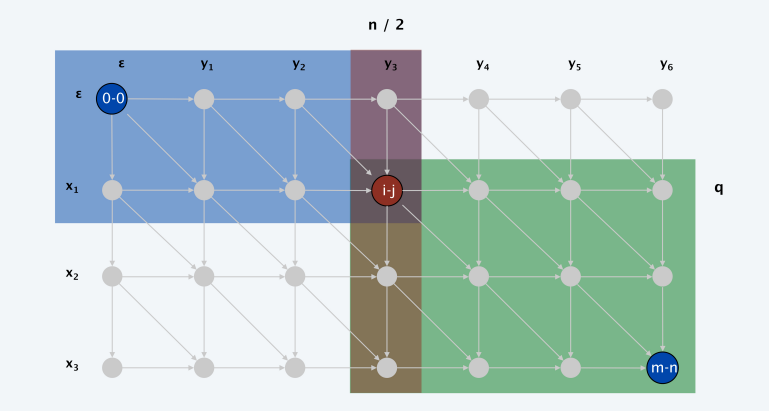

Figure 3: The dynamic programming array with divide and conquer calls highlighted and overlaid on top of the array

We will now prove the runtime of the given algorithm.

Proof: Let $T(M, N)$ denote the running time of Hirschberg's Linear Space Algorithm being executed on strings of length N and M. The algorithm performs work on the order of magnitude $O(NM)$ to build the 2 dynamic programming arrays and then recursively calls itself on strings of size $(M/2, k)$ and $(M/2, N-k)$ in a divide-and-conquer fashion. The recurrence relation is as follows:

$$\begin{aligned} T(m,n) &= cmn + T(M/2,k) + T(M/2, N-k) \\ T(m,2) &\leq cm \\ T(2,n) &\leq cn \end{aligned} \tag{6}$$

To solve this recurrence relation, we have to make a guess about the form of the solution. Assuming a perfect bisection, (k = M/2), we arrive at:

$$T(n) \leq 2T(n/2) + cn^2 \tag{7}$$

Therefore, we will guess that the solution is in the form of $T(n) = O(MN)$ and prove this by induction.

$$\begin{aligned} T(m,n) &\leq cmn + T(M/2,k) + T(M/2, N-k) \\ &\leq cmn + c'(M/2)k + c'(M/2)(N-k) \\ &= cmn + c'(M/2)k + c'(M/2)N - c'(M/2)k \\ &= (c + c'/2)mn \end{aligned} \tag{8}$$

We choose $c' = 2c$. The runtime of this algorithm is $O(MN)$. ■

For the space complexity of the algorithm, we will need O(N) for storing the computation arrays and O(N+M) to store the optimal alignment. The proof for the space complexity is significantly less difficult and is left as an exercise to the reader.

5 BLAST

The *BLAST* paper [Altschul et al. 1990] provides a heuristical based alignment algorithm for finding DNA and protein sequences with high alignment scores. Heuristical Local Alignment algorithms such as *BLAST* are necessary because running one of the previously presented algorithms with $O(MN)$ runtime would take too long when M or N are large.

The insight behind *BLAST* is that highly scoring alignment sequences will have highly scoring alignment sequences within themselves (a substring will also be highly scoring). Therefore, we take all "words" of length k for a given input string and use them as "seeds", from which we extend out to search for potential alignment matches (hit extension). In practice, *BLAST* uses words of length 3 for protein sequences and of length 11 for nucleic acids.

To make the hit extension step terminate, we implement a threshold k where if we have more than k mismatches, we terminate. Our output are simply the local alignments with a score above the statistical threshold.

In practice, the *BLAST* algorithm is surprisingly good at finding most significant local alignments. Intuitively, as the length of the "word" that we search for increases, our speed will increase but our sensitivity will decrease. This is because we will get less "seed hits" and therefore less hit extensions to investigate. When our "word" length is decreased, our speed will decrease but our sensitivity will increase - imagine what would happen if we used a "word" length of 1. Quantitative speed and sensitivity tradeoff data can be found in [Kent WJ et al. 2002].

Methods to improve the sensitivity/speed trade off are:

- Pairs of words
  - Using two smaller words (length n and m) that are a fixed distance apart. Intuitively, this would keep our speed the same as looking for a word length $n + m$, but increase our sensitivity.
- Inexact words
  - Using a threshold for acceptable errors in our word comparison to start a hit extension would take about the same speed but increase our sensitivity.
- Patterns - Basepairs at non consecutive positions
  - Patterns will increase the likelihood of *at least one* match within a long conserved region. The sensitivity/similarity graph of three different patterns can be seen in Figure 4.

6 FASTA

Another widely used heuristical local alignment algorithm is *FASTA*. *FASTA* [Pearson and Lipman 1988] uses a multistep approach to finding alignments. The algorithm first uses a lookup table to locate words of length k in both sequences and does the same hit extension step as *BLAST*. Then the algorithm checks to see if any ungapped regions can be joined by a gapped region. Finally, the sequences passing a certain threshold are run on one of the full dynamic programming algorithms presented previously. The details of the *FASTA* algorithm implementation are beyond the scope of this course.

7 Acknowledgements

Kleinberg and Tardos - Algorithm Design, 1st edition

Durbin, Eddy, Krogh, and Mitchison - Biological Sequence Analysis

Gusfield - Algorithms on Strings, Trees, and Sequences

Cormen, Leiserson, Rivest, Stein - Introduction to Algorithms, 3rd Edition

Altschul et al. - Basic local alignment search tool

Pearson and Lipman - Improved tools for biological sequence comparison