CS262 (Winter 2015)
Scribe: Kimberle Shen

Lecture 3: Sequence Alignment continued, BLAST

Last lecture: Aligning two strings (allowing gaps) to maximize the alignment score
This lecture: Algorithms to align strings in linear space and time
Next lecture: Burrows-Wheeler Transform (BWT)

I) Scoring gaps more accurately

Scoring function we have used so far:

Score F = (#matches)*m – (#mismatches)*s – (#gaps)*d

A caveat is that every gap is penalized the same. Take the following two alignments for instance:

TA - - - CG
TATATCG

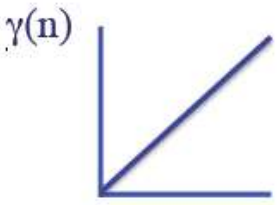
T - - A -CG
TATATCG

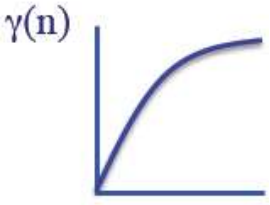
These two alignments have the same score, but the first one is biologically more probable because it means that the replication machinery slipped at one place by three nucleotides, rather than at two places by two and one nucleotides. Therefore, gaps usually occur in bunches.

IA) Convex gap scoring function

This gap scoring function penalizes every position of a gap less than the previous gap:

$\gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$ for all n

Previous model: 

Convex gap penalty function: 

We would thus need to know how long the previous gap is, and this can be computationally expensive.

The convex gap dynamic programming algorithm:

Initialization: same

Iteration:

$$F(i,j) = \max \text{ of } : \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\ldots i-1} F(k, j) - \gamma(i-k) \\ \max_{k=0\ldots j-1} F(i, k) - \gamma(j-k) \end{cases}$$

Termination: same

Running Time: $O(N^2M)$ (assume N>M)

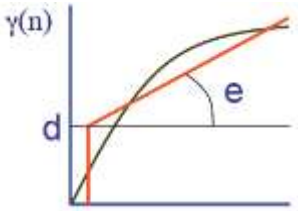
Space: O(NM)

IB) Affine gap penalty function

The affine gap scoring function is a compromise between the constant gap penalty function and the computationally-expensive convex gap penalty function. It is the most commonly used gap scoring function.

The affine gap penalty function assigns different penalties for opening (-d) and extending (-e) a gap:

$$\gamma(n) = d + (n-1)\times e$$



The motivation behind using this function is that it closely models evolution: To introduce the first gap, a break in the DNA has to occur. Multiple consecutive gaps are then more likely to occur after this event. Therefore, there is a fixed cost for opening a gap and a linear cost increment for increasing number of gaps.

To compute the optimal alignment, we need to remember the best score if the gap is "open" or if the gap is "not open". Therefore, for dynamic programming to work, we will consider both cases in separate tables.

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$, if x_i aligns to y_j [first case, gap is not open]

G(i, j): score if x_i aligns to a gap after y_j [second case, x has an open gap]

H(i, j): score if y_j aligns to a gap after x_i [third case, y has an open gap]

V(i, j) = best score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$

Assume that we know the optimal values up to i-1, then by inductive assumption, the rest will be correct. (The first row and the first column in the matrix is always correct.)

We need three matrices because we do not want to fix our decision to align x_i and yj as that affects the score of aligning x_{i+1} and y_{j+1}. For instance if a gap is assigned:

$G(i+1,j) = F(i,j) - d$ [meaning $G(i,j) < V(i,j)$]

But

$G(i+1,j) = G(i,j) - e$ [meaning $G(i,j) - e > V(i,j) - d$]

The affine gap dynamic programming algorithm:

Initialization: $V(i, 0) = d + (i - 1) \times e$
$V(0, j) = d + (j - 1) \times e$

Iteration:

$F(i, j) = V(i - 1, j - 1) + s(x_i, y_j)$

$G(i, j) = \max$ of: $V(i - 1, j) - d$
$G(i - 1, j) - d$

$H(i, j) = \max$ of: $V(i - 1, j) - d$
$H(i - 1, j) - d$

$V(i, j) = \max\{ F(i, j), G(i, j), H(i, j) \}$

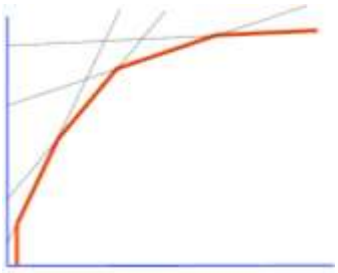
Termination:

V(i, j) has the best alignment

Running Time: O(NM)

Space: O(NM)

Generalization: For gap functions with repeated reduction of gap penalties (graph below), you will need another G and H for each additional linear segment.



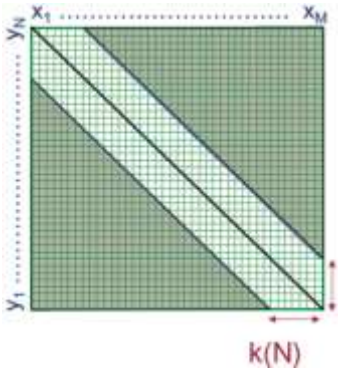
II) Bounded Dynamic Programming

Assuming that sequences x and y are similar, ie # gaps (x, y) < k(N),

Xi
|
Y_j

Xi matched to Y_j means that | i – j | < k(N).

Based on this assumption, we can perform bounded dynamic programing which allows us to align X and Y in a computationally more efficient way. If we know two segments that align perfectly, then we want to extend the alignment while not introducing too many gaps such that the alignment deviates too far from the diagonal.



The bounded dynamic programming algorithm:

Initialization:

F(i,0), F(0,j) undefined for i, j > k

Iteration:

For i = 1…M:

For j = max(1, i – k)…min(N, i+k):

$$F(i,j) = \max \text{ of } : \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same

Some properties of bounded dynamic programming:

- You cannot take the maximum from a “disallowed square”
- Easy to extend to affine gap case

III) Linear space

If we use Smith-Waterman algorithm to align a human and mouse genome, it will be too computationally expensive. To make such alignment practical, many techniques have been developed. BLAST is the most commonly used. The BLAST algorithm and program published in the Journal of Molecular Biology in 1990 [1] is one of the most highly cited papers in science and has unlocked the potential of computational genomics.

IIIA) Subsequence and substrings

A subsequence of a string is a sequence that can be derived from the string by removing certain elements but without changing the order of the rest of the elements.

A substring of a string is sequence that occurs in the original string. Therefore, a substring is always a subsequence but not necessarily the other way round.

A common subsequence (U) is a subsequence that can be derived from all strings X, Y..

What algorithm would you use for the common subsequence problem and what scoring functions?
Ans: Local alignment with gap penalty of 0 and mismatch score of x where x<0. Therefore, where there is a mismatch, it would be counted as gaps instead. The match score can then be any positive number and the resulting score will be simply the number of matches.

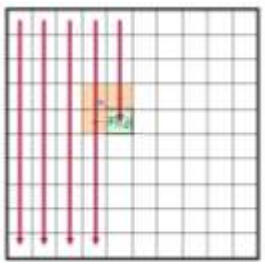
Common subsequence algorithm:

$$F(i,j) = \max \text{ of } : \begin{cases} F(i-1, j-1) + [1 \text{ if match, } 0 \text{ otherwise}] \\ F(i, j-1) \\ F(i-1, j) \end{cases}$$

Ptr(i, j) = (same as in N-W)

Termination: trace back from Ptr(M, N), and prepend a letter to u whenever
Ptr(i, j) = DIAG && F(i – 1, j – 1) < F(i, j)

It is easy to compute F(M,N) in linear space because you only need the previous row and column to calculate the score of a given cell in the matrix. You can thus delete rows and columns two rows/columns before and store only two rows/columns at any given time. The caveat is that going backwards to get the alignment will be a problem. Saving all the back-pointers to keep track of the alignment is out of the question because it will take quadratic space.



```
Allocate ( column[1] )
Allocate ( column[2] )

For   i = 1...M
  If    i > 1, then:
        Free( column[ i – 2 ] )
        Allocate( column[ i ] )
  For   j = 1...N
        F(i, j) = ...
```

IIIB) Hirschberg's algorithm

Hirschberg's algorithm solves this is linear space using a divide and conquer approach! Hirschberg's algorithm performs smaller alignment computations, each using only linear space. Compute the alignment up to the middle and iterate the procedure to the left and the right.

Notation:

- X^r is the reverse of X (not reverse complement)
- Fr(i, j): optimal score of aligning $X^r_1 \ldots X^r_i$ and $Y^r_1 \ldots Y^r_j$ = optimal score of aligning $X_{M-i+1} \ldots X_M$ and $Y_{N-j+1} \ldots Y_N$

General scheme:

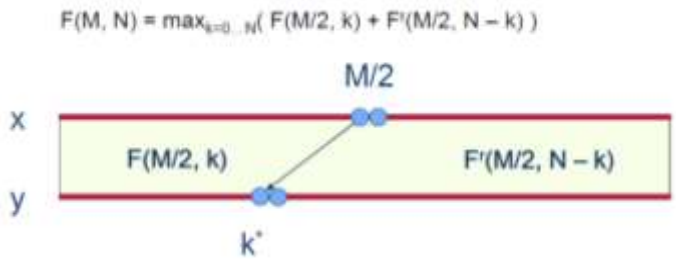
- Find the middle of X and see where it aligns in Y. Using only 2 columns, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N-k), meaning finding all the ways to align M/2 letters of X to one/two/three/etc letters of Y, doing the same for the last M/2 letters of X.
- Then, we can find k* which maximizes F(M/2, k) + F^r(M/2, N-k).
- M/2 aligns to either a letter or a gap (k*). The first i letters of X consumes the first k letters of Y. Back-pointers are also saved.
- Trace the path exiting column M/2 from k*.
- When you find the middle, you can de-allocate the space.
- Iterate the procedure to the left and to the right recursively (ie finding the middle again and again). There will be $\log_2 n$ levels of recursion, and no more than O(m) space will be used.
- Then combine by putting the two matrices together and add scores of diagonally connected elements*. The backpointers, which are kept, is used to trace the alignment.

*You are not allowed to connect horizontally because the letter k is already consumed; therefore you have to start at k+1. K cannot be used in both alignments.

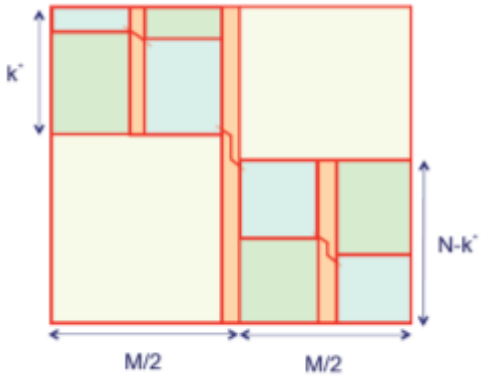
Lemmas:

1) $V(n, m) = \max_{0<*<m} [V(n/2, k) + V(n/2, m - k)]$
2) The position of k* can be found in O(nm) time and O(m) space. Moreover, a subpath $L_{n/2}$ can also be traced and stored in those time and space bounds.

F(M/2, k*) [left part]
F(M/2, N-k*) [right part]



0				M/2				M	M+1
k				*					
k+1					*				



Hirschberg linear-space algorithm:

MEMALIGN(l, l', r, r'): (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1. Let h = (l'-l)/2

2. Find (in Time O((l' – l) × (r' – r)), Space O(r' – r)):

Optimal path, Lh, entering column h – 1, exiting column h

Let:

k1 = position at column h – 2 where Lh enters
k2 = position at column h + 1 where Lh exits

3. MEMALIGN (l, h – 2, r, k1) [recursive call for left]

4. Output Lh

5. MEMALIGN(h + 1, l', k2, r') [recursive call for right]

Top level call: MEMALIGN(1, M, 1, N)

Time-Space analysis of the Hirschberg algorithm:

To compute optimal path at middle column:

Space: 2N (for size M x N)

Time: 2cMN (cMN for the first half. Subsequent steps take less than half, thus summation becomes 2cMN)

Total Time: O(MN)

Total Space: O(N) for computation, O(N + M) to store the optimal alignment

IIIC: Heuristic local aligners (partly covered during lecture)

NCBI blast allows you to take your query and align it to everything that has been sequenced so far, in a matter of minutes.

BLAST process: Query is done in linear time in the length of the query. Instead of aligning the whole sequence, an exact match is found with a k-mer (~10). This is the seed. It is then extended with dynamic programming left and right, until the score drops below a statistical threshold. All local alignments with a score above the statistical threshold are output.

References:

1. Altschul, Stephen; Gish, Warren; Miller, Webb; Myers, Eugene; Lipman, David (1990). "Basic local alignment search tool". Journal of Molecular Biology 215 (3): 403–410.