Lecture 9: Local Alignments & LCS

Study Chapter 6.4-6.8

1

Local vs. Global Alignment

- The Global Alignment Problem tries to find the longest path between vertices *(0,0)* and (*n,m*) in the edit graph.
- The Local Alignment Problem tries to find the longest path among paths between **arbitrary vertices** (*i,j*) and (*i′, j′*) in the edit graph.
- In the edit graph with negatively-scored edges, Local Alignment may score higher than Global Alignment

2

The Local Alignment Recurrence

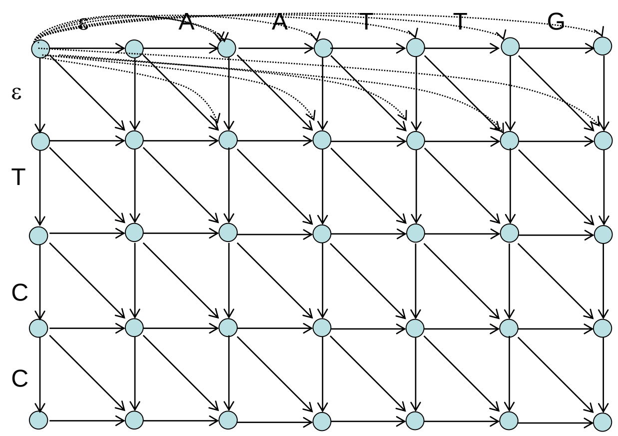
- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.
- Smith-Waterman local alignment
- The recurrence:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Power of ZERO: there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex

3

Smith-Waterman Local Alignment



4

An Example

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0												
2	C	0												
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	$s_{1,1}$											
2	C	0												
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	5	$s_{1,2}$										
2	C	0												
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	5	0										
2	C	0	0	$s_{2,2}$										
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

	0	G	C	T	G	G	A	A	G	G	C	A	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	0	0	5	5	0	0	5	5	0	0	0
C	0	0	10	3	0	1	1	0	0	1	10	3	0
A	0	0	3	6	0	0	6	6	0	0	3	15	8
G	0	5	0	0	11	5	0	2	11	5	0	8	11
A	0	0	1	0	4	7	10	5	4	7	1	5	4
G	0	5	0	0	5	9	3	6	10	9	3	0	1
C	0	0	10	3	0	2	5	0	3	6	14	7	0
A	0	0	3	6	0	0	7	10	3	0	7	19	12
C	0	0	5	0	2	0	0	3	6	0	5	12	15
T	0	0	0	10	3	0	0	0	0	2	0	5	17

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

	0	G	C	T	G	G	A	A	G	G	C	A	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	0	0	5	5	0	0	5	5	0	0	0
C	0	0	10	3	0	1	1	0	0	1	10	3	0
A	0	0	3	6	0	0	6	6	0	0	3	15	8
G	0	5	0	0	11	5	0	2	11	5	0	8	11
A	0	0	1	0	4	7	10	5	4	7	1	5	4
G	0	5	0	0	5	9	3	6	10	9	3	0	1
C	0	0	10	3	0	2	5	0	3	6	14	7	0
A	0	0	3	6	0	0	7	10	3	0	7	19	12
C	0	0	5	0	2	0	0	3	6	0	5	12	15
T	0	0	0	10	3	0	0	0	0	2	0	5	17

Match = 5, Mismatch = -4, Indel = -7

Local Alignment

```
G  A  A  G  -  G  C  A
|     |  |     |  |  |
G  C  A  G  A  G  C  A
```

6 matches: 6 × 5 = 30
1 mismatch: -4
1 indel: -7
Total: 19

Longest Common Subsequence

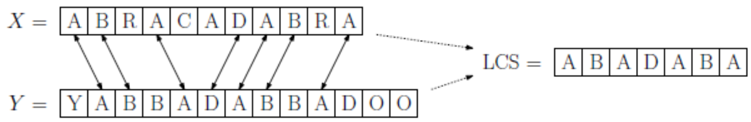
- A special case of edit distance where no substitutions are allowed
- A subsequence need not be contiguous, but order must be preserved
  - Ex. If v = ATTGCTA then AGCA and TTTA are subsequences of v, but TGTT and ACGA are not
- The length of the LCS, *s*, is related to the strings edit distance, *d*, by:

$$d(u,w) = len(v) + len(w) - 2\ s(u,w)$$

Longest Common Subsequence (LCS)

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le n)$ such that $Z = \langle x_{i1}, x_{i2}, \ldots, x_{ik} \rangle$.

For example, let X = <ABRACADABRA> and let Z = <AADAA>, then Z is a subsequence of X.



LCS Problem: Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ determine the length of their longest common subsequence, and more generally the sequence itself.

Brute Force for LCS

Brute Force: compare each subsequence of X with the symbols in Y

If $|X| = m$, $|Y| = n$, then there are 2^m subsequences of x; we must compare each with Y (n comparisons)

Running time: $O(n\,2^m)$

Rhodes College

DP Formulation for LCS

Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.

Subproblems: "find LCS of pairs of *prefixes* of X and Y"

A prefix of a sequence is just an initial string of values, $X_i = <x_1, . . . , x_i>$. X_0 is the empty sequence.

Let lcs(i, j) denote the length of the longest common subsequence of X_i and Y_j

Example: X_5 = <ABRAC> and Y_6 = <YABBAD>. Their longest common subsequence is <ABA>. Thus, lcs(5, 6) = 3.

Rhodes College

DP Formulation for LCS

Base Case: lcs(i, 0) = lcs(j, 0) = 0.

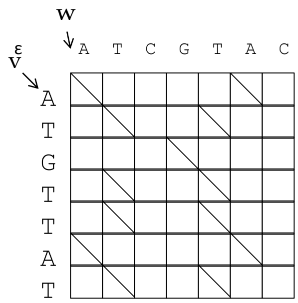
Last characters match: Suppose $x_i = y_j$. For example: Let X_i = <ABCA> and let Y_j = <DACA>. Since both end in 'A', it is easy to see that the LCS must also end in 'A'.

Last characters do not match: Suppose that x_i != y_j . x_i and y_j cannot both be in the LCS. Either x_i is not part of the LCS, or y_j is not part of the LCS (and possibly both are not part of the LCS).

- Option 1: (x_i is not in the LCS) - the LCS of X_i and Y_j is the LCS of X_i−1 and Y_j , which is given by lcs(i − 1, j).
- Option 2: (y_j is not in the LCS) - the LCS of X_i and Y_j−1, which is given by lcs(i, j − 1).

LCS as a Dynamic Program

- All possible alignments can be represented as a path from the string's beginning (source) to its end (destination)
- Horizontal edges add gaps in v. Vertical edges add gaps in w. Diagonal edges are a match
- Notice that we've only included valid diagonal edges in our graph

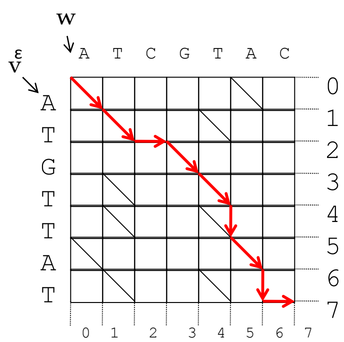


Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

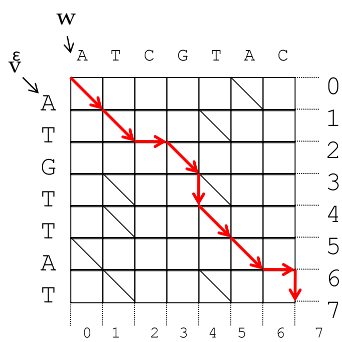
```
0 1 2 2 3 4 5 6 7 7
v A T _ G T T A T _
w A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7

Path:
 (0,0), (1,1), (2,2), (2,3),
 (3,4), (4,5), (5,5), (6,6),
 (7,6), (7,7)
```



Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

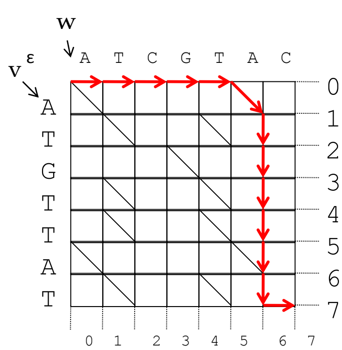



```
0 1 2 2 3 4 5 6 6 7
v A T _ G T T A _ T
w A T C G _ T A C _
0 1 2 3 4 4 5 6 7 7
```

```
Path:
 (0,0), (1,1), (2,2), (2,3),
 (3,4), (4,4), (5,5), (6,6),
 (6,7), (7,7)
```


Even Bad Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment




```
0 0 0 0 0 0 1 2 3 4 5 6 7 7
v _ _ _ _ _ _ A T G T T A T _
w A T C G T A _ _ _ _ _ _ _ T
0 1 2 3 4 5 6 6 6 6 6 6 6 7
```

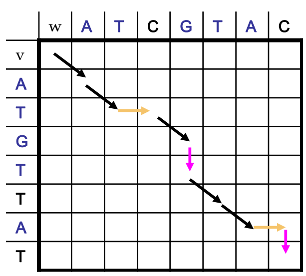
```
Path:
 (0,0), (0,1), (0,2), (0,3),
 (0,4), (0,5), (1,6), (2,6),
 (3,6), (4,6), (5,6), (6,6),
 (7,6), (7,7)
```

What makes a Good Alignment?

- Using as many diagonal segments (matches) as possible
- The end of a good alignment from (j...k) begins with a good alignment from (i..j)
- Set diagonal street weights = 1, and horizontal and vertical weights = 0

Alignment: Dynamic Program

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if \ v_i = w_i \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$



Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0							
T	0							
G	0							
T	0							
T	0							
A	0							
T	0							

Initialize *1st* row and *1st* column to be all zeroes.

Or, to be more precise, initialize *0th* row and *0th column to be all zeroes.*

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \;\; (\text{diagonal}) \\ s_{i-1,j} & (\text{down}) \\ s_{i,j-1} & (\text{right}) \end{cases}$$

w = ATCG-TAC-
v = AT-GTTA-T

LCS Code

```
def LCS(v, w):
    s = [[0 for i in range(len(w)+1)] for j in range(len(v)+1)]
    b = [[0 for i in range(len(w)+1)] for j in range(len(v)+1)]
    for i in range(1,len(v)+1):
        for j in range(1,len(w)+1):
            if (v[i-1] == w[j-1]):
                s[i][j] = max(s[i-1][j], s[i][-1], s[i-1][j-1]+1)
            else:
                s[i][j] = max(s[i-1][j], s[i][j-1])
            if (s[i][j] == s[i][j-1]):
               b[i][j] = 1  →
            elif (s[i][j] == s[i-1][j]):
               b[i][j] = 2  ↓
            else:
               b[i][j] = 3  ↘
    return (s[i][j], b)
```


Backtracking Code

```
def PrintLCS(b,v,i,j):
    if ((i == 0) or (j == 0)):
        return
    if (b[i,j] == 3):
        PrintLCS(b,v,i-1,j-1)
        print v[i-1],
    else:
        if (b[i,j] == 2):
            PrintLCS(b,v,i-1,j)
        else:
            PrintLCS(b,v,i,j-1)
```

Alignment: Backtracking

Arrows show where the score came from.

- ↓ if from the top
- → if from the left
- ↘ if $v_i = w_j$

Our table only keeps track of the longest common subsequence so far. How do we figure out what the subsequence is?

We'll need a second table to keep track of the decisions we made… and we'll use it to backtrack to our answer.

Changing the Scoring

- Longest Common Subsequence (LCS) problem
  - the simplest form of sequence alignment
  - allows only insertions and deletions (no mismatches).
- In the LCS Problem, we scored 1 for matches and 0 for indels
- Consider penalizing indels and mismatches with negative scores
- Simplest *scoring schema*:
  - *+1* : **match premium**
  - *-μ* : **mismatch penalty**
  - *-σ* : **indel penalty**