Richard Chen

Homework 4

1. Say we are using dynamic programming to find approximate occurrences of P (|P | = n) in T (|T | = m) using the edit-distance-like method we discussed in lecture.

a. (1 pt) Exactly how many dynamic programming matrix elements do we have to fill in, including the first row and column? Don't forget the empty prefixes.

We would need to to fill in (m+1)(n+1) matrix elements. Including the empty prefix character, the length of P and T is n+1 and m+1 respectively, making the dimension of the matrix size (m+1)(n+1). Since dynamic programming fills in every possible cell in the matrix (*O(mn)*), all (m+1)(n+1) cells need to be filled to yield global alignment value.

b. (1 pt) How do we initialize the first row and column?

We would initialize the first row with 0's, rather than increasing integers, then fill the matrix. The first column would be filled with increasing integers from 0 to n, where |P| = n.

c. (3 pts) Say we were interested only in the best approximate occurrence (the one with the fewest mismatches and gaps) of P in T. Describe briefly how we would find it given the filled-in matrix, and how we would find where the mismatches and gaps in the alignment are? Assume there is no tie.

To find the best approximate occurrence of P in T in a filled matrix D, we would pick the lowest edit distance in the bottom row, and trace diagonally to the top left until we reach the top row. This edit distance D[i,j] represents the optimal edit distance between the length-I prefix of P and a substring of Tending at position j. We trace diagonally because in our scoring matrix *s*, mismatches (transitions and tranversions) are preferred to gaps in P and T. In our traceback, horizontal and vertical moves are immediately associated with gaps, and diagonal moves with changes in the edit distance are associated with mismatches.

2. Local Alignment Matrix

a. (4 pts) Fill in this local alignment matrix. Rows are labeled with characters from X and columns are labeled with characters from Y . Use a scoring function where matches get a bonus of +1, and mismatches, insertions and deletions get a penalty of -1.

	ε	A	T	A	G	C
ε	0	0	0	0	0	0
G	0	0	0	0	1	0
T	0	0	1	0	0	0
A	0	1	0	2	1	0
T	0	0	2	1	1	0
G	0	0	1	1	2	1
C	0	0	0	0	1	3

b. (2 pts) What are the substrings of X of Y with maximum global alignment value and what is that global alignment value?

The maximum global alignment value is 3. Thee are two pairs of substrings of X of Y that tie for the maximum global alignment value:

X = TATGC Y = TAGC
X = ATGC Y = ATAGC

3. (26 pts) Solve the problem here: http://bit.ly/CG_UnpairedAsmChallenge. Please format your solutions exactly as described on that page. Ask if you have any questions about the format.

a. Submit your overlap graph as a file named overlaps.txt.

See Code.

b. Submit your unitigs as a file named unitigs.txt.

See Code.

c. Submit your assembled genome as a file named solution.fa.

See Code.

4. (5 pts) In lecture, we saw an example where Greedy-SCS collapsed a repeat. The input consisted of all length-6 substrings of *a_long_long_long_time*. The output was:

a_long_long_time

When we tried length-8 substrings instead, Greedy-SCS got the correct answer. Here's another input:

to_every_thing_turn_turn_turn_there_is_a_season

When I run Greedy-SCS taking length-6 substrings, the repeat is collapsed:

to_every_thing_turn_turn_there_is_a_season

When I try length-8 substrings, the repeat is still collapsed. Why doesn't using length-8

substrings fix the problem in this case? Would using length-9 substrings fix the problem? Why or why not?

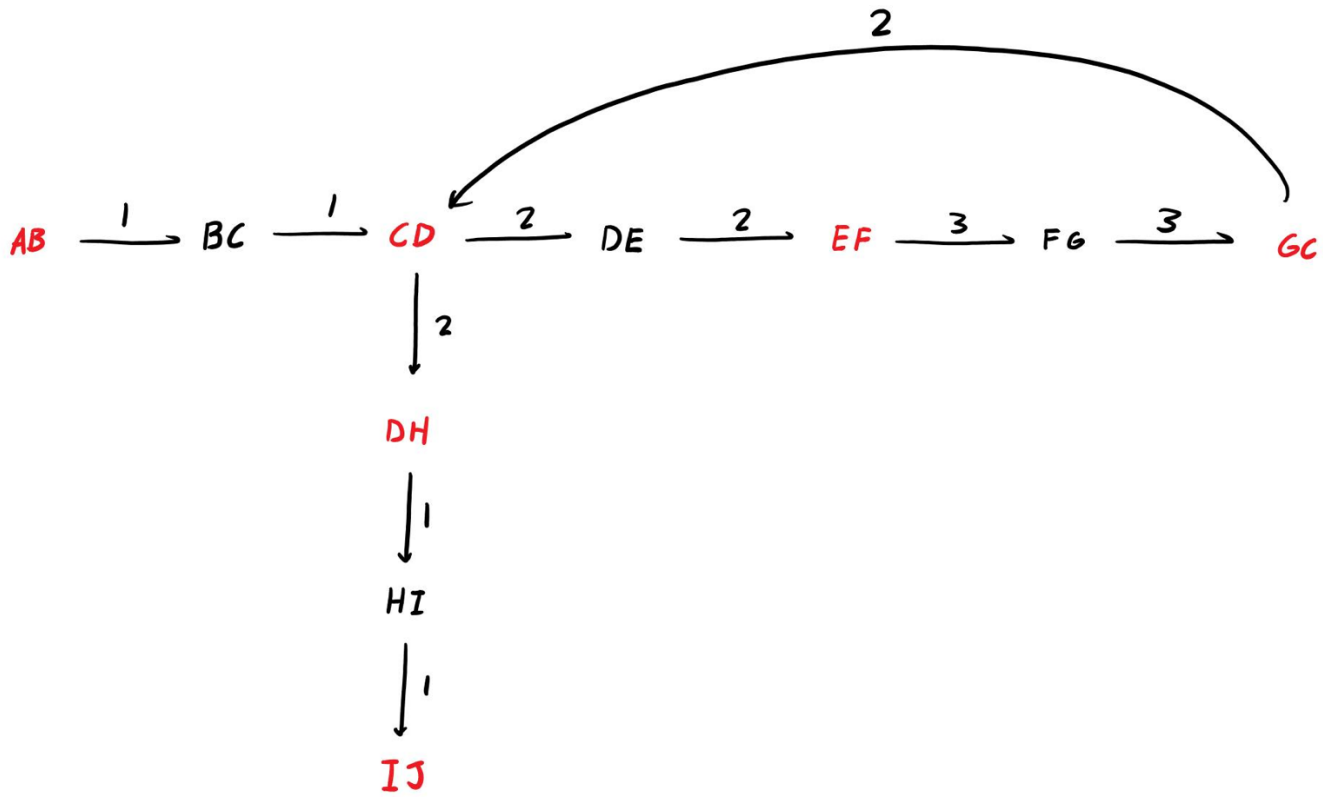
The 3rd law of assembly states that repeats make assembly difficult, which is the case in using Greedy-SCS on *a_long_long_time* and *to_every_thing_turn_turn_turn_there_is_a_season.* In the example in the lecture, using 6-mers collapsed the tandem repeat *long*, whereas using 8-mers produced a unique substring *g_long_l* that spanned across all 3 repeats of *long,* and produced the correct string. The string *to_every_thing_turn_turn_turn_there_is_a_season* runs into this same problem. There is not enough coverage using 8-mers that distinguishes the number of repeats of *turn*. 9-mers would fix this problem, as using 9-mers produces a unique substring *n_turn_tu* that spans across all three repeats of *turn.*

5. **We discussed De Bruijn graph assembly in class. Say we have the following input reads:**

ABCDEFGC
EFGCDHIJ
CDEFGCDH

And the De Bruijn graph k-mer length = 3. Edges correspond to k-mers and nodes to k-1-mers.

   a. **(2 pts) Draw a De Bruijn graph for this data set. Draw one weighted edge per distinct k-mer, with weight equal to the number of times the k-mer occurs in the input.**



b. (2 pts) Prove that the graph either is or is not Eulerian.

From *Jones and Pevzner,* a directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced. A node is balanced if indegree equals outdegree, and semi-balanced if indegree differs from outdegree by 1. In the De Brujin Graph I drew in a), there are a total of 6 semi-balanced nodes (shown in red): AB, IJ, DH, CD, EF, and GC. Thus, the graph is not Eulerian.

c. (2 pts) Give an example of a walk through the graph that traverses three nodes and spells out a 4-mer that does not appear in any of the input reads. (This is an example of how De Bruijn graphs lack "read coherence".)

An example of a walk through the graph that traverses three nodes and spells out a 4-mer that does not appear in any of the input reads is BC -> CD -> DH. This walk spells out BCDH, which is not contained in *ABCDEFGC, EFGCDHIJ, or CDEFGCDH.*