Chris Bohlman

April 17, 2018

CSc 445

Algorithms Homework #5

1. LCS("ABCD","AABBCC"):

		A	A	B	B	C	C
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
B	0	1	1	2	2	2	2
C	0	1	1	2	2	3	3
D	0	1	1	2	2	3	3

At max, length of LCS(x, y) = 3

But how many are at length 3? We can trace the matrix backwards in several ways, by starting at the cell in the bottom left corner c[n, m] (highlighted), and moving back towards cells (either c[i-1, j], c[I, j-1], or c[i-1, j-1]) that have a value in them less than or equal to the current cell's value.

Possible subsequences, based on "AABBCC"

First A, first B, first C

First A, first B, second C

First A, second B, first C

First A, second B, second C

Second A, first B, first C

Second A, first B, second C

Second A, second B, first C

Second A, second B, second C

Total of 8 different alignments of length 3

2. Make array c of objects. These objects store weight of longest common subsequence up to that point. Initialize array c the same way as before, the weight of LCS 0 on the borders. Iterate through for loops the same way as in LCS, except:

```
If x[i] = y[i],
        c[i, j]->weight = c[i-1,j-1]->weight+ x[i]->weight
Else
        c[i,j]->weight = max{c[i-1,j]->weight, c[i, j-1]->weight}
```

This will ensure common sequence up to that point is primarily determined by the weights of said sequence instead of it's length. To get the weightiest subsequence, start from c[m, n] and iterate backwards through array:

If x[i] = y[j], add adjacent character to sequence and move to c[i -1, j-1].

Else, move to cell which has higher value of weight, between c[i-1, j] or c[i, j-1].

Since the for loop goes from I = 1 to n and inside from j = 1 to n, this algorithm will run in O(n^2) time.

3. To find if string x is a subsequence of string y:

Start at leftmost character of both of strings (x[i=0] and y[i=0]).

if x[i] = y[i], move forward one character in both strings

if x[i] != y[i], move forward one character in string y

Re-compare current characters recursively. If you reach the end of string x, then x is a valid subsequence of y. If you reach the end of string y only, then x is not a valid subsequence of string y.

Running time: In worst case, you could completely iterate through string x and string y, so the running time would be O(|x|+|y|).

4. Ed("HAEALALAO", "HELLO")

		H	E	L	L	O
	0	1	2	3	4	5
H	1	0	1	2	3	4
A	2	1	1	2	3	4
E	3	2	1	2	3	4
A	4	3	2	2	3	4
L	5	4	3	2	2	3

A	6	5	4	3	3	3
L	7	6	5	4	3	4
A	8	7	6	5	4	4
O	9	8	7	6	5	4

From the array, we need to construct the optimal sequence of edit distances.
Start at c[i,j] = c[9, 5]
If letters at x[i] and y[j] are the same, (x[9] and y[5] = O), move to c[i-1, j-1], or in this case, c[8, 4]. This means keeping the letter the same at x[i] and y[j].
If letters at x[i] and y[j] are not the same, look at the three squares c[i-1, j], c[i-1, j-1], and c[i, j-1], and move to the minimum of these squares.
c[i-1,j-1] means swapping letter, c[i-1, j] means deletion of letter, and c[i, j-1] means insertion.
Use these rules to determine what to do to get the optimal subsequence and reconstruct the steps of the edit distance in that way. In this case, it would be:
Keep O
Delete A
Keep L
Delete A
Keep L
Delete A
Keep E
Delete A
Keep H
For a total of 4 operations (all deletions from X).

5. Fréchet(P, Q) is smallest value of r for which Fréchet(P, Q) = YES. r represents the largest distance between any two vertexes along the shortest path in the DTW of two lines. This is because r in the Fréchet distance problem has to be a minimum length for the problem to work out. Since dynamic time warping returns smallest sum over 2n vertexes, it makes sense that the minimum Fréchet distance (r) would be at least the same size or larger than all of the lines added up in DTW. Since r is the maximum the distance leash has to be between 2 points in order to traverse the lines, we can say that at maximum, the answer for DTW has to be 2n of these distances. Anything below that, and the DTW length returns something less than 2n*r. Therefore, 2n*r >= DTW

6. Consider the idea that the algorithm only needs to hold 2 rows (or columns) of edit distances at one time. Let's assume that min(|X|, |Y|) happens to be string Y. Fill in the first row of c according to the algorithm from before, where the row will look like 0, 1, 2, 3,…,n. Now, move onto the row underneath. Start at c[1, 0] with the same relaxation

step as before, except now, if the square does not exist (i.e. c[1, -1]), ignore that possibility. You can look up a square at the previous row; that's why we're holding it. Fill in that row from 0 all the way to n, and then delete the first row. Move down another letter, and do this until the entire string has been analyzed. Through this method, it is impossible to completely trace back the edit distance operations, but to prove the space complexity, you need 2 |X| arrays at max to hold the information you need., making the space complexity O(|X|), meaning that it's also equal to O(min(|X|,|Y|).

7. Use LCS code to start off with, expect co-opt it for segments.

```
//X and Y are arrays of objects, storing label and segment number
C = [0..m, 0..n]
   for i = 0 to m
      C[i,0] = 0
   for j = 0 to n
      C[0,j] = 0
   for i = 1 to m
      for j = 1 to n
         if X[i].label = Y[j].label
            if (|Y[j].segNum – X[i].segNum| <= L(a number))
               C[i,j] = C[i-1,j-1] + 1
         else
            C[i,j] = max(C[i,j-1], C[i-1,j])
   return C[m,n]
```

Time for algorithm: O(n^2)

Proof:

This modified algorithm, is based off of LCS. In effect, what you would want to do is find the longest common subsequence of these 2 lines (Hence, points being connected if they have the same label). In addition, segments don't cross, since this is based off of LCS. However, there are rules added, specifically the rule regarding the length of segments. Therefore, the if statements when two letters match (an absolute value of their segment numbers) fails if it is over value L. Finally, to prove time complexity, there are two nested for loops, so this algorithm would be O(n^2).

8. So basically, with this problem, you want to find the smallest value of r that makes the Fréchet algorithm output a yes. The hint being that there exists two points that output the correct distance r*.

   To start off with, what if you got the distance r between point p_i and q_j, see if that r value outputted a yes for the Fréchet distance, and stored it if it was the smallest value up to that point. Iterate through all points. The time complexity ends up being:

```
Double smallest_r = infinity
For every point in p
    For every point in q
        R = |pi – qj|
        If (Frechet(P, Q, r) == true && r < smallest_r)
            Smallest_r = r
```

   Time complexity = O(n^2) * Fréchet = O(n^2) * O(n^2) = O(n^4)

   But we can do better

   What if we did a sort of reverse merge sort?

   Get all the values for r from all pairs of points, and sort them by distance:

   O(n^2) + O(nlogn)

   Start with the r in the middle of the group. If Fréchet(P, Q, r) = false, your r is too small. Go to the halfway value of this r and the r at the end and repeat.

   If Fréchet(P, Q, r) = true, your r might be too large. Go to the r in the middle of the first r and this one and repeat.

   Eventually, you will arrive to the correct r.

   Time complexity: Since you're splitting through halves until you get to the correct r, you are performing O(logn) Fréchet operations. So this time complexity = O(logn) * O(n^2) =

   O(n^2*logn), which is the correct answer to the problem.

9. Assuming you create a table F[1..m,1..n], where each cell is either blue or red: F[I,j] is blue is $|p_i-q_j|$<= L. Otherwise, F[I,j] is red. For a necessary condition for the weak Fréchet distance algorithm to return true, there needs to be a path from p_1 and q_1 to p_m and q_n. Therefore, there must be an unbroken line of adjacent blue squares through array F. Since only one thing (person or dog) is moving at any iteration, diagonal spaces wouldn't count. Therefore, to find whether there is a sequence of jumps for which the distance between person and dog is always <= L, you would build the array F, and then do a DFS on the array to see if there is an unbroken blue path from p_1 and q_1 to p_m and q_n (You can move backwards, we're talking about *any* adjacent cells). If you reach a square that has either had everything adjacent to it explored or is surrounded by red cells, mark it as visited. Algorithm returns true if there exists such a path.

10. To minimize the sum of all leashes used in all time stamps, first, construct a table F[1..m,1..n], where each cell holds the length of $|p_i-q_j|$. Now, modify Dijkstra's algorithm using a min heap as a priority queue. Instead of using edge weights as the modifiers for array d[], use the values of $|p_i-q_j|$ at each vertices (In addition, all edges point to just the adjacent cells, similar to the previous problem). Start Dijkstra's at p_1 and q_1, and continue to p_m and q_n. This will give you the minimum sum leash path between the verticies, since Dijkstra will find the absolute shortest path (and therefore the shortest sum) between points.
The running time for this is: O(n^2) for the construction of the graph
+ O((V + E)log(n) for Dijkstra's algorithm.
Since there are n*m <= n^2 vertices and >4n^2 edges for this grid, full running time is = **O(n^2 * logn)**.