Assignment 3

CS110 - Professor Stern, P.

Ash Nguyen

2018/12/06

1 Longest Common Subsequence Of DNAs

The length of the longest common subsequence of two strings X and Y can be determined by a recursive algorithm with memoization: a dynamic program. It is implemented as followed:

```
import pandas as pd
import numpy as np

def LCS(X, Y):
    m = len(X)
    n = len(Y)
    C = np.zeros((m+1,n+1),dtype=int)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                C[i][j] = C[i - 1][j - 1] + 1
            else:
                C[i][j] = max(C[i][j - 1], C[i - 1][j])
    return C

def lenLCS(dna):
    l=len(dna)
    tab=np.zeros((l,l),dtype=int)
    for _ in range(l):
        for i in range(l):
            C=LCS(dna[_][1],dna[i][1])
            tab[_,i]=C[len(dna[_][1])][len(dna[i][1])]
    return pd.DataFrame(tab)

DNAs=[
(0,'TTCTACGGGGGGAGACCTTTACGAATCACACCGGTCTTCTTTGT
TCTAGCCGCTCTTTTTCATCAGTTGCAGCTAGTGCATAATTGCTCACAAACGTATC'),
(1,'TCTACGGGGGGGCGTCATTACGGAATCCACACAGGTCGTTATGTT
CATCTGTCTCTTTTCACAGTTGCGGCTTGTGCATAATGCTCACGAACGTATC'),
(2,'TCTACGGGGGGGCGTCTATTACGTCGCCAACAGGTCGTATGTTCA
TTGTCATCATTTTCATAGTTGCGGCCTGTGCGTGCTTACGAACGTATTCC'
(3,'TCCTAACGGGTAGTGTCATACGGAATCGACACGAGGTCGTATCT
```

```
TCAATTGTCTCTTCACAGTTGCGGCTGTCCATAAACGCGTCCCGAACGTTATG'
 (4, 'TATCAGTAGGGCATACTTGTACGACATTCCCCCGGATAGCCACTT
TTTTCCTACCCGTCTCTTTTTCTGACCCGTTCCAGCTGATAAGTCTGATGACTC'
 (5, 'TAATCTATAGCATACTTTACGAACTACCCCCGGTCCACGTTTTTC
CTCGTCTTCTTTCGCTCGATAGCCATGGTAACTTCTACAAAGTTC')
 (6, 'TATCATAGGGCATACTTTTACGAACTCCCCCGGTGCACTTTTTTC
CTACCGCTCTTTTTCGACTCGTTGCAGCCATGATAACTGCTACAAACTTC')]

lenLCS(DNAs)
```

The function $lenLCS()$ print out a table of the lengths of the longest common subsequence between any pair of the 7 DNA strings. The output table is:

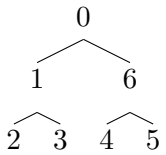
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	100	82	73	72	72	70	80
1	82	96	83	81	67	65	70
2	73	83	94	73	62	61	67
3	72	81	73	97	62	60	63
4	72	67	62	62	98	71	82
5	70	65	61	60	71	89	79
6	80	70	67	63	82	79	94

The complexity of the LCS procedure is $O(M^2)$ (M is the maximum length of the strings) for each pair of strings, M and K being the length of the pair of strings. If we are calculating for N genes, with pairwise LCS, we strictly has a time complexity of $O\left(M^2 \frac{N!}{2!(N-2)!}\right)$ because we have such many combinations of two strings for N number of strings, and we only need to calculate the LCS once, assuming M is the maximum length of each string. We also has a space complexity of $O(M^2)$, since every time we need a $M \times M$ table.

We know that from a string of DNA, there a small probability of either inserting, deleting or mutation randomly any of its elements. Therefore, the longest common subsequence between any pair of DNAs tells us the *similarity level* between the two pairs: the longer the common subsequence, the more the pair are *closely related*. This is because the longest common subsequence indicates the longest string of DNA that is similar to both strings in the pair. By manually examine the table above, we can induce that the original string is 0, which gave rise to 1 and 6 because they are the most similar to 0. Next, the 1 string probably gave rise to 2 and 3 judging by their similarities, and the 6 string probably gave rise to 4 and 5. More formally though, we should examine all the possible starting case, as in presume that each of the 7 strings can be the original parent. However, since there is a small probability that one string will insert/delete/mutate to turn into another string, and the original string does so independently (because the changing process of each element in each time is independent) twice, the brother/sister strings will only be "similar" to theirs parent only, not with each other.

Because each string gave rise to two children, we can deduce that the original string must be very similar to exactly two strings, since the probability is small and we will not expect the children of the children to somehow, randomly, be more similar to the children themselves to the original parent. Each of the two children of the original parent will be very similar to exactly three strings: the original parent and the two children it gave rise to, since two brother/sister strings are not expected to be very similar. Finally, the final descendants: four children of the children of the original parent can be very similar to exactly one other string: their respective parents.

Using these arguments, we can see that indeed the 0 string has only two very similar string that stand out, which is 1 and 2 and the respective length of LCS of 82 and 80, compare to the other ranging from 70-73, which is what we expect from the original parent. Each of the string 1 and 6 has three very similar strings, 0,2,3 and 0,4,5 respectively, which is what we expect from the middle generation. And finally, each of the strings 2,3,4,5 are very similar only to their respective parent, which are 1 and 6, and this is what we expect from the last generation. The relationship is expressed in the following tree diagram:



We can further test our argument by try printing out all possible longest common subsequences for every pair of strings. Since the similarity between the two strings will determine how many possible LCSs they can be, we can automate this process by the following code:

```
#Adapted from the backTrackAll() function in our discussion
#about palindromes
def PossibleLCS(C, X, Y, i=None, j=None):
    if i is None:
        i = len(X)
    if j is None:
        j = len(Y)
    if i == 0 or j == 0:
        return set([""])
    elif X[i - 1] == Y[j - 1]:
        return set([Z + X[i - 1] for Z in PossibleLCS(C, X, Y, i - 1, j - 1)])
    else:
        R = set()
        if C[i][j - 1] >= C[i - 1][j]:
            R.update(PossibleLCS(C, X, Y, i, j - 1))
        if C[i - 1][j] >= C[i][j - 1]:
            R.update(PossibleLCS(C, X, Y, i - 1, j))
        return R
```

```
19
20 def PosLCS(dna):
21     l=len(dna)
22     tab=np.zeros((l,l),dtype=int)
23     for _ in range(l):
24         for i in range(l):
25             C=LCS(dna[_][1],dna[i][1])
26             tab[_,i]=len(PossibleLCS(C,dna[_][1],dna[i][1]))
27     return pd.DataFrame(tab)
28
29 PosLCS(DNAs)
```

For example, the number of possible LCSs between 0 and 1 is 40, and between 0 and 6 is 2, which are very small compared to between 0 and 5, which is 336. We then know that 0 is more closely related to 1 and 6 than to 5, as we argue before using the length of the LCS. Similar arguments can be made that lead us to the above tree diagram.

2 Relationships Between DNAs And Mutation Probability

2.1 Mutation

First, let us consider how to define the difference between the two strings. From the LCS of any pair of strings, we can calculate the sum of absolute differences between the initial string-LCS and the transformed string-LCS by using the length distance. Let D be the distance between two string A and B.

$$D = |len_A - len_{LCS_{AB}}| + |len_B - len_{LCS_{AB}}|$$

By this mean of calculation, we are getting the total number of insertion, deletion and changing, with the changing counts twice. This is a valid position, as we can understand that changing equals a deletion AND an insertion, so it worths twice as much as a regular insertion/deletion. Specifically, the changing operation should be counted twice because if it is not, then there will be no insertion/deletion because changing can be used to replace them both (assuming that the cost of each operation is identical, and nature tends to use the path of least resistance (or least cost), it should always be better off with just changing rather than insertion or deletion). Thus, we will count changing twice.

Nevertheless, the quantity of interest here is the difference between LCS and the resulted string after transformation, which is the second term on the right hand side of the above equation. The following code count the difference between each pair of strings and print out a table:

```
1 def diff(dna):
2     l=len(dna)
3     tab=np.zeros((l,l),dtype=int)
4     for _ in range(l):
```

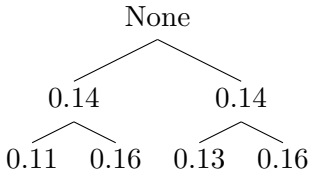
```
        for i in range(l):
            C=LCS(dna[_][1],dna[i][1])
            a=C[len(dna[_][1])][len(dna[i][1])]
            tab[_,i]=abs(len(dna[i][1])-a)
    return pd.DataFrame(tab)

diff(DNAs)
```

The resulted table is:

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	14	21	25	26	19	14
1	18	0	11	16	31	24	24
2	27	13	0	24	36	28	27
3	28	15	21	0	26	29	31
4	28	29	32	35	0	18	12
5	30	31	33	37	27	0	15
6	20	26	27	34	16	10	0

Divide these differences with the total length of the original string before transformation we will have the approximate probabilities of undergoing a transforming operator:



So the probability of transforming is around 0.14%. But this is only an estimation, since the distance between the LCS and the resulted string is only an approximation of the number of transformation, not the actual number of transformation (due to the interchangability of insertion/deletion and changing). Therefore, we need more data to average out the probability in each generation. We are interested in the true probability of transformation, and the best estimate we can get is the true mean of the probability. Since each transformation between a string to the next is independent (the probability remains unchanged), we can treat each of the probability computed as a independent random variable. The Central Limit Theorem tells us that if we repeatedly take the average of the calculated probability in each generation, the averages will form a normal curve, and the mean of the normal curve is our closest estimate of the true mean.

2.2 Inferring Relationships

In the general case where the number of DNA are larger than 6 and we cannot manually examine the distance table to figure out which DNA strings are related to which,

we can use Kruskal's Minimum Spanning Tree algorithm to try to estimate this relationship. Let DNA strings be vertices in a graph, and the distance between the LCS and the resulted string is the weights of the edges connecting two vertices. We define a minimum spanning tree is a path that connect all the DNA strings, using the minimum edge's weights. Kruskal's algorithm is a greedy algorithm that follows these procedures:
1. Initialize an empty list of "used" edge.
2. Choose the edge with the lowest weight in the graph and is not in the "used" list.
3. Check if the edge, along with the other edges in the "used" list, form a circuit. If it does, discard the edge and go back to step 1 to choose the next lowest weight edge.
4. Recorded the edge as "used".
5. Go back to step 1 until all vertices are connected.

We will now prove that Kruskal's greedy algorithm will lead us to the global optimum.

Clearly, the problem of minimum spanning tree has optimal substructure: suppose we choose a random vertex to start with (the choice clearly does not matter, since every vertices need to be connected anyway), we can evaluate each of the choice of edge connecting the current vertex to the others by a recursion. This leave us with a subproblem: finding the minimum spanning tree for the graph, without the current vertex and its edges (since we already choose the optimal edge for the current vertex). This is clearly a optimal substructure. Of course, the subproblems are overlapping. Therefore we can use a dynamic program to solve this problem. But the problem of minimum spanning tree also has greedy property. Suppose for the sake of contradiction that the optimal solution does not consist of a minimum edge connecting vertex A with any other vertices, edge λ_1. That means we can find another path to A that has lower weight than the minimum weight, $\lambda_2 + ... + \lambda_n < \lambda_1$, with λ_2 is the edge connecting A with some other vertex (this edge must exist, otherwise our path won't be connected to A). But since λ_1 is the minimum edge connecting A with any other vertices, $\lambda_1 < \lambda_2$. This lead to a contradiction, assuming (validly) that the weights are all positive (because the distance between the LCS and a string cannot be negative). Therefore the greedy choice of the minimum weighted edge will lead us to the global optimum.

Implementing the Kruskal's algorithm in our current example of 7 strings:

```
from scipy.sparse import csr_matrix as mat
from scipy.sparse.csgraph import minimum_spanning_tree as mst

X = mat([[ 0, 14, 21, 25, 26, 19, 14],
         [18,  0, 11, 16, 31, 24, 24],
         [27, 13,  0, 24, 36, 28, 27],
         [28, 15, 21,  0, 36, 29, 31],
         [28, 29, 32, 35,  0, 18, 12],
         [30, 31, 33, 37, 27,  0, 15],
         [20, 26, 27, 34, 16, 10,  0]])

results=mst(X)
```

```
results.toarray()
```

We have the following results:

```
[Output]:
array([[  0.,  14.,   0.,   0.,   0.,   0.,  14.],
       [  0.,   0.,  11.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,  15.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,  12.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,  10.,   0.]])
```

The resulting graph (plot in R) of minimum spanning tree (in red) is shown in figure 1. Clearly we can see 0 is connected to 1 and 6, 1 is connected to 2 and 3 while 6 is connected to 4 and 5. As presented, it is easy to know which one gave rise to which: notice that 0 has two edges in the MST, 1 and 6 each has three while 2,3,4,5 each has only 1. These results are to be expected with the original parent, the first generation and the second generation.

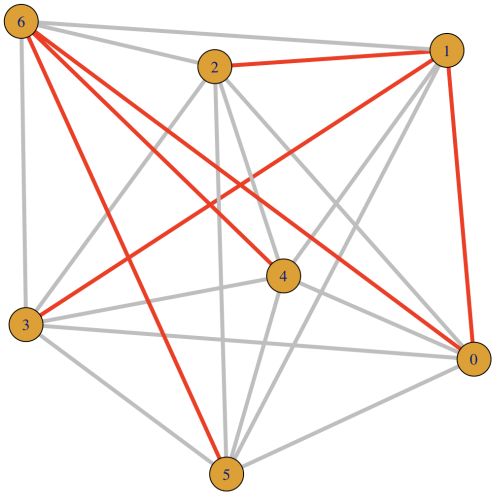


Figure 1: Minimum Spanning Tree By Kruskal's Algorithm

2.3 Practical Considerations

For real gene sequences, between the original parents (P) and the subsequent generations (F_1, F_2,...,F_n) beside the random mutation as we discussed in this assignment there will also be differences due to the nature of genetic reproduction: the parents each will donate half of their gene sequences to the child, creating in differences in

brother/sister genes even without mutations. According to Mendel's theory of genetic inheritance, depends on the characteristics of the species, each cluster of DNA sequence will specify different aspects of the gene and depends on how these DNAs of the parents interact, the following generation might have different DNA sequence indicating different characteristics with independent probabilities. This further complicates our analysis.