Statistical computing HW5

Jeffrey Lee
UWNetID: 1826649

Problem 1

(a)

```
Breiman
Best
Baxter
Barnett
None
Lotka
Finlaison
Hogben
[Finished in 0.2s]
```

(b)

Last time I implement as list to form the nodes. This will cause a problem since I have to check if the value is inside the list. This is acceptable for a small sample like last time since it's still polynomial O(n) to check the list. This time I implemented as dictionary and set the key as index, value as nodes. Also, I set a variable to store the name index when reading through the file. This way, I can take out the node data easily by simply use the index in the dictionary, which will only take O(1).

Make-Set: This is for initializing the node as a set when it first comes out. I simply set its parent and representative as itself. Rank is 0. This will take O(1) to initialize.

Find_Set: This function is for finding the representative of the set; therefore, when it's not the representative, it will recursively call the function to find it. Worst case is when it form a super unbalance tree and will take O(n) to climb to the top.

Union: This is to form the tree by link the representatives. Therefore, we have to find the representative by Find_Set function and link it together. In link function, we compare the rank of the representatives, the node with larger rank becomes the parent. When the rank is equal, we arbitrary choose a node as parent. In my implementation, I choose node2 as node1's parent, which means node1 points to node2. Those comparison of values and assignment operation will take O(1). Finally, we return the union and that's how we get the forest.

Problem 2

(a) Thm 15-1 X = <X_1, X_2,.... X_m>, Y = <Y_1, Y_2,.... Y_n>, Z = <Z_1, Z_2, ...Z_k>

1. If X_m = Y_n, then Z_k= X_m = Y_n, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If X_m != Y_n, then Z_k!= X_m implies that Z is and LCS of X_{m-1} and Y.
3. If X_m != Y_n, then Z_k!= Y_n implies that Z is and LCS of X and Y_{n-1}.

In the first description, if the last char of X and Y are the same, we use copy and the subproblem remained is converting from X_{m-1} to Y_{n-1}. And the optimal solution of X_{m-1} to Y_{n-1} will help us get optimal solution. Therefore, the property holds.

The second is when the last char of X and Y are different. We either use delete or insert to form the target. Assuming the last operation is delete the char from the source. Now, we can simply view the subproblem as X_{m-1} and Y. The optimal solution of subproblem will help us get optimal final solution.

In the third description, assuming the last operation is insert the char to the target and X remain unchanged. The subproblem becomes X and Y_{n-1}. The optimal solution of this subproblem will help us get optimal final solution.

(b) Without twiddle function

From part A, we knew that we can form an OPT table by comparing the characters of source and target. If it's a match, we simply have to copy the character. If it's not, since the cost of replace is 1, which is less than the insert and delete. I will choose replace the character first and insert the remaining part if we don't reach the target. However, this becomes a greedy algorithm if we always find the optimal solution in each step. Therefore, we still need to take the cost of insert and delete into account. And the rule of dynamic programming becomes

Z[i, j] = cost(insert) , i = 0. (Base case, when source has no context)
cost(delete), j = 0 (Base case, when target has no context)
min(Z[i-1, j-1] + cost(copy), X[i] = Y[j]
Z[i-1, j-1] + cost(replace), X[i] != Y[j]
Z[i-1, j] + cost(delete),
Z[i, j-1] + cost(insert))

Pseudo:

```
Initialize Z array for store
for i in range(m):
    Z[i, 0] = i*cost(delete)
for j in range(n):
    Z[0, j] = j*cost(insert)
for i in range(m):
    for j in range(n):
        Z[i, j] = inf
        if x[i]==y[j]:
            Z[i, j] = Z[i-1, j-1] + cost(copy)
        if x[i]!=y[j] && c[i-1, j-1]+cost(replace) < c[i,j]
            Z[i, j] = Z[i-1, j-1] + cost(replace)
        if Z[i-1, j] + cost(delete) < Z[i, j]
            Z[i, j] = Z[i-1, j] + cost(delete)
        if Z[i, j-1] + cost(insert) < Z[i, j]
            Z[i, j] = Z[i, j-1] + cost(insert)
for i in range(m-1):
    if Z[i, n] + cost(kill) < Z[m, n]
        Z[m,n] = Z[i, n] + cost(kill)
return Z
```

Runtime:

Clearly we can see the nested for loop will dominate and will take O(mn)

Space:

To form the table, we need O(mn) space.

(c) With twiddle function

This is similar to the part b instead we have also take the cost of twiddle into account to form the OPT table

Z[i, j] = cost(insert) , i = 0. (Base case, when source has no context)
cost(delete), j = 0 (Base case, when target has no context)
min(Z[i-1, j-1] + cost(copy), X[i] = Y[j]
Z[i-1, j-1] + cost(replace), X[i] != Y[j]
Z[i-2, j-2] + cost(twiddle) X[i]=Y[j-1] && X[i-1]=Y[j]
Z[i-1, j] + cost(delete),
Z[i, j-1] + cost(insert))

Pseudo:

```
Initialize Z array for store
for i in range(m):
    Z[i, 0] = i*cost(delete)
for j in range(n):
    Z[0, j] = j*cost(insert)
for i in range(m):
    for j in range(n):
        Z[i, j] = inf
        if x[i]==y[j]:
            Z[i, j] = Z[i-1, j-1] + cost(copy)
        if x[i]!=y[j] && c[i-1, j-1]+cost(replace) < c[i,j]
            Z[i, j] = Z[i-1, j-1] + cost(replace)
        if (i >= 2 && j>=2 && X[i]==Y[j-1] && X[i-1]==Y[j] && Z[i-2,
          j-2]+cost(twiddle) < Z[i, j])
            Z[i, j] = Z[i-2, j-2] + cost(twiddle)
        if Z[i-1, j] + cost(delete) < Z[i, j]
            Z[i, j] = Z[i-1, j] + cost(delete)
        if Z[i, j-1] + cost(insert) < Z[i, j]
            Z[i, j] = Z[i, j-1] + cost(insert)
for i in range(m-1):
    if Z[i, n] + cost(kill) < Z[m, n]
        Z[m,n] = Z[i, n] + cost(kill)
return Z
```

Runtime:

Clearly we can see the nested for loop will dominate and will take O(mn)

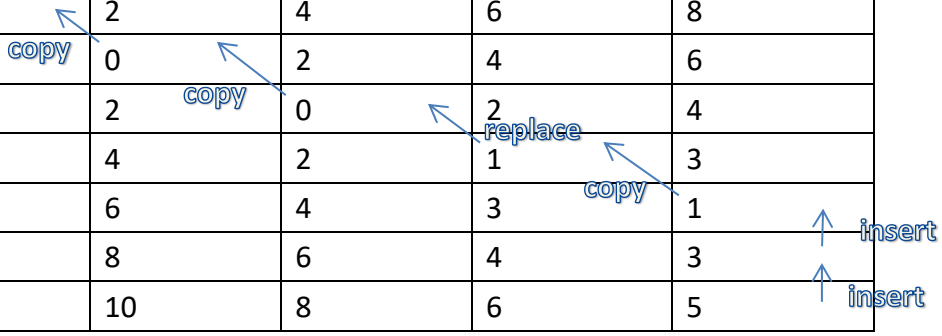
Space:

To form the table, we need O(mn) space.

(d)

Table b

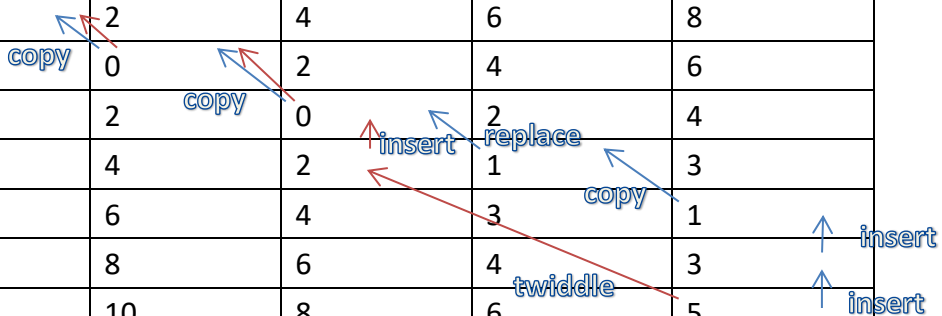
v	0	S	I	L	K
0	0	2	4	6	8
S	2	0	2	4	6
I	4	2	0	2	4
C	6	4	2	1	3
K	8	6	4	3	1
L	10	8	6	4	3
E	12	10	8	6	5



3*cost(copy) + cost(replace) + 2*cost(insert) = 3*0 + 1 + 2*2 = 5.

Table C

	0	S	I	L	K
0	0	2	4	6	8
S	2	0	2	4	6
I	4	2	0	2	4
C	6	4	2	1	3
K	8	6	4	3	1
L	10	8	6	4	3
E	12	10	8	6	5



3*cost(copy) + cost(replace) + 2*cost(insert) = 3*0 + 1 + 2*2 = 5.

Though LK and KL are in reverse-ordered, there is a C inside, which we will need to insert and twiddle, which will also take 2*cost(copy) + cost(twiddle) + 2*cost(insert) = 5. Still cost five, but the traceback operation can be different two path. I marked it as red and blue path.