CS 514

Assignment 6 – Dynamic Programming

report6.txt

Q1. In this assignment you will implement the basic dynamic program for edit distance and edit string between two input strings. Edit distance is a widely used algorithm with a variety of applications from spell checkers to version control to DNA matching. The basic edit distance algorithm is based on dynamic programming and has the complexity of *O(mn).* There are 3 possible operations, insert (I), replace (R), and delete (D), each of which costs 1 unit (unless the characters exactly match).

For example, if x= [babble] and y='apple', the best alignment and the corresponding edit distance are.

B A B B L E

- A P P L E

edit distance: 3

Write a Python function called `editDistance' that takes two strings as inputs and returns the edit distance between the two strings.

Answer-

Please find the implementation of edit distance algorithm:

```
import time
def editDistance(str1,str2):

    # Declaring array 'D' with rows = len(a) + 1 and columns =
len(b) + 1:
    D = [[0 for i in range(len(str2) + 1)] for j in
range(len(str1) + 1)]

    # Initializing first row:
    for i in range(len(str1) + 1):
        D[i][0] = i

    # Initializing first column:
    for j in range(len(str2) + 1):
        D[0][j] = j

    for i in range(1, len(str1) + 1):
        for j in range(1, len(str2) + 1):
            if str1[i - 1] == str2[j - 1]:
                D[i][j] = D[i - 1][j - 1]
```

```
            else:
                # Adding 1 to account for the cost of operation
                insertion = 1 + D[i][j - 1]
                deletion = 1 + D[i - 1][j]
                replacement = 1 + D[i - 1][j - 1]

                # Choosing the best option:
                D[i][j] = min(insertion, deletion, replacement)
    return D[len(str1)][len(str2)]

if __name__ == "__main__":
    print(editDistance("ATCAT", "ATTATC"))


print(editDistance("taacttctagtacatacccgggttgagcccccatttcttggttg
gatgcgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat",
"taacttctagtacatacccgggttgagcccccatttccgaggaacattacgctagaggaacaa
caaggtcagaggcctgttactcctat"))

    print(editDistance("CGCAATTCTGAAGCGCTGGGGAAGACGGGT",
"TATCCCATCGAACGCCTATTCTAGGAT"))

print(editDistance("tatttacccaccacttctcccgttctcgaatcaggaatagacta
ctgcaatcgacgtagggataggaaactccccgagtttccacagaccgcgcgcgatattgctcgc
cggcatacagcccttgcgggaaatcggcaaccagttgagtagttcattggcttaagacgcttta
agtacttaggatggtcgcgtcgtgccaa",
"atggtctccccgcaagataccctaattccttcactctctcacctagagcaccttaacgtgaaa
gatggctttaggatggcatagctatgccgtggtgctatgagatcaaacaccgctttctttttag
aacgggtcctaatacgacgtgccgtgcacagcattgtaataacactggacgacgcgggctcggt
tagtaagtt"))
```

a) (20 points) Make sure that your program works on this test data.

editDistance("ATCAT", "ATTATC") == 2
editDistance("taacttctagtacatacccgggttgagcccccatttcttggttggatgcgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat",
"taacttctagtacatacccgggttgagcccccatttccgaggaacattacgctagaggaacaacaaggtcagaggcctgttactcctat")==11
editDistance("CGCAATTCTGAAGCGCTGGGGAAGACGGGT", "TATCCCATCGAACGCCTATTCTAGGAT") ==18
editDistance("tatttacccaccacttctcccgttctcgaatcaggaatagactactgcaatcgacgtagggataggaaactccccgagtttccacagaccgcgcgcgatattgctcgccggcatacagcccttgcgggaaatcggcaaccagttgagtagttcattggcttaagacgctttaagtacttaggatggtcgcgtcgtgccaa",
"atggtctccccgcaagataccctaattccttcactctctcacctagagcaccttaacgtgaaagatggctttaggatggcatagctatgccgtggtgctatgagatcaaacaccgctttctttttagaacgggtcctaatacgacgtgccgtgcacagcattgtaataacactggacgacgcgggctcggttagtaagtt") ==112

Solution – On running my code, it gives the correct output for all the test cases mentioned.

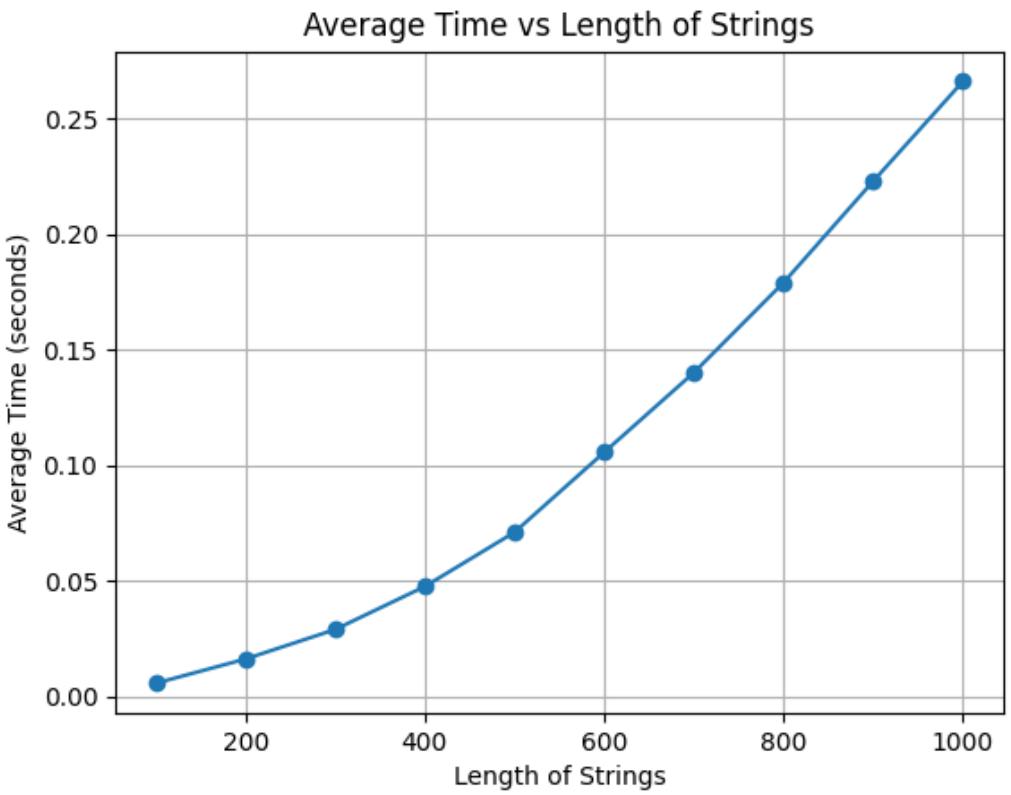
b) (20 points) Generate 10 random pairs of edit strings of length 100, 200, ...,1000, and find their edit distances. Plot the average time taken to compute their edit distances as a function of the length of strings. Comment on the performance of your algorithm.

Solution –

I generated random pairs of edit strings and plotted the graph. Given below are my readings and my plot.

S. no	Length of Strings	Time taken
1	100	0.005533933639526367
2	200	0.026521921157836914
3	300	0.05439186096191406
4	400	0.10334491729736328
5	500	0.16489601135253906
6	600	0.278789758682251
7	700	0.3472139835357666
8	800	0.4521059989929199
9	900	0.5746569633483887
10	1000	0.6551532745361328

The plot of the above readings is as shown –



We ran the program with random string of lengths 100 to 1000 as inputs and got these readings/plot. We can see from the graph that the algorithm clearly shows a representation of $O(n^2)$ time complexity.

On looking at the theoretical time complexity of the edit-distance algorithm, which is O(m*n) where m and n are lengths of the two input strings, we can see that the code provided as answer aligns with the theoretical time complexity. The plot represents a slope of $O(n^2)$ and as here, the two input strings have the same length, i.e.- m = n, it is accurate to say that the worst-case time complexity of this algorithm aligns with that of the theoretical time complexity.

Q2. (10 points) Give an *O(mn)* algorithm for finding the longest common substring of two input strings of length *m* and *n*. For example, if the two inputs are 'Philanthropic" and "Misanthropist," the output should be "anthropi."

Solution –

The algorithm for finding the longest common substring using dynamic programming concepts is similar to the edit distance algorithm.

Algorithm –

```
def longestCommonSubstring(str1, str2):
    # Initialize a 2D table to store lengths of common suffixes
    dp = [[0] * (n + 1) for _ in range(m + 1)] #m and n are lengths of input strings
    # Variables to track the length and ending index of the longest common substring
    max_length = 0
    end_index = 0
    # Iterate through the input strings
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > max_length:
                    max_length = dp[i][j]
                    end_index = i - 1
            else:
                dp[i][j] = 0
    # Extract the longest common substring
    longest_substring = str1[end_index - max_length + 1:end_index + 1]
    return longest_substring
```

This algorithm uses a similar dynamic programming approach to fill a table (dp) and extracts the longest common substring based on the information stored in the table.

Explanation of the algorithm -

1. We initialize a 2D table `dp` of size (m+1) * (n+1) to store the lengths of common substrings.
2. We initialize variables `max_length` and `end_index` to keep track of the length of the longest common substring and its ending index.
3. We iterate through the input strings `str1` and `str2` using two nested loops.

4. For each pair of characters in the input strings, if they are equal, we update the `dp` table using the relation dp[i][j] = dp[i-1][j-1] + 1. If the characters are not equal, we set dp[i][j] = 0 because a mismatch breaks the common substring.
5. During the table filling process, whenever we update a cell in the table, we check if the updated length is greater than the current `max_length`. If it is, we update `max_length` and set `end_index` to the ending index of the current substring.
6. Once the table is filled, the longest common substring's length is stored in `max_length`, and its ending index is stored in `end_index`. We extract the substring from `str1` using the calculated ending index and length.

Time Complexity -
The time complexity of this algorithm is O(mn) because each cell in the dynamic programming table is computed in constant time. The nested loops iterate over the lengths of both input strings, resulting in O(mn) total operations. This time complexity is efficient and ensures that the algorithm scales well with the size of the input strings.

Q3. (30 points) BigBucks wants to open a set of coffee shops in the I-5 corridor. The possible locations are at miles d_1, ... , d_n in a straight line to the south of their Headquarters. The potential profits are given by $p_1 ... p_n$. The only constraint is that the distance between any two shops must be at least k (a positive integer).

A. Construct a counterexample to show that a greedy algorithm that chooses in the order of profits could miss the optimal (most profitable) solution.

Solution –

We take an example where BigBucks' plan to open coffee shops along the I-5 corridor, denoted at distances 1, 4, 7, 10, and 13 miles from their headquarters. The associated profits for these locations are 10, 30, 20, 5, and 25, respectively.

The crucial constraint here is that the distance between any two shops must be at least (k = 4) miles.

Now, if we employ a greedy algorithm that prioritizes locations based on profits, it might select them in the order 4, 30, 25, 10, 5. This choice seems intuitive as it maximizes profits at each step. However, this greedy strategy leads to distances between adjacent shops of 26, 1, 15, and 5 miles, respectively, violating the specified constraint.

In contrast, the optimal solution could involve choosing locations in the order 4, 10, 13. This arrangement ensures distances of 6, 3, and 3 miles, all satisfying the (k = 4) mile constraint. The total profit in this case is 55, surpassing the profit obtained by the greedy algorithm.

This counterexample illustrates that a profit-driven greedy algorithm may overlook the spatial constraints between coffee shop locations, and thus, fail to secure the most profitable solution**.**

B. Give an efficient dynamic programming-based algorithm to maximize the profit.

Solution –

P[i] is defined as the maximum expected profit at location i Based on the constraints we have, we come up with the following recursive definition of P[i].

$$P[i] = \max(\max_{j<l} \{P[j] + \beta(d_i,d_j) * p[i]\}, p[i])$$

where β is a function that decides whether there can be a shop at location d.
The definition of β is = 0 if di – dj < k or 1 if di – dj >= k.

For the location i, the maximum profit come from the maximum of the expected profit of the location j and whether one can open the shop at the location i. The profit of opening the shop at the location i is pi. The maximum profit that is being expected at the location j is represented by P[j]. There may be or may not be a shop at the location j. There are some cases where there is no coffee shop at the location j, then these conditions will be represented by cases k in which k < j. The profit obtained at the location i, pi is greater than P[j] + $\beta(d_i,d_j)$ .Therefore, for these cases comparison with P[i] is needed.

Algorithm –

procedure expected-profit(N,P)
Input: N locations; P[1,..,N] where P[i] denotes profit at location i
Output: Maximum expected profit Pmax
Declare an array of maximum expected profit Profit[1..N]: Profit[i] denotes maximum expected profit at location i
for i = 1 to N:
    Profit[i] = 0
for i = 2 to N
    for j = 1 to i-1
        temp = Profit[j] + β (di , dj) · P[i]
        if temp > Profit[i]:
            temp = Profit[i]
    if Profit[i] < P[i]:
        Profit[i] = P[i]

Time complexity – There are two for loops so the time complexity is $O(n^2)$.

Q4. In a rope cutting problem, cutting a rope of length *n* into two pieces costs *n* time units, regardless of the location of the cut. You are given m desired locations of the cuts, X_1, ..., X_m. Give a dynamic programming-based algorithm to find the optimal sequence of cuts to cut the rope into m+1 pieces to minimize the total cost.

Hint: Let X_0 and $X_{[m+1]}$ be the two ends of the rope. Write a Bellman equation for Cost[X_i, X_j] which represents the minimum cost of cutting the part of the rope from location X_i to location X_j into j-i pieces in between.

Solution –

To solve the rope cutting problem using dynamic programming, we can define a cost function and use a bottom-up approach to fill a 2D table representing the minimum cost of cutting the rope at various locations. The cost function can be defined as follows:

Let Cost[X_i , X_j] represent the minimum cost of cutting the part of the rope from location X_i to location X_j into j-i pieces in between.

Algorithm –

```
from sys import maxsize
def minCost( n, cuts) -> int:
    c = len(cuts)
    cuts.append(n)
    cuts.insert(0,0)
    cuts.sort()
    dp = [[0 for j in range(c + 2)] for i in range(c + 2)]
    for i in range(c, 0, -1):
        for j in range(1, c + 1):
            if i > j:
                continue
            mini = maxsize
            for k in range(i, j + 1):
                cost = cuts[j + 1] - cuts[i - 1] + dp[i][k - 1] + dp[k +
1][j]
                mini = min(mini, cost)
            dp[i][j] = mini
    cuts.remove(0)
    cuts.remove(n)
    return dp[1][c], cuts
```

Explanation –

Here's an explanation of the algorithm:
1.Add the two ends of the rope (X0 and X[m+1]) to the list of cuts.
2. Sort the cuts in ascending order.
3. Create a 2D array dp to store the minimum cost of cutting the rope between different pairs of cut locations.
4. Iterate over the cuts in reverse order (from m to 0) and fill in the dp array using the Bellman equation.

The Bellman equation for `Cost[$X_i$ , $X_j$]` can be defined as follows:

$$\mathbf{Cost[X_i , X_j] = min_{i<=k<=j}\{ Cost[X_i , X_k] + Cost[X_k , X_j] \} + (X_j - X_i)}$$

It is represented in the code as –

Cost = cuts [Xj + 1] – cuts [Xi -1] + dp[i][k-1] + dp[k+1][j]

Where i<=k<=j. This equation calculates the cost of cutting the rope between locations Xi and Xj at different possible cut points k, and the minimum cost is stored in the ‘dp’ array.

Finally, the optimal solution is obtained from dp[1][c], where c is the number of cuts. And the sequence is obtained from cuts.
This algorithm has a time complexity of O(n^3) and space complexity of O(n^2) where n is the length of the rope.