CS301-HW4

Ahmet Furkan Ün (28315) - Taner Giray Sönmez (27744)

December 24, 2023

(a)

(i) Recursive Formulation

Assuming that c[i, j] denotes the update distance between the prefix X[1..i] (i.e., the first i tuples of X) and the prefix Y [1..j] (i.e., the first j tuples of Y) we can define the c[i, j] as the following.

In the case when both i and j is 0, we do not need any update on the tuples since the prefixes are empty. When i is 0, this means the prefix of X is empty, to make the prefix of X same as prefix of Y, we need to insert all tuples in the prefix of Y to the prefix of X meaning j insert operations. When j is 0, this means the prefix of Y is empty, to make the prefix of X same as prefix of Y, we need to delete all tuples in the prefix of X meaning i delete operations.

After base cases, we need to consider recursive cases. If X[i] = Y [j] meaning that the last elements of the prefixes are the same, c[i, j] = c[i-1, j-1] since we do not need to make any operations due to the last tuples of each prefixes. If X[i] $\neq$ Y [j] meaning that the last elements of the prefixes are not same, we need to perform one operation based on the last elements. If the operation is one insertion to the prefix of X, c[i, j] = 1 + c[i, j-1] where plus one comes from the insertion operation. If the operation is one deletion from the prefix of X, c[i, j] = 1 + c[i-1, j] where plus one comes from the deletion operation. Lastly if the operation is one replacement on the prefix of X, c[i, j] = 1 + c[i-1, j-1] where plus one comes from the replacement operation. However since we are trying to find the minimum update distance, we need to find which operation results in the minimum number of operations.

$$
c[i,j] = \begin{cases}
0 & \text{if } i = 0 \text{ and } j = 0 \text{ (both prefixes of X and Y are empty)} \\
j & \text{if } i = 0 \text{ (Prefix of X is empty, insert j tuples to X)} \\
i & \text{if } j = 0 \text{ (Prefix of Y is empty, delete i tuples from X)} \\
c[i-1, j-1] & \text{if } X[i] = Y[j] \text{ (no operation needed)} \\
1 + \min \begin{cases} c[i, j-1] & \text{(insertion)} \\ c[i-1, j] & \text{(deletion)} \\ c[i-1, j-1] & \text{(replace)} \end{cases} & \text{if } X[i] \neq Y[j] \text{ (one of the operations is needed)}
\end{cases}
$$

(ii) Optimal Substructure Property

Case $X[i] = Y[j]$:
Assuming that $c[i, j] = z$ is the minimum update distance between the prefix $X[1..i]$ and the prefix $Y[1..j]$, prove that $c[i-1, j-1] = z$ is the minimum update distance between the prefix $X[1..i-1]$ and the prefix $Y[1..j-1]$. Assume otherwise, there exists a smaller number w of updates that can be performed on $X[1..j-1]$ to obtain $Y[1..i-1]$ where $w < z$. Then, if you cut these updates and paste them into the updates on the prefix $X[1..j]$ when calculating $c[i, j]$, it implies obtaining a new update distance $0 + w < z$. This contradicts our initial claim of z being the minimum update distance. Therefore, we can conclude that there are no updates on $X[1..j-1]$ where the update distance is $w < z$.

Case $X[i] \neq Y[j]$ and operation is insert:
Assuming that $c[i, j] = z$ is the minimum update distance between the prefix $X[1..i]$ and the prefix $Y[1..j]$, prove that $c[i, j-1] = w$ is the minimum update distance between the prefix $X[1..i]$ and the prefix $Y[1..j-1]$, where $z = w+1$. Assume otherwise, there exists a smaller number w' of updates that can be performed on $X[1..j-1]$ to obtain $Y[1..i]$ where $w' < w$. Then, if you cut these updates and paste them into the updates on the prefix $X[1..j]$ when calculating $c[i, j]$ and perform just one more insertion operation, it implies obtaining a new update distance $1 + w' < 1 + w = z$. This contradicts our initial claim of z being the minimum update distance. Therefore, we can conclude that there are no updates on $X[1..j-1]$ where the update distance is $w' < w$.

Case $X[i] \neq Y[j]$ and operation is deletion:
Assuming that $c[i, j] = z$ is the minimum update distance between the prefix $X[1..i]$ and the prefix $Y[1..j]$, prove that $c[i-1, j] = w$ is the minimum update distance between the prefix $X[1..i-1]$ and the prefix $Y[1..j]$, where $z = w+1$. Assume otherwise, there exists a smaller number w' of updates that can be performed on $X[1..j]$ to obtain $Y[1..i-1]$ where $w' < w$. Then, if you cut these updates and paste them into the updates on the prefix $X[1..j]$ when calculating $c[i, j]$ and perform just one more delete operation, it implies obtaining a new update distance $1 + w' < 1 + w = z$. This contradicts our initial claim of z being the minimum update distance. Therefore, we can conclude that there are no updates on $X[1..j]$ where the update distance is $w' < w$.

Case $X[i] \neq Y[j]$ and operation is replace:
Assuming that $c[i, j] = z$ is the minimum update distance between the prefix $X[1..i]$ and the prefix $Y[1..j]$, prove that $c[i-1, j-1] = w$ is the minimum update distance between the prefix $X[1..i-1]$ and the prefix $Y[1..j-1]$, where $z = w+1$. Assume otherwise, there exists a smaller number w' of updates that can be performed on $X[1..j-1]$ to obtain $Y[1..i-1]$ where $w' < w$. Then, if you cut these updates and paste them into the updates on the prefix $X[1..j]$ when calculating $c[i, j]$ and perform just one more delete operation, it implies obtaining a new update distance $1 + w' < 1 + w = z$. This contradicts our initial claim of z being the minimum update distance. Therefore, we can conclude that there are no updates on $Y[1..j-1]$ where the update distance is $w' < w$.

(iii) Overlapping Subproblems

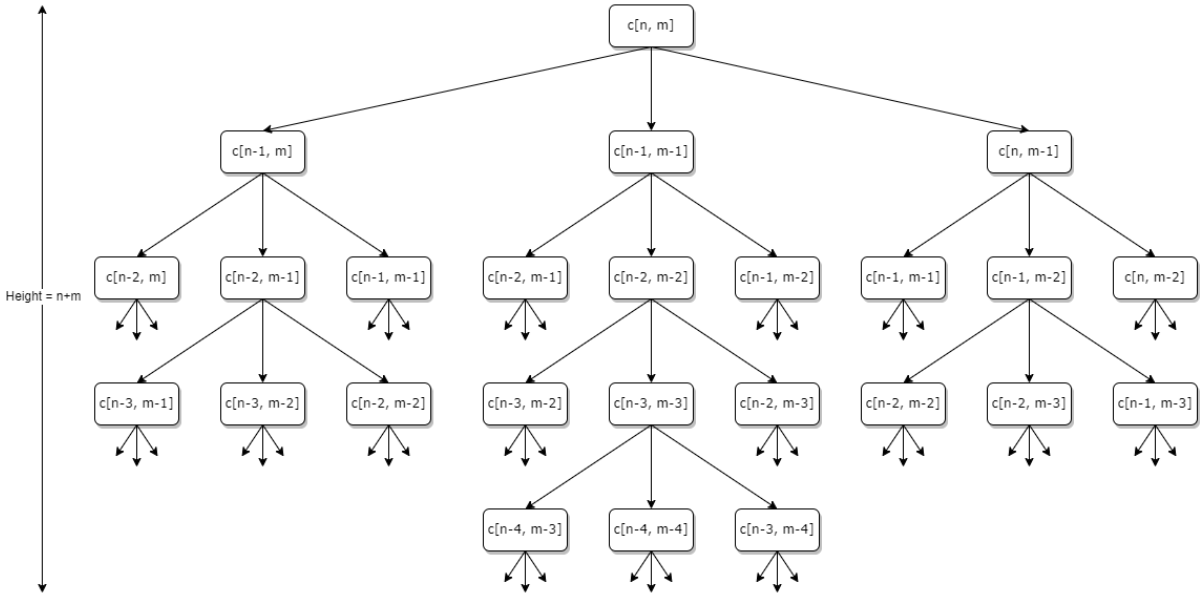

Figure 1: Recursion Tree

As illustrated in Figure 1, the recursion tree for the tuple update distance problem exhibits a significant amount of overlapping subproblems. This phenomenon arises due to the recursive nature of the solution and the redundant computation of certain subproblems multiple times. The recurrence relation lead to the creation of a large number of subproblems, and the exponential growth of the recursion tree size is evident.

The number of elements in the recursion tree is determined by the branching factor raised to the power of the tree height, which is given by $branching^{height} = 3^{n+m}$. Here, the branching factor represents the number of recursive calls made at each level of the tree, and the height corresponds to the depth of recursion. In the context of the update distance problem, the branching factor is associated with the choices made at each step (delete, insert, replace), leading to a large number of possible paths through the recursion tree.

On the other hand, the number of distinct subproblems is proportional to the product of the lengths of the input sequences, denoted as $n \cdot m$. This difference between the number of distinct subproblems and the size of the recursion tree is the indicator of overlapping subproblems. This situation makes dynamic programming solution a lot more efficient.

(iv) Topological Ordering of Subproblems

Regarding the existence of a topological ordering of subproblems, a topological ordering is possible because the recursion tree in the Figure 1, forms a Directed Acyclic Graph (DAG). This acyclic nature ensures that each subproblem depends only on subproblems that are solved earlier in the topological ordering and there is no cyclic dependence between the subproblems. In the context of dynamic programming, a topological ordering guarantees that each subproblem is solved exactly once and before its dependencies, ensuring the alorithm will successfully terminate at the end and return the correct result.

(b) Pseudocode

Our algorithm provided below adopts a dynamic programming paradigm, utilizing a 2D array named dp to store and manage intermediate results. The function `TupleUpdateDistance` accepting two input sequences, X and Y, and proceeding with the computation of their respective lengths, n and m.

The initial phase of the algorithm involves the setup of the dp array, configured with dimensions $(n+1) \times (m+1)$. Subsequently, the algorithm enters a nested loop structure responsible for filling the dp table. Within this loop, each $dp[i][j]$ entry represents the minimum update distance between the prefixes $X[1..i]$ and $Y[1..j]$. During this process, the algorithm carefully handles base cases, accounting for scenarios where one or both of the prefixes are empty. Specifically, if i and j are both zero, indicating empty prefixes, the update distance is naturally zero. In cases where i or j is zero, signifying an empty prefix for either X or Y, the algorithm incorporates the requisite number of insertions or deletions.

The primary dynamic programming loop takes center stage, determining the minimum update distances for non-trivial cases. It assesses whether the last tuples in the prefixes ($X[i-1]$ and $Y[j-1]$) match. If a match is detected, signifying similarity in the tuples, the algorithm concludes that no operations are necessary for the last tuples, and $dp[i][j]$ inherits the value of $dp[i-1][j-1]$. Conversely, when the last tuples differ, the algorithm strategically computes the update distance. Three potential operations are considered: deletion, insertion, and replacement. The algorithm selects the minimum distance among these operations, ensuring an optimal and efficient approach to the transformation.

Post the dynamic programming phase, the algorithm engages in a backtracking process to identify the optimal sequence of operations. Indices i and j are initialized to the lengths of X and Y, respectively. The algorithm iterates through the dp table, determining the operations required for the transformation. These operations are stored in a list named `operations`. To ensure the correct order of operations, the algorithm reverses this list. Lastly, we need to consider the index changes when insert or delete operations performed. To adjust this, the algorithm traverses all the operations found and updates the indices at which the operation will performed.

Algorithm 1 Tuple Update Distance

```
procedure TupleUpdateDistance(X, Y)
    m ← length of X
    n ← length of Y
    dp ← 2D array of size (m + 1) × (n + 1) filled with zeros
    for i ← 0 to m do
        for j ← 0 to n do
            if i = 0 and j = 0 then
                dp[i][j] ← 0
            else if i = 0 then
                dp[i][j] ← j
            else if j = 0 then
                dp[i][j] ← i
            else if X[i − 1] = Y[j − 1] then
                dp[i][j] ← dp[i − 1][j − 1]
            else
                dp[i][j] ← 1 + min(dp[i − 1][j], dp[i][j − 1], dp[i − 1][j − 1])
            end if
        end for
    end for
    i, j ← m, n
    operations ← empty list
    while i > 0 or j > 0 do
        if i > 0 and j > 0 and X[i − 1] = Y[j − 1] then
            i ← i − 1
            j ← j − 1
        else if i > 0 and dp[i][j] = dp[i − 1][j] + 1 then
            Append ('delete', i) to operations
            i ← i − 1
        else if j > 0 and dp[i][j] = dp[i][j − 1] + 1 then
            Append ('insert', j − 1, Y[j − 1]) to operations
            j ← j − 1
        else
            if i > 0 and j > 0 then
                Append ('replace', i, Y[j − 1]) to operations
            end if
            i ← i − 1
            j ← j − 1
        end if
    end while
    Reverse operations
    offset ← 0
    for operation in operations do
        if operation is insert then
            operation.index ← operation.index + offset
            offset ← offset + 1
        else if operation is delete then
            operation.index ← operation.index + offset
            offset ← offset - 1
        else
            operation.index ← operation.index + offset
        end if
    end for
    return dp[m][n], operations
end procedure
```

(c) Best Worst-case Asymptotic Time Complexity Analysis

The best worst-case asymptotic time complexity analysis of our algorithm involves two primary operations: the initialization and filling of the dynamic programming table, and the subsequent backtracking phase to identify the optimal sequence of operations.

In the first phase, the algorithm creates a 2D array, denoted as dp, with dimensions $n+1$ rows and $m+1$ columns. The initialization phase focuses on filling the edges of the array, a process that takes $O(n+m)$ time. Subsequently, the algorithm fills the inner cells by traversing the array and considering the left, diagonal and upper cells at each step. This inner cell filling involves comparison of 3 values and finding the minimum and takes constant-time operations, resulting in a time complexity of $O((n-1)\times(m-1))$. Consequently, the overall time complexity for filling the dp array is $O(n+m)+O((n-1)\times(m-1))$, which simplifies to $O(n\times m)$. We can also achieve this result by multiplying the cost of one subproblem and the number of unique sub problems. To solve each subproblem, we either directly assign a value or compare 3 values and choose the minimum. This takes constant time, in other words the cost of each subproblem is $O(1)$. The number of unique subproblems are the size of the dp matrix which is $n\times m$. By multiplying this two values we reeach the complexity of this phase as $O(1\times n\times m)=O(n\times m)$

Moving on to the second operation, the backtracking phase occurs after the completion of the dynamic programming table. At each step, the algorithm either moves diagonally, left, or up, traversing the table with constant-time operations per step. After each step either i and j, or both i and j decreases by 1, meaning that at most this step traverses $n+m$ elements. After finding the operations, we need to traverse all of the operations once to update the indices of them. The number of operations in the worst case can be n in the case where we change all elements of X to make it eqaul to Y. Therefore, the backtracking phase contributes $O(2n+m)$ to the overall time complexity.

Combining the complexities from both the dynamic programming table filling and the backtracking phase, the overall best worst-case asymptotic time complexity of the algorithm is $O(2n+m)+O(n\times m)$, which simplifies to $O(n\times m)$.

(d) Results of Performance Evaluations

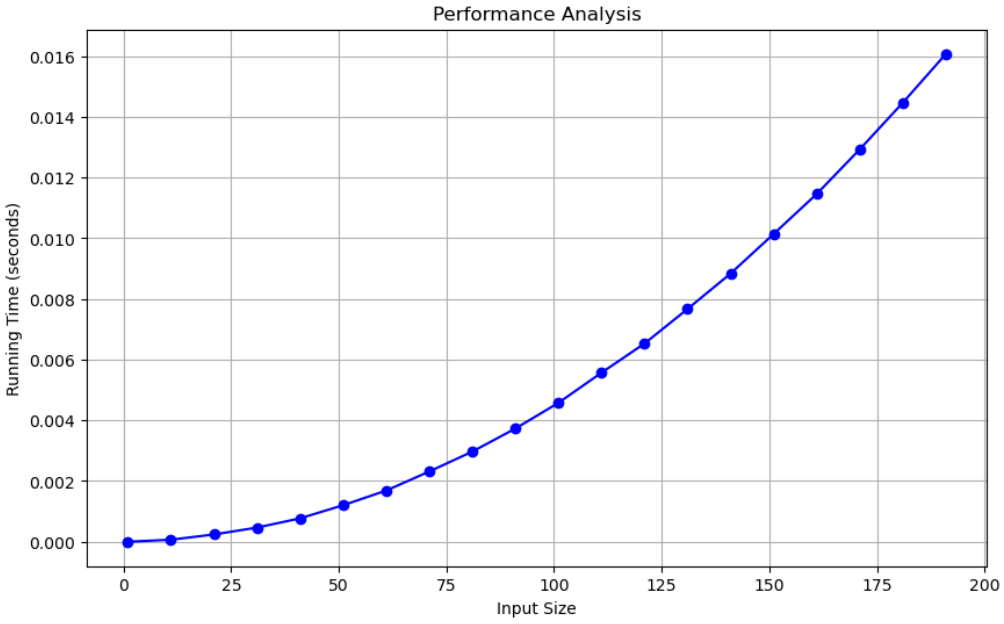

Figure 2: Performance Results

To evaluate the performance of our algorithm, we run the algorithm on 20 different input sizes for 1000 different random inputs for each size. Then we calculated the average running time of our algorithm on each input size. In our graph, the x-axis represents the input size, and the y-axis represents the average running time in seconds. First thing we can notice that, as the input size increases, the running time appears to increase as well, which is consistent with the expected $O(n \times m)$ time complexity of our dynamic programming solution. The graph shows that the increase in running time is **polynomial**, exactly what we found as $O(n \times m)$.

For small inputs, the running time is very quick, but as the size of the problem instances increases, the running time grows quickly. This shows us exactly the nature of polynomial algorithms.

An important point worth noting is the characteristics of the system on which the algorithm runs. Our system has the following features and can be considered a high-end system. For this reason, we reached a solution within a split second, but the results may be different for changing system features. Our system specs for testing:
CPU: AMD Ryzen 7 3700X 8-Core Processor 3.6 GHz
RAM: 16GB
OS: x64 Windows 10
GPU: NVIDIA RTX 3070
When discussing scalability in terms of how the time changes as the input size increases by a certain factor, we can make the following observations from the time complexity $O(n \times m)$ of our algorithm: If we double the input size such that increase n to $2n$ and m to $2m$. Then, time complexity will increase by a factor of four because $O((2n) \times (2m)) = 4 \times O(n \times m)$. This means that if we doubles the input size, then the complexity will be four times larger. If we look at the values of input size 50 and 100 to see this in the graph, for input size 50 we measured the running time as approximately 0.001. If we double the input size and make it 100, we measured the running time as approximately 0.004, which means four times 0.001. This observation supports and validates the expected behavior of the algorithm as we found.