CS 301 A4 Efe - Alpay

Alpay Naçar - Hamit Efe Çınar

December 2023

1 Introduction

Update distance problem requires the transformation of one sequence of tuples into another through a minimum set of operations. The task involves two sequences, X and Y, each comprising pairs of non-negative integers, and the objective is to convert sequence X into sequence Y using the fewest possible operations, which include insertion, deletion, and replacement of tuples. Input of this procedure is two sequences of tuples, and the output is the minimum update distance and the operations needed for that minimum update.

2 Recursive Formulation

2.1 Subproblem Definition

Recursive definition of the problem can be expressed as follows (where i and j denotes the i'th and j'th element of the sequence to be updated (X) and the sequence to be arrived at (Y) respectively):

$$\text{c[i,j]} = \text{UpdateDistance}(X, Y, i, j) = \begin{cases} \text{UpdateDistance}(X, Y, i-1, j-1) & \text{if } X[i] = Y[j], \\ \min \begin{pmatrix} \text{UpdateDistance}(X, Y, i, j-1) + 1, \\ \text{UpdateDistance}(X, Y, i-1, j-1) + 1, \\ \text{UpdateDistance}(X, Y, i-1, j) + 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Operations given inside the minimum functions represent:

- UpdateDistance(X, Y, i-1, j-1): No action needed, as X[i] = Y[j]
- UpdateDistance(X, Y, i, j-1) + 1: Insertion needed, as X lacks Y[j] in the needed position
- UpdateDistance(X, Y, i-1, j-1) + 1: Replacement needed, as X[i] should be Y[j]
- UpdateDistance(X, Y, i-1, j) + 1: Deletion needed, as X[i] is unnecessary

2.2 Optimal Substructure Property

UpdateDistance(X,Y,N,M) is the minimum update distance between the sequence X of length N, and the sequence Y of length M.

Cut and Paste: Claim that UpdateDistance(X,Y,i,j) is the optimal substructure (minimum update distance of the subproblem) to compute the main problem (denoted above). To prove the optimal substructure via proof by contradiction, assume that UpdateDistance(X,Y,i',j') is a substructure of the problem whose value is less than UpdateDistance(X,Y,i,j) and requires same amount of updates (d) as UpdateDistance(X,Y,i,j) to arrive itself from UpdateDistance(X,Y,M,N). It can be formulated as UpdateDistance(X,Y,i',j') + d < UpdateDistance(X,Y,i,j) + d = UpdateDistance(X,Y,N,M). As it can be observed, UpdateDistance(X,Y,i',j') + d's value is less than UpdateDistance(X,Y,N,M) which contradicts the fact that UpdateDistance(X,Y,N,M) is the minimum update distance. Thus, optimal substructure property's proof follows this assumption's contradiction and the claim being held.

2.3 Recursion Tree

For this tree the recurrence function is denoted as c[N,M] for two sequences of length N and M:

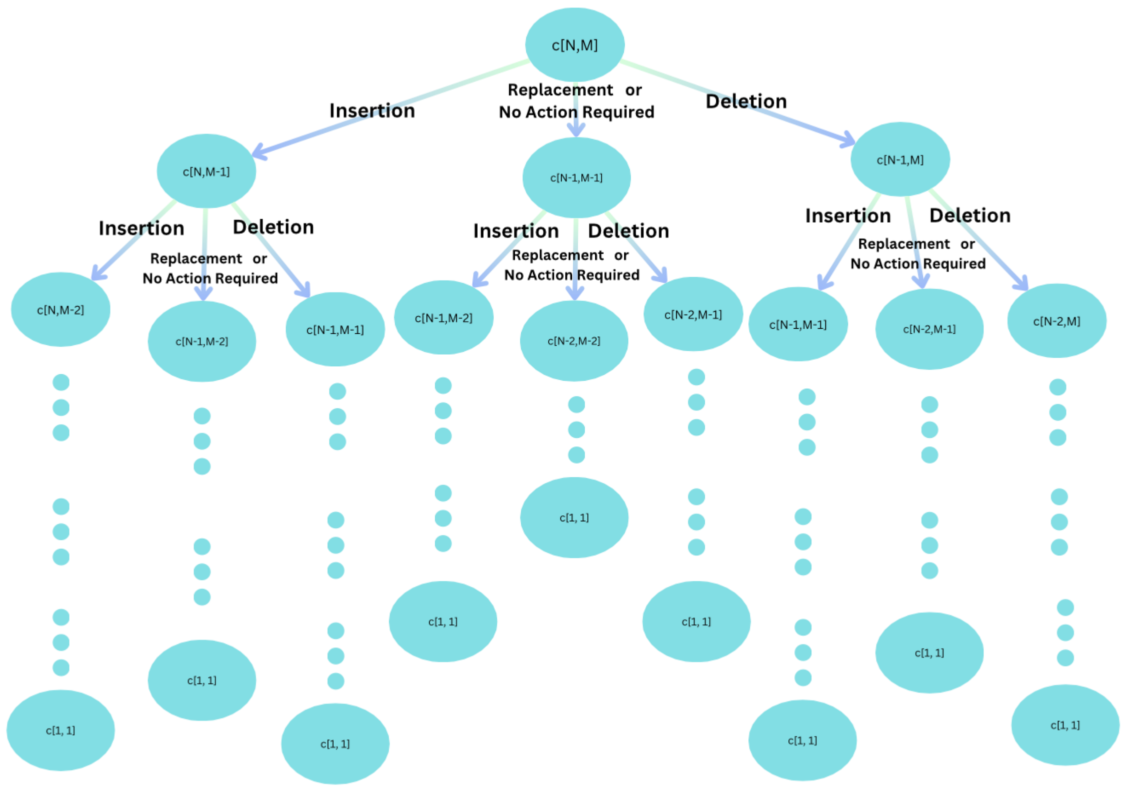

Figure 1: Recursion Tree

As can be observed from the recursion tree, considerable amount of subproblems are overlapping such as c[1,1], c[N-1, M-1] and so on. But to understand the amount of overlapping, the number of subproblems and the number of unique subproblems can be compared. The number of subproblems can be estimated through considering its height and width. Its width is 3 as the subproblems are solved by considering three other subproblems as showed in Figure 1. The height can be found by following the worst-case for the tree which are the insertion and deletion cases as they decrement only one index. As they decrement one from one index in each step, the height is the sum of the lengths of the sequences, N+M. The number of nodes is nearly $height^{width}$, so the number of subproblems is 3^{N+M}. On the other hand, the number of unique subproblems can be found by considering that the combinations iterate over the range of N for the range of M. It is because of the fact that i'th and j'th element of the sequences X of length N and Y of length M respectively, and to do that their elements are iterated over. Thus, the number of unique subproblems is M*N. So, it can be concluded that M*N distinct subproblems are overlapping in the recursion tree to a total of nearly 3^{N+M} subproblems, which takes the problem to an exponential complexity from polynomial if a naive recursive algorithm is applied as the distinct subproblems are recomputed excessively and unnecessarily.

2.4 Topological Ordering of Subproblems

To be able to use dynamic programming, there should be topological ordering in between subproblems, in other words, there shouldn't be a cycle in the dependency graph of subproblems.
To show, topological ordering, we can introduce levels, to our recursion tree, where each subproblem is located at the level according to the sum of i and j values. Tree is still the same, and amount of subproblems are the same, the only change is the places where of subproblems are located. If we can show that each dependency of a subproblem is located at lower levels, we can say that there is an ordering in between subproblems using proof by contradiction, and the fact that each dependency is located at lower levels.

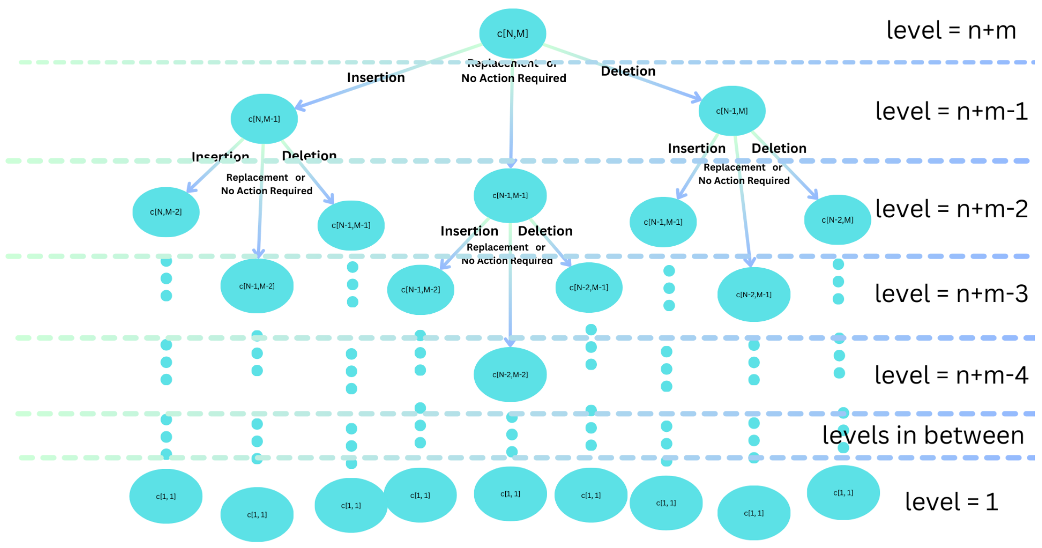

Figure 2: Leveled Recursion Tree

As you can see from figure 2, for every subproblem, insert and delete dependencies are located one level lower, and replace dependency is located at two levels lower, when compared to the level of subproblem itself. We can conclude that every dependency of a subproblem is located at lower levels than the subproblem itself.
For a cycle to be present in our leveled recursion tree (dependency graph), for any node with level k, it should depend on a subproblem with a level higher than k so that it can depend on the same subproblem on level k using delete, or insert, or replace, but it is impossible to happen because all dependencies of this subproblem are located under level k, therefore using this contradiction, we can say that there exists a topological ordering in between our subproblems.

3 Pseudocode

```
function update_distance_recursive($X, Y$, memo, $i, j$) :
  if memoized:
    return memoized
  if $i == 0$ :
    insert for $j$ times
    memoize and return
  if $j == 0$ :
    delete for $i$ times
    memoize and return
  if $X$'s $i$'th tuple equals Y's $j$'th tuple:
    return update_distance_recursive($X, Y$, memo, $i - 1, j - 1$)
  else:
    insert = update_distance_recursive($X, Y$, memo, $i, j - 1$)
    delete = update_distance_recursive($X, Y$, memo, $i - 1, j$)
    replace = update_distance_recursive($X, Y$, memo, $i - 1, j - 1$)
    if insert[0] $\leq$ delete[0] and insert[0] $\leq$ replace[0] :
      insert and memoize
    elif delete[0] $\leq$ insert[0] and delete[0] $\leq$ replace[0] :
      delete and memoize
    else:
      replace and memoize
  return memoized
```

```
function update_distance($X, Y$) :
    $n$ = length of $X$
    $m$ = length of $Y$
    initialize memo matrix of size $(n + 1) \times (m + 1)$
    memo[0][0] = (0, [])
    return update_distance_recursive($X, Y$, memo, $n, m$)
```

4 Best Worst-Case Asymptotic Time Complexity Analysis

Since our algorithm is a dynamic programming algorithm and uses memoization to avoid recomputing overlapping subproblems, the algorithms' best worst-case asymptotic time complexity is the number of unique subproblems for given two sequences of lengths N and M. Our algorithm recursively calculates each subproblem, if it is not calculated before and X[i] equals Y[j], it calls the function recursively with indexes i-1 and j-1, if X[i] and Y[j] are not equal, then uses the best result of insert, delete, and replace. The unique subproblems are stored in a 2D array whose size is M*N because of the subsequence combinations (subarrays' sizes being decremented from the end makes it M*N combinations as explained in section 2.3). As subproblems' values are stored in a 2D array retrieval of already computed subproblems take constant time. The comparison made for the last tuples of the subarrays take constant time too as the 2 integer comparisons made for these comparisons taking O(1) time as well. So, for the first times that subproblems computed, the running time is N*M*O(1)=O(N*M). For the times that a retrieval from the memoized info is applied, it takes O(1) time to retrieve from the 2D array without the need to calculate its 3 subproblems. In conclusion, each subproblem is calculated once, and each subproblem can use results of 3 subproblems at maximum and compare them to find minimum of them, therefore each subproblem calculated using O(1) time. Since the overlapped subproblems are retrieved without looking at their subproblems in these cases, the number of retrievals do not exceed the O(N*M) complexity. Thus, the best worst-case asymptotic time complexity of this algorithm is O(N*M).

5 Performance Evaluation

We chose two random integers 10 times from a range that is increasing, and created two random sequences of tuples 100 times whose sizes are those integers. A random pick in certain ranges for the sizes is determined beforehand to create cases where insertion and deletion are needed. For example, if Y were always bigger than X in terms of size, the minimum update distance would not use deletion for X.
After that, we evaluated the time takes for our algorithm to compute the minimum update distance and the operations between them. Here are the results:

Range	**1-10**	**11-20**	**21-30**	**31-40**	**41-50**	**51-60**	**61-70**	**71-80**	**81-90**	**91-100**
Time	1.34e-5	1.76e-4	8.44e-4	1.47e-3	2.25e-3	3.62e-3	4.91e-3	7.03e-3	9.73e-3	1.33e-2

Then, we have plotted the results with the theoretical worst-case running time complexity function of the algorithm:

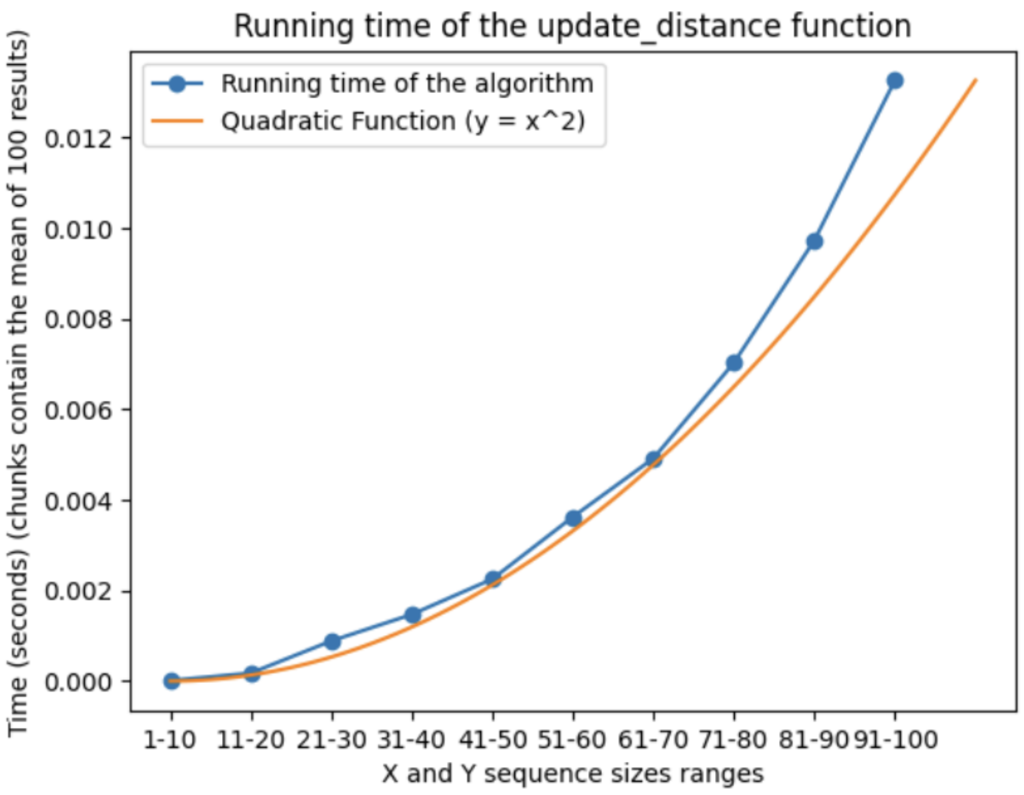

Figure 3: Plot for Performance Evaluation

As can be observed from Figure 3, growth rate of our algorithm near-perfectly aligned with the theoretically expected running time of the function, M*N which may corresponds to a quadratic function as N and M values are close in these cases.

6 Testing

6.1 Benchmark Suite A

Tests below are designed to test the correctness of our algorithm.

6.1.1 Whitebox Testing

These tests are designed with knowledge of the internal structure of the algorithm. They specifically target different code paths to ensure comprehensive coverage.

1. **Empty X Case:** Tests the scenario when the initial list X is empty, and elements need to be inserted to match list Y.
2. **Empty Y Case:** Checks the behavior when the target list Y is empty, requiring deletion of all elements from X.
3. **Same Array Case:** Ensures that no operations are needed when both lists are identical.
4. **Multiple Deletion Case:** Evaluates the algorithm's ability to handle multiple deletions when X has more elements than Y.
5. **Multiple Insertion Case:** Tests multiple insertions when Y has more elements than X.
6. **Replace Case:** Assesses the algorithm's replace functionality for a single element.

6.1.2 Blackbox Testing

These tests are designed without considering the internal implementation, focusing on the functionality from a user's perspective.

1. **Base Case:** Verifies the algorithm's behavior with two empty lists, where no operations are required.
2. **Multiple Replace Case:** Tests the replacement of multiple elements in list X to match list Y.
3. **Tuple Sum Equal Case:** Checks the algorithm when the elements have the same sum but are different, requiring replacements.
4. **Combined Operations Case:** Complex scenario testing a mix of delete, insert, and replace operations.
5. **Element Shuffling Case:** Evaluates how the algorithm handles cases where elements in X and Y are the same but in different orders.
6. **End Replace Case:** Focuses on replacing an element at the end of the list.
7. **Beginning Replace Case:** Tests replacing an element at the beginning of the list.
8. **Middle Replace Case:** Assesses the replacement of an element in the middle of the list.
9. **Multiple Insertion to End Case:** Tests adding multiple elements to the end of X to match Y.

6.2 Benchmark Suite B

6.2.1 Benchmarking Process

1. **Random Test Case Generation:** We created random test cases for lists X and Y. The size of these lists varies based on the input parameters (size1 and size2). Each element in these lists is a tuple with random integers, providing a diverse range of test cases.

2. **Performance Measurement:** We have measured the time it takes for our algorithm to compute the minimum amount of updates needed and the operations themselves on these randomly generated test cases.

3. **Benchmarking Different Sizes:** Tests systematically vary the sizes of X and Y through two lists (sizes1 and sizes2). Each size is generated to increase progressively, allowing for the testing of the algorithm's performance across a spectrum of case sizes.

6.2.2 Key Aspects That Are Looked Out For

- Scalability Testing: By increasing the sizes of X and Y, the benchmarking tests how well the algorithm scales with larger inputs.

- Diverse Test Cases: Random generation of test cases ensures a wide range of scenarios, capturing the algorithm's behavior across different kinds of inputs.

- Repeatability: Running multiple test cases for each size combination and averaging the results provide a more reliable measure of performance.